


2012

Detecting exploit patterns from network packet streams

Bibudh Lahiri
Iowa State University

Follow this and additional works at: <https://lib.dr.iastate.edu/etd>

 Part of the [Computer Engineering Commons](#), and the [Computer Sciences Commons](#)

Recommended Citation

Lahiri, Bibudh, "Detecting exploit patterns from network packet streams" (2012). *Graduate Theses and Dissertations*. 12374.
<https://lib.dr.iastate.edu/etd/12374>

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

Detecting exploit patterns from network packet streams

by

Bibudh Lahiri

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:

Srikanta Tirthapura, Co-major Professor

Yong Guan, Co-major Professor

Soma Chaudhuri

Daji Qiao

Aditya Ramamoorthy

Iowa State University

Ames, Iowa

2012

Copyright © Bibudh Lahiri, 2012. All rights reserved.

TABLE OF CONTENTS

LIST OF TABLES	v
LIST OF FIGURES	vi
ACKNOWLEDGEMENTS	ix
ABSTRACT	x
DECLARATION	xi
1. INTRODUCTION	1
1.1 Challenges in Network Monitoring	1
1.2 Data Streams: Model and Algorithms	3
1.3 Denial of Service attacks	7
1.4 Port Scans	9
1.5 NIDS: The Current State of the Art	11
1.6 Thesis Contributions	12
2. Space-Efficient Tracking of Persistent Items in a Massive Data Stream	16
2.1 Introduction	16
2.1.1 Contributions	20
2.1.2 Roadmap	22
2.2 Problem Definition	22
2.2.1 Exact Tracking of Persistent Items	23
2.2.2 Approximate Tracking of Persistent Items	25

2.3	An Algorithm for Approximate Tracking of Persistent Items	25
2.3.1	Fixed Window	26
2.3.2	Sliding Windows	33
2.3.3	Space Complexity	36
2.4	Evaluation	38
2.5	Related Work	54
2.6	Conclusion	56
3.	Identifying Correlated Heavy-Hitters over a Data Stream	58
3.1	Introduction	58
3.1.1	Approximate CHH	60
3.1.2	Contributions	61
3.2	Related Work	63
3.3	Algorithm	64
3.4	Correctness	69
3.5	Analysis	72
3.6	Experiments	75
3.7	Conclusion and Future Work	82
4.	Identifying Frequent Items in a Network using Gossip	83
4.1	Introduction	83
4.1.1	Related Work	86
4.2	Model	87
4.3	Frequent Items with Relative Threshold	90
4.3.1	Analysis	93
4.3.2	Analysis of M^t	93
4.3.3	Analysis of Gossip.	98
4.4	Frequent Items with an Absolute Threshold	103
4.4.1	Analysis	104
4.5	Simulation Results	109

4.5.1	Input Data and Metrics Used.	110
4.5.2	Convergence Time	111
4.5.3	Observations	113
4.6	Synchronous Model	116

LIST OF TABLES

Table 2.1	Distribution of persistence for all datasets	39
-----------	--	----

LIST OF FIGURES

Figure 2.1	CDF of persistence values from 3 windows for the HeaderTrace dataset . . .	43
Figure 2.2	CDF of persistence values from the [1,288] window for the Synthetic1 dataset	43
Figure 2.3	CDF of persistence values from the [1,288] window for the Synthetic2 dataset	44
Figure 2.4	Trade-off between accuracy and space for the small-space algorithm over sliding windows for the HeaderTrace dataset. Each point in each plot is an average from 33 data points - 3 runs over 11 query windows each. Note that the Y-axis is different for each plot. Also, for each value of α , the values of ϵ range from 0.1α to 0.7α	45
Figure 2.5	Trade-off between accuracy and space for the small-space algorithm over sliding windows for the Synthetic1 dataset. Each point in each plot is an average from 30 data points - 3 runs over 10 query windows each. The Y-axis is different for each plot. For each value of α , the values of ϵ range from 0.1α to 0.7α	46
Figure 2.6	Trade-off between accuracy and space for the small-space algorithm over sliding windows for the Synthetic2 dataset. Other details are same as Synthetic1 . .	47
Figure 2.7	The variation of the physical memory taken, the number of true positives, false positives, true negatives and false negatives with ϵ for the Synthetic1 dataset. All the plots are for $\alpha = 0.5$ and the query window [2593, 2880]. So, each point in each plot is an average from 3 data points corresponding to the 3 different seed values (10, 20, 30). Note that the horizontal lines in the three plots represent respectively the actual memory taken by the naive algorithm, the actual number of persistent items and the actual number of transient items, all measured in the same query window, and hence does not vary with ϵ . The Y-axis is different for each plot. The values of ϵ range from 0.1α to 0.7α	51

Figure 2.8 The variation of the physical memory taken, the number of true positives, false positives, true negatives and false negatives with the seed of the random number generator for the **Synthetic1** dataset. All the plots are for $\alpha = 0.5, \epsilon = 0.15$ and the query window $[2593, 2880]$. Note that the horizontal lines in the three plots represent respectively the actual memory taken by the naive algorithm, the actual number of persistent items and the actual number of transient items, all measured in the same query window, and hence does not vary with the seed. The Y-axis is different for each plot. The values of the seed used are 10, 20 and 30. 52

Figure 2.9 The variation of the physical memory taken, the number of true positives, false positives, true negatives and false negatives with the query window for the **Synthetic1** dataset. All the plots are for $\alpha = 0.5, \epsilon = 0.15$ and seed = 10. Note that the horizontal lines in the three plots represent respectively the actual memory taken by the naive algorithm, the actual number of persistent items and the actual number of transient items - the first one shows slight increase with the progress of time (increasing query window number) but the other two are practically constant. The Y-axis is different for each plot. The query windows are $[1, 288], [289, 576], \dots, [2593, 2880]$ and the values on the X-axis are the endpoints of the query windows. 53

Figure 3.1 On the X-axis are the ranks of the eight (heavy-hitter) destination IPs, that co-appear with maximum number of distinct source IPs. For each destination IP, the Y-axis shows 1) the number of distinct source IPs co-occurring with it, 2) the number of heavy-hitter destination IPs co-appearing with it. Note that the Y-axis is logarithmic. 76

Figure 3.2 Comparison of space (left) and time (right) costs of the naive and the small-space algorithms. The space is the total number of distinct tuples stored, summed over all distinct destination IP addresses. The time is the number of hours to process the 248 million records. Note that the Y-axis for the left graph is logarithmic. 77

Figure 3.3	Error statistic in estimating the frequencies of the heavy-hitter destination IPs in “IPPair”. The graph shows the theoretical maximum ($\frac{1}{s_1}$), the experimental maximum and the experimental average.	78
Figure 3.4	Error statistic in estimating the frequencies of the CHH source IPs in “IPPair”, for $s_2 = 1100, 1500$ and 2000 respectively. The graph shows the theoretical maxima ($\frac{1}{\phi s_1} + \frac{1}{s_2}$), the experimental maxima and the experimental average.	79
Figure 3.5	Error statistic in estimating the frequencies of the heavy-hitter destination ports from “PortIP”	80
Figure 3.6	Error statistic in estimating the frequencies of the CHH destination IPs in “PortIP”. The three graphs are for $s_2 = 1100, s_2 = 1500$ and $s_2 = 2000$ respectively.	81
Figure 4.1	The number of rounds till convergence versus network size N	112
Figure 4.2	The error rate as a function of the sketch size for the relative error algorithm, with the dataset generated by the Pareto-like distribution. $\phi = 0.081, \psi = 0.02$ and $\delta = 0.1$	113
Figure 4.3	The error rate as a function of c_a , a multiplier in the sampling probability, for the absolute error algorithm. The dataset is generated by the Pareto-like distribution. Note that the expected sketch size increases linearly with the sampling probability. $\delta = 0.1$	114
Figure 4.4	The error rate as a function of the sketch size for the relative error algorithm, with the dataset generated by the mixed distribution.	115
Figure 4.5	The error rate as a function of c_a , a multiplier in the sampling probability, for the absolute error algorithm. The dataset is generated by the mixed distribution.	115

ACKNOWLEDGEMENTS

I would like to thank my major professor Dr. Srikanta Tirthapura for

- his excellent course on randomized algorithms. I literally looked forward to going to this class.
- teaching me how to extract the gist of things and not to get overwhelmed by details
- his emphasis on quality work
- teaching me to ask the right questions, which has earned me appreciation later on a few occasions

I would also like to thank my co-major professor Dr. Yong Guan and my committee members, Dr. Soma Chaudhuri, Dr. Daji Qiao and Dr. Aditya Ramamoorthy, for their advice in this work; my high school history teacher, Mr. Asit Chatterjee, for being a mentor throughout my life so far and providing me with valuable wisdom; my parents, Dr. Balendra Nath Lahiri and Mrs. Manjari Lahiri, my sister Baishali and my wife (and fellow graduate student) Madhumita for standing besides me through thick and thin. I am also grateful to my colleagues Dr. Bojian Xu and Dr. Zhenhui Shen for their help and advices during and after my stay in Iowa; and to my colleagues Dr. Fabian Moerchen and Dr. Ioannis Akrotirianakis, who worked with me on [78] during my internship at Siemens Corporate Research, and from whom I learnt how to make machine learning algorithms useful in practical problem solving; and to my research collaborator Dr. Jaideep Chandrashekar for his valuable inputs during our collaborative work on [79].

ABSTRACT

Network-based Intrusion Detection Systems (NIDS), e.g., Snort, Bro or NSM, try to detect malicious network activity such as Denial of Service (DoS) attacks and port scans by monitoring network traffic. Research from network traffic measurement has identified various patterns that exploits on today's Internet typically exhibit. However, there has not been any significant attempt, so far, to design algorithms with provable guarantees for detecting exploit patterns from network traffic packets. In this work, we develop and apply data streaming algorithms to detect exploit patterns from network packet streams.

In network intrusion detection, it is necessary to analyze large volumes of data in an on-line fashion. Our work addresses scalable analysis of data under the following situations. (1) Attack traffic can be stealthy in nature, which means detecting a few covert attackers might call for checking traffic logs of days or even months, (2) Traffic is multidimensional and correlations between multiple dimensions maybe important, and (3) Sometimes traffic from multiple sources may need to be analyzed in a combined manner. Our algorithms offer provable bounds on resource consumption and approximation error. Our theoretical results are supported by experiments over real network traces and synthetic datasets.

DECLARATION

Publications: The work presented in this thesis has been published in the following conference proceedings and journals:

- The content of Chapter 2 has been published in [79].
- The content of Chapter 3 was published in [81].
- The content of Chapter 4 first got published in [80], and later in [82].

My Contributions: This thesis is the result of collaborative work with my advisor Dr. Srikanta Tirthapura and my research collaborator Dr. Jaideep Chandrashekar, then with Intel Labs Berkeley. My contributions are as I describe below:

- The problem of detecting temporally persistent items, in the context of network intrusion detection, was addressed by our collaborator Dr. Jaideep Chandrashekar in his prior work [59]. The mathematical formulation of the problem as in our work [79] and the initial intuition behind the algorithms are due to my advisor Dr. Tirthapura. Dr. Tirthapura and I worked jointly on the theoretical proofs. The experiments, including the design of the synthetic dataset, are due to myself.
- I formulated the problem of detecting correlated heavy-hitters before my PhD preliminary examination. I worked with Dr. Tirthapura to conceive the idea of the algorithm and do the theoretical proofs. The experiments, including the collection of the dataset and choosing the tools, are due to myself.
- The formulation of the problem of detecting heavy-hitters from a distributed dataset, and the theoretical proofs, are result of joint work by Dr. Tirthapura and myself. The experiments, including the design of the synthetic dataset, are due to myself.

CHAPTER 1. INTRODUCTION

An Intrusion Detection System (IDS) is a piece of software or hardware designed to detect unwanted attempts at accessing, manipulating and disabling of computer systems through a network such as the Internet. An Intrusion Detection System can be network-based, protocol-based or host-based. Network-based Intrusion Detection Systems (NIDS), e.g., Snort [99], Bro [95] or NSM [67], try to detect malicious activity such as Denial of Service (DoS) attacks [91], port scans [102] or even attempts to gain access to computers by monitoring network traffic. The network traffic measurement researchers have identified various patterns that the typical exploits on today's Internet (e.g., DoS attacks, port scans, worms) exhibit [113, 114]. However, there has not been any significant attempt, so far, to design algorithms - with theoretical guarantees on the space and/or time requirement, or the extent of approximation - for detecting these known exploit patterns from network traffic packet data and applying them in NIDSs. We observe that data stream algorithms, which compute various aggregates from massive data streams online and in small space, can be applied to detect such exploit patterns from network packet data. The goals of this research are (1) formalization of the notion of exploit patterns and (2) design and analysis of efficient algorithms for detecting these patterns from network packet streams.

1.1 Challenges in Network Monitoring

The Internet consists of routers connected to each other that forward IP packets. Traffic at the routers may be viewed at several levels of abstraction [93, 56].

1. **Packet logs:** Each TCP segment (or UDP datagram) has a header that has source and destination ports, and the IP packet that the TCP segment (or UDP datagram) envelops

contains the source and destination IP addresses in its header. This can be collected at switches, routers or network taps by tools like tcpdump [1] or Wireshark [2].

2. **Flow logs:** Each flow is a collection of packets with same values for certain key attributes such as the source and destination IP addresses. For each flow, the log contains cumulative information about number of bytes and packets sent, start and end times, protocol types etc. This is typically collected at border routers or taps by tools like FlowScan [96], YAF [3] or Argus [4].
3. **Traffic counters:** A traffic counter keeps track of the number of bytes sent over each link every few minutes. An example is MRTG logs [5], which record the volume of traffic through SNMP-enabled devices.

It is most valuable to analyze flow logs and packet logs, because they contain maximum information. However, this, at the same time, implies that we should be prepared to deal with enormous volume of data. For example, the backbone of a typical regional ISP today is a OC-192 network line with a transmission speed of 10 Gbits/sec, of which the packet overhead part alone is 332 Mbits/s! Given the *volume* and the *short-lived* nature of this data, it is almost impossible to store these data on a hard disk. Cisco routers that have IOS NetFlow [6] or CS-MARS (Cisco Security Monitoring, Analysis, and Response System) [38] feature enabled aggregate packets to flows and generate NetFlow records; these are exported to analyzing software like NetFlow Collector (NFC) [7]. The potential devices where such monitoring algorithms would be deployed are gateway routers (e.g., Cisco ASR 1000). Considering a typical Cisco home router has 32-64 MB RAM and 4-8 MB Flash memory, and a typical desktop computer (where a product like NetFlow Collector might be installed) or a gateway router has 6-8 GB of RAM, none of these devices are able to store such network data streams in their entirety in main memories. Any analysis on the data has to be performed *online*, i.e., accepting the fact that we get to see each data item only once.

1.2 Data Streams: Model and Algorithms

A *data stream* is an abstract model for applications where data is generated continuously (e.g., stock quotes, network flows, call records in a telephone exchange, high-energy particle physics experiments).

Definition 1.2.1 A data stream $A = (a_1, a_2, \dots, a_m)$ is a sequence of elements, where each a_i is a member of $[n] = \{1, 2, \dots, n\}$.

The sheer volume and transience of data streams forces us to perform any computation on the data in a single pass and using limited memory. These constraints have motivated the emergence of a class of data structures called *sketches*, defined as follows:

Definition 1.2.2 A sketch is a data structure with the following properties:

- **Property 1.2.1** A sketch requires small space, typically polylogarithmic in the size of the stream, or the size of a subset of the stream we are interested in
- **Property 1.2.2** It can be updated, in constant or polylogarithmic time, as the elements of the stream are received
- **Property 1.2.3** The aggregate that we want to compute on the stream can be computed approximately based on the sketch
- **Property 1.2.4** Some applications, with distributed streams (to be defined later), requires the following property: if a separate sketch is maintained for each of two or more streams, then the combination of the sketches should be able to answer the desired aggregate on the union of the streams, with guaranteed accuracy

A sketch can be a simple uniform or weighted random sample of the stream elements, or a projection along random vectors, or any other transformation that satisfies the above properties.

We now present some basic research findings in data streams. Let $m_i = |\{j : a_j = i\}|$ denote the number of occurrences of i in the sequence A . For each $k > 0$, a useful statistics of

the sequence is the k^{th} *frequency moment*, defined as $F_k = \sum_{i=1}^n m_i^k$. In particular, F_0 is the number of distinct elements in the sequence, $F_1 (= m)$ is the length of the sequence, and F_2 is known as the “surprise index”.

In a seminal work, Alon *et al* [13] analyzed the space complexity of computing the frequency moments. They presented lower bounds showing that an exact computation of F_k , or even an accurate deterministic approximation of it, requires $\Omega(n)$ space, in the worst case. However, a randomized approximation to F_k (for $k \geq 2$) can be found as follows. First, choose a random element a_p from A . Then maintain the count $X = |\{q : q \geq p, a_q = a_p\}|$. In other words, count the number of reoccurrences of the element a_p in the portion of the stream that succeeds a_p (including a_p). Then, the random variable $Y = m[X^k - (X - 1)^k]$ is an unbiased estimator of F_k , i.e. $E[Y] = F_k$. Further, it can also be shown that the variance of Y is small. This approach is termed as “sample and count”, and we show, in Chapter 2, how a variant of this approach can be used for a different problem. Alon *et al* [13] also proved that F_0 , F_1 and F_2 can be approximated in logarithmic space, whereas the approximation of F_k for $k \geq 6$ requires $n^{\Omega(1)}$ space. Of these, the problem of estimating F_0 has drawn significant attention of the researchers, and has been addressed by Flajolet and Martin [51] and Gibbons and Tirthapura [57].

The problem of identifying the frequently occurring items [34, 87, 89] (often termed “heavy-hitters”) from a stream has also been studied quite thoroughly. For any user-input threshold $\phi \in (0, 1)$, Misra and Gries [89] came up with a deterministic algorithm to find the data items that occur more than ϕm times in an array of size m . Their algorithm required $O(m \log \frac{1}{\phi})$ time and $O(\frac{1}{\phi})$ space for the sketch. The problem with their algorithm was that it was *two-pass* - the first pass could identify the candidates for frequent items, and kept track of the counts of these elements; and in the second pass, one had to eliminate, from these candidates, the ones that were not actually frequent. However, with minor modifications of the original algorithm, and a little sacrifice in precision, it is possible to come up with a *single-pass, approximate* variant of the algorithm that provides the following approximation guarantees, for some user-input threshold ϕ and approximation error $\epsilon < \phi$ (note that for an *online* algorithm, m is the number of elements received *so far*) :

- All items whose frequencies exceed ϕm are output. There are no false negatives.
- No item with frequency less than $(\phi - \epsilon)m$ is output.
- Estimated frequencies are less than true frequencies by at most ϵm .

The algorithm is simple: it maintains an associative array (of size at most $\frac{1}{\epsilon}$) of (value, count) pairs. On receiving each item a_i , we check whether the value a_i is already in the associative array. If it exists, we increment its count by 1; otherwise, we add the pair $(a_i, 1)$ to the array. Now, if adding a new pair to the array makes its size exceed $\frac{1}{\epsilon}$, then for each of the (value, count) pairs in the array, we decrement the count by one; and throw away any value whose count falls to zero after decrement. Note that this ensures at least the element which was most recently added (with a count of one) would get discarded, so the size of the array, after processing all pairs, would come down to $\frac{1}{\epsilon}$ or less. Thus, the space requirement of this algorithm is $O(\frac{1}{\epsilon})$. In Chapter 3, we apply an extension of this idea to identify correlated heavy-hitters from multidimensional streams.

In data stream applications, often, the more recent an item is, the more is our interest in it. So, a popular model for studying data streams is the *sliding-window* model, where we focus on computing the aggregates on the last N items of the stream, using $o(N)$ space. Datar *et al* [40] solved the *basic counting* problem in the sliding-window model: given a stream of bits, they came up with an ϵ -approximate algorithm for counting the number of 1's among the last N bits, using $O(\frac{1}{\epsilon} \log^2 N)$ bits of memory. Their algorithm processed each item in $O(1)$ amortized and $O(\log N)$ worst-case time. They also extended their algorithm to maintain the sum of last N elements, with a relative error of at most ϵ , in a stream of positive integers in the range $[0..R]$. This algorithm needed $O(\frac{1}{\epsilon}(\log N + \log R)(\log N))$ memory bits. The arrival of each new element was processed in $O(\frac{\log N}{\log R})$ amortized time and $O(\log N + \log R)$ worst-case time. Gibbons and Tirthapura [58] came up with improved results for the same problem, using a novel data structure called the *wave*. For the basic counting problem, they improved the per-item processing time to $O(1)$ in worst case. For the sum problem, they improved the worst-case per-item processing time to $O(1)$. We use the sliding window model for a different problem in Chapter 2.

With the emergence of huge networks, today's applications often collect data not from a single source, but from distributed sources. In such a scenario, we are interested in computing aggregates over the *union* of the streams emerging from different sources. As we have already discussed, network monitoring devices observe streams of packets. Each device has a small workspace in which to store information on its observed stream, and the contents are periodically sent to a central data analyzer, in order to compute aggregated statistics on the streams. Some existing network monitoring tools, e.g., Lucent's InterpretNet and some products implementing Cisco's NetFlow protocol, use this mechanism for traffic monitoring.

As Gibbons and Tirthapura [57] pointed out, there is a subtle difference between the distributed streams model and the *merged* streams model. In the latter model, there is only one party who observes both streams, and the streams are interleaved in an arbitrary order by an adversary. In the distributed streams model, each party observes its own stream, and computes the sketch on it. The sketches have the property that when combined, they can give approximate answers to queries over the unions of all the streams. Gibbons and Tirthapura [57] showed that for $t > 2$ streams and for any function f , the deterministic merged stream complexity (i.e., space bound) is within a factor of t of the deterministic t -party distributed stream complexity. It followed that deterministic merged streams algorithms can be designed assuming that the streams are not interleaved, at a penalty of at most t .

Our study of the intrusion detection literature reveals that most of the signature-based NIDS tools (e.g. Snort [99]), work by checking the packet payloads for signatures of well-known attacks. This is not a very scalable method as the attacker can evade detection with minor changes in the signature, but can still amount the same damage to the victim. Also, the signature set can grow very large, e.g., the signature set of Snort contains 3,400 distinct signatures. Matching these signatures sequentially with the packet payload is very expensive. Among the anomaly-based intrusion detection tools, PAYL [111] compares the frequency distributions byte contents of training and test datasets to detect anomalous payloads. However, Kolesnikov and Lee [76] and Fogla *et al* [52] showed that detection by PAYL and other anomaly detection systems ([77, 104]) could be evaded by specially-crafted polymorphic blending attacks. Moreover, none of these approaches paid much attention to the scalability issue.

We found that in the area of network traffic measurement, there has been systematic study of real network traffic data to identify various patterns of the typical exploits on today's Internet (e.g., DoS attacks, port scans, worms) [113, 114, 115]. Xu *et al* [113, 114] applied data mining and information-theoretic techniques to automatically extract useful information from largely unstructured data. This shows that exploit patterns can be detected by computing summary statistics over large volumes of data, rather than inspecting the payload of each and every packet for an exact or partial signature match. We observed that DoS attacks [91] and port scans [102] are two types of attacks on the Internet where data-driven approaches can play major roles in identifying potential threats, and hence discuss them in detail in the following sections.

1.3 Denial of Service attacks

A DoS attack is an attempt to make a computer resource (e.g., a website or a database server) unavailable to its intended users. The methods generally involve saturating the target machine with external communications requests, such that it cannot respond to legitimate traffic, or responds so slowly as to be rendered effectively unavailable; or, saturating some other resource, such as a system buffer.

In February 2000, a series of massive DoS attacks incapacitated several high-visibility Internet e-commerce sites, including Yahoo, Ebay, and E*trade. Next, in January 2001, the name server infrastructure of Microsoft was disabled by a similar assault. Many other domestic and foreign sites have also been victims, ranging from smaller commercial sites, to educational institutions, public chat servers and government organizations. Moore *et al* [91] monitored a lightly utilized network, comprising 2^{24} distinct IP addresses, over a period of 25 days in February 2001, and observed 12,805 attacks on over 5,000 distinct Internet hosts belonging to more than 2,000 different organizations during this period. The following few types of DoS attacks are most common:

SYN Flood: This is perhaps the most common and widely studied form of DoS attack. When a client attempts to start a TCP connection to a server, the client and server exchange a series of messages which normally runs like this:

1. The client requests a connection by sending a SYN (synchronize) message to the server.
2. The server acknowledges this request by sending SYN-ACK back to the client.
3. The client responds with an ACK, and the connection is established.

This is called the *TCP three-way handshake*, and is the foundation for every connection established using the TCP protocol. The SYN Flood attack works if a server allocates resources for the connection after receiving a SYN, but before it has received the ACK. The basic mechanism to launch a SYN flood is the following: if half-open connections bind resources on the server, it may be possible to take up all these resources by flooding the server with SYN messages. Once all resources set aside for half-open connections are reserved, no new connections (legitimate or not) can be made, resulting in denial of service. There are the two following methods to launch a SYN flood attack - both involve the server not receiving the ACK.

- A malicious client can skip sending this last ACK message.
- By falsifying the source IP address in the SYN, it can make the server send the SYN-ACK to the falsified IP address, and thus never receive the ACK.

UDP Flood: A UDP flood attack can be initiated by sending a large number of UDP packets to random ports on a remote host. Since the target host finds (with high probability) that no application is listening at that port, it replies with an “ICMP Destination Unreachable” packet. Thus, for a large number of UDP packets, the victim host is forced to send many ICMP packets, eventually leading it to be unreachable by other clients.

There are some other forms of DoS attacks (e.g., Smurf attack, Ping flood), the underlying idea behind all of them being the same: the attacker pretends to be a benign host, and initiates some form of communication request with the victim, the scale of the communication being large enough to eventually exhaust all the system resources of the victim. A variant of DoS attacks worth mention here is the Distributed DoS (DDoS) attack, where instead of a single attacker, multiple compromised systems flood the bandwidth or resources of a targeted system. The attack can be initiated by a single attacker, who compromises other systems to launch an attack on the end victim(s) on a large scale. Note that if we treat the traffic flow of a

single compromised system as a data stream, then the distributed streams model [57, 58] can be useful in mining patterns from the union of these streams.

A DoS attack, or at least an attempt to launch one, can be detected by applying heavy-hitter algorithms on traffic flow data. Where the frequent sources can be the potential attackers, the frequent destinations can be the potential victims. In Chapter 3, we discuss our work on an extension of the heavy-hitter problem [81] to design more informative sketches for detecting the attackers and victims in a DoS attack under certain conditions. In Chapter 4, we discuss some sketches we developed [80, 82] for identifying heavy-hitters from the union of multiple data streams, which can be used for detecting DDoS attacks.

1.4 Port Scans

Some applications, running on specific ports, have some known vulnerabilities, and sometimes the network-based exploits take advantage of these vulnerabilities. The exploit traffic is hence directed towards this port, e.g., the W32/Blaster worm took advantage of a buffer overflow in the Microsoft DCOM RPC locator service, an application that runs on TCP port 135, to create a SYN flood. *Port scanning* is a technique to search a network host for *open* ports, i.e., ports where a deployed application with known vulnerabilities is looking for an incoming connection.

Jung *et al* [69] pointed out that a number of difficulties arise when we attempt to formulate an effective algorithm for detecting port scans. The first is that there is no crisp definition of the activity, e.g., an attempted HTTP connection to the main web server of a site should not raise an alarm. However, whether a sweep through the entire address space looking for HTTP servers should concern us depends on what *intent* the sweep is being done with, e.g., some search engines not only follow embedded links but also scan ports in order to find web servers to index. In addition, some applications (e.g., SSH, some P2P and Windows applications) have modes in which they scan in a benign attempt to gather information or locate servers. Ideally, we would like to separate out such benign use from overtly malicious one. We would note, however, that the question of whether scanning by search engines is benign will ultimately be a *policy* decision that will reflect the sites view of the desirability to have information about its

servers publicly accessible.

Since we never know whether the intent behind a port scan is harmful or not, being able to detect port scans is important from a security perspective. If we are allowed to make multiple passes over data gathered from packet/flow logs, then port scans can surely be detected. However, detecting them online and using small space presents the usual challenge of network monitoring.

We present a classification of port scans from [102]:

1. **Vertical Scan:** A sequential or random scan of multiple ports of a single IP address from the same source in a given time window. These are usually an attempt to survey which of several well known vulnerabilities applies to this host.
2. **Horizontal Scan:** A scan from a single source of several machines in a subnet aimed at the same target port, i.e., the same vulnerability. In this case the attacker is searching for any machine that is running specific service and does not care about any single machine in particular.

Just like DoS attacks can be distributed ones, horizontal or vertical port scans can also be launched by multiple sources working in tandem (sometimes referred to as “Coordinated Scans” in the literature). However, horizontal or vertical port scans can be easily detected offline from packet/flow logs, unless the scan is a *stealth scan*. A stealth scan is initiated with a very low frequency to avoid detection. The key parameters in the definition of stealth scan include the maximum threshold and the minimum threshold for the average interscan distance. An average interscan distance below the minimum threshold indicates that the scan was not stealthy, i.e., not intended to evade NIDS systems. Two successive scans from the same source that are separated by more than the maximum interscan distance are considered to be unrelated or parts of different scanning episodes. We will discuss in Chapter 2 how we can formalize the notion of “persistence” [79] of data items to detect sources that launch stealthy port scans.

1.5 NIDS: The Current State of the Art

Having discussed two major exploit patterns, DoS attacks and port scans, we now move on to discuss some of the measures that have been adopted so far to address these problems. Traditionally, intrusion detection tools are classified into two broad categories: *signature-based* and *anomaly-based* [66]. Signature-based NIDSs aim to detect well-known attacks as well as slight variations of them, by identifying the *signatures* that characterize these attacks. Due to its nature, a signature-based NIDS has *low false positives* but it is unable to detect any attacks that lie beyond its knowledge. An anomaly-based NIDS is designed to capture any deviations from the established profiles of users and the normal behavior patterns of systems. Although in principle, anomaly detection has the ability to detect new attacks, in practice this is far from easy. Anomaly detection has the potential to generate too many false alarms, and it is very time consuming and labor-expensive to sift true intrusions from the false alarms.

Historically, most signature-based NIDS tools function by detecting N or more events in a time window of T seconds. Network Security Monitor (NSM) [67] was the first NIDS to work on such algorithm. It had rules to detect any source IP address connecting to more than 15 distinct destination IP addresses within a given time window. Similarly, Snort [99], a tool developed later by Martin Roesch, checks whether a given source IP address connected to more than X number of ports or more than Y number of destination IP addresses within Z seconds, where X , Y , Z are configurable parameters. However, Bro [95], another popular NIDS, worked on the observation that *failed* connection attempts are *better* indicators for identifying port scans. Since scanners have little knowledge of network topology and system configuration, they are likely to often choose an IP address or port that is not active. The algorithm provided by Bro treated connections differently depending on their services (application protocols). For connections using a service specified in a configurable list (e.g., HTTP, SSH, SMTP etc), Bro only performs bookkeeping if the connection attempt failed (was either unanswered, or elicited a TCP RST response). For others, it considers all connections, whether or not they failed. It then tallies the number of distinct destination addresses to which such connections (attempts) were made. If the number reaches a configurable parameter N , then Bro flags the source

address as a scanner.

The developers of the NIDS tools focussed on building full-fledged applications that can monitor network traffic, generate logs and report incidents (the definition of “incident” depends on the policy of the user, and is configurable within the tool). While these tools are lightweight and (mostly) open-source, and some detailed documentation regarding their technical architectures are available [46], it seems that the overall approach is *not* very *scalable*. Most NIDSs look for an *exact* match for a signature, which is a sequence of bytes, in the payloads of packets. However, when using tight signatures, the matcher has no capability to detect attacks other than those for which it has explicit signatures; the matcher will in general completely miss novel attacks, which, unfortunately, continue to be developed at a brisk pace. Also, the total number of patterns contained in the signature set of a NIDS can be quite large, hence matching them sequentially is time-wise expensive. Although Gonzalez and Paxson [61] came up with some sampling-based modifications of Bro to identify the heavy-hitters from the traffic streams, there is no published literature on their algorithms with theoretical guarantees, and there is certainly scope for identifying more such aggregates from traffic streams. Gu *et al* [63] attempted to create a comprehensive information-theoretic framework for analyzing intrusion detection systems, but even their work does not address the computational resource issues like memory or CPU time, unlike our work.

1.6 Thesis Contributions

There is no silver-bullet solution to network intrusion detection, and we do not claim that our algorithms can be immediately converted to ready-to-deploy network monitoring tools. Lately, there have been significant attempts to apply machine learning algorithms, particularly classification techniques, to identify network traffic as benign or harmful. Recently, Sommer and Paxson [101] did a comprehensive critique of the machine-learning based approaches in the network intrusion detection domain; one of their main arguments was that machine-learning algorithms are more effective in classifying objects based on “learning” from past (labeled) data, rather than identifying outliers, as the anomaly detection systems need [8].

We focus on detecting a set of patterns from network packet streams that have repeatedly

been shown to be characteristic of network intrusions, or attempts thereof. The major limitations of the current NIDSs are that while signature-based systems fail to detect unforeseen attacks, anomaly-based systems are very likely to generate false alarms. However, even a low false positive rate can make an NIDS ineffective [18], while undetected attacks can cause major damage to a network. Also, as Axelsson [18] pointed out, applying a classification-based approach for the intrusion detection problem is hard because of the base-rate fallacy - the number of packets that contain traffic from malicious sources is typically a very small fraction of the number of packets observed in a monitor over a given period of time.

The common theme underlying the work presented in Chapters 2, 3 and 4 are the following:

- While most of the research in intrusion detection focused on improving precision and recall, our work in Chapters 2, 3 and 4 address the challenges presented by limitation of computational resources (memory, CPU time). They also mathematically establish the trade-off between the amount of resource (mostly memory) used and the accuracy of results. Usually, *both* the precision and the recall improve as the space budget is increased, unlike many machine learning applications where there is a trade-off involved *between* the precision and the recall [14].
- In network intrusion detection, scalability can be an important issue because of the following reasons: (1) attack traffic can be stealthy in nature, which means detecting a few covert attackers might call for checking traffic logs of days or even months, (2) we may need to analyze the traffic data along multiple dimensions, or (3) we may need to analyze the traffic data from multiple sources. We address those issues in Chapters 2, 3 and 4 respectively.
- Our algorithms (and data stream algorithms, in general) in Chapters 2, 3 and 4 offer provable bounds on the probabilities of generating false positives and false negatives, and/or the errors made in estimating quantities like frequency and temporal persistence.

We design space and/or communication-efficient algorithms for detecting such patterns online; present lower-bound proofs, wherever appropriate, for showing some problems are intrinsically hard; and demonstrate the practical viability of our algorithms with experiments on

real and synthetic datasets. While our motivation of working on these data stream problems came from the network security domain, these are novel problems in the literature of streaming algorithms; particularly, our work on persistent items in Chapter 2 shifts the focus from the oft-discussed frequency distribution in streams to the temporal dynamics of items appearing in a stream.

Here are the principal contributions of this thesis:

We formalize the notion of temporally persistent items occurring in a data stream. Unlike frequent items or heavy-hitters which have drawn significant attention in data stream research, persistent items do not necessarily contribute a lot to the volume of incoming traffic towards a potential victim, since the attacker attempts to conduct the probe in the stealth mode. However, the (source) IP address or port number of the attacker has to show up in a sufficiently large number of distinct timeslots in order for the attack to be effective. We call such items (IP addresses or port numbers) as “temporally persistent”. Our contributions are: (1) We show that any algorithm that tracks *all* temporally persistent items in the stream must take space superlinear in the number of distinct items, which is prohibitively expensive. (2) We design a “sketch”, a hash-based technique of creating a summary of the data stream, to approximately identify persistent items from an entire stream (“fixed window”). (3) We also extend the sketch for an alternative model where the persistent items are tracked over a sliding window of recent timeslots, and this sketch takes space that is (on expectation) within a factor of two of that taken by the sketch for the fixed window version. Both sketches detect all persistent items *with high probability*, and do not report any items, provably, whose persistence fall below a threshold. (4) We experimented with three different datasets (one real and two synthetic) to see how the accuracy and memory footprint of the algorithm varies with the skewness of the dataset. Our algorithms performed best for the two datasets out of three which had highest skewness of persistence and lowest mean persistence. (5) Our experiments also show that typically the persistence of IP addresses in real network traffic traces have a very skewed distribution, which works to the advantage of our algorithm since we save space at the cost of *not* storing items with very low persistence.

We consider online mining of correlated heavy-hitters (CHH) from a data stream, i.e.,

queries of the following form: “In a stream S of (x, y) tuples, on the substream H of all x values that are heavy-hitters, maintain those y values that occur frequently with the x values in H ”. In intrusion detection, this sort of query will be useful in multidimensional analytics of the attack traffic, e.g., in a Denial of Service (DoS) attack, while the frequent sources detected at a large ISP backbone router (like the Cisco 12000 Series [9]) may identify the attackers, the frequent destinations in the sub-stream of a single attacker may identify the potential victims. We advance the state of the art by (1) formulating an approximate version of the CHH problem, (2) designing a sketch for approximately tracking of CHHs, with provable guarantees on the maximum error estimates, (3) theoretically deriving the minimum space requirement for our sketch under constraints imposed by our problem formulation, and (4) conducting experiments that demonstrate the space-accuracy trade-off on a large stream of IP packet headers from a backbone network link.

We present algorithms for identifying frequently occurring items in a large distributed data set. In the parlance of network exploit pattern detection, this will be useful if a group of coordinated attackers send a lot of traffic to a victim to launch a Distributed Denial of Service (DDoS) attack. The IP address of the victim may not be frequent in the packet stream coming from a single source, but may be frequent if the union of the packet streams from all the sources are considered. However, sending all packet streams to a central aggregator is neither a scalable nor a fault-tolerant solution. (1) Our algorithms use gossip [42] as the underlying communication mechanism, since gossip does not rely on any central control, or on an underlying network structure, such as a spanning tree. If this process continues for a (short) period of time, the desired results are computed, with probabilistic guarantees on the accuracy. (2) Our algorithm for identifying frequent items is built by layering a novel small-space “sketch” of data over a gossip-based data dissemination mechanism. We prove that the algorithm identifies the frequent items with high probability, and provide bounds on the time till convergence. (3) We experiment with our algorithms on two different synthetic datasets, generated from different distributions, to show that both the sketch size and the number of rounds of gossip needed till convergence, can in practice be kept orders of magnitude lower than what the theoretical analyses demand.

CHAPTER 2. Space-Efficient Tracking of Persistent Items in a Massive Data Stream

Motivated by scenarios in network anomaly detection, in this chapter, we consider the problem of detecting persistent items in a data stream, which are items that occur “regularly” in the stream. In contrast with heavy-hitters, persistent items do not necessarily contribute significantly to the volume of a stream, and may escape detection by traditional volume-based anomaly detectors.

We first show that any online algorithm that tracks persistent items exactly must necessarily use a large workspace, and is infeasible to run on a traffic monitoring node. In light of this lower bound, we introduce an approximate formulation of the problem and present a small-space algorithm to approximately track persistent items over a large data stream. We experimented with three different datasets to see how the accuracy and memory footprint of the algorithm varies with the skewness of the dataset. Our algorithms performed best for the two datasets out of three which had highest skewness of persistence and lowest mean persistence. To our knowledge, this is the first systematic study of the problem of detecting persistent items in a data stream, and our work can help detect anomalies that are temporal, rather than volume based.

2.1 Introduction

We consider the problem of tracking *persistent* items in a large data stream. This problem has particular relevance while mining various network streams, such as the traffic at a gateway router, connections to a web service, etc. Informally, a persistent item is one that occurs “regularly” in the stream.

More precisely, suppose that the time at the stream processor is partitioned into non-overlapping intervals called “timeslots”. Consider a stream of elements of the form (d, t) where d is an item identifier, and t is a timeslot during which the item arrived. The t values are in an increasing order within the stream. Multiple items can arrive in the same timeslot, and the same item may arrive multiple times within a time slot. Suppose the total number of timeslots in the stream is n . The persistence of an item d is defined to be the number of distinct timeslots in which d was observed. The persistence of any item is an integer between 0 and n (inclusive). An item is said to be α -persistent, for some constant $0 < \alpha \leq 1$, if its persistence is at least αn . Given a user-defined α , the problem is to output the set of α -persistent items in the stream.

Persistent items exhibit a repeated and regular pattern of arrival, and are significant for many applications. Giroire *et al.* [59] monitored traffic from end-hosts to detect communication across botnet channels. They observed that persistent destinations were likely to belong to one of two classes: (1) either they were malicious hosts associated with a botnet, or (2) they were frequently visited benign hosts. It was also observed that the latter set of hosts could be identified easily and assembled into a “whitelist” of known good destinations. They found that tracking persistent items in the network stream, followed by filtering out items contained in the whitelist, resulted in reliable identification of botnet traffic.

More broadly, persistent items are often associated with specific anomalies in the context of network streams: periodic connections to an online advertisement in a pay-per-click revenue model [107] is an indicator of click fraud [117], repeated (failed) connections observed in the stream is indicative of a failed or unreachable web service [64]; botnets periodically “phone home” to their bot controllers [59]; attackers regularly scan for open ports on which vulnerable applications are usually deployed [102]. While the narrative in this paper draws from applications in the network monitoring space, it appears that the problem of detecting persistent items in a data stream is broadly applicable in other data monitoring applications. For example, persistent use of gathering techniques such as telephone interception or satellite imaging might indicate an “Advanced Persistent Threat” (APT) [105] for a target group, e.g., a government.

The persistent items in a stream could be very different from the frequently occurring items (or “heavy-hitters”) in a stream. An item is called a ϕ -heavy hitter if it contributes to at

least a ϕ fraction of the entire volume of the stream. There is a large body of literature on heavy-hitter identification (including [89, 87, 50, 29, 33, 36]). A persistent item need not be a heavy hitter. For example, the item may appear only once in each time slot and may not contribute significantly to the stream volume. Such “stealthy” behavior was indeed observed in botnet traffic detection [59]; the highly persistent destinations which were not contained in the whitelist did not contribute in any meaningful way to the traffic volume. In fact, the traffic to these destinations was stealthy and very low volume, perhaps by design to evade detection by traditional volume-based detectors. Conversely, a heavy-hitter need not be a persistent item either – for example, an item may occur a number of times in the stream, but all its occurrences may be within only a couple of timeslots. Such an item will have a low persistence. Clearly, the set of persistent items in a stream can be very different from the set of heavy-hitters in the stream; their intersection can very well be empty. There seems to be no easy reduction from the problem of tracking persistent items to the problem of tracking heavy-hitters. For example, one could attempt to devise a “filter” that eliminated duplicate occurrences of an item within a time slot, and then apply a traditional heavy-hitter algorithm on the resulting “filtered” stream. But this approach does not work in small space, because such a filter would itself take space proportional to the number of distinct items that appeared within the timeslot, and this number may be very large, especially for the type of network traffic streams that we are interested in.

A closely related problem is the problem of identifying *heavy distinct hitters (HDHs)* in a data stream (Venkataraman *et al.*[110] and Bandi *et. al.* [20]). In the heavy distinct hitters problem, we are given a stream S' of (x, y) pairs, of length N . For a parameter $\beta, 0 < \beta \leq 1$, the set of β -HDHs in S' is defined as the set of all those values of x that have occurred with more than $N\beta$ distinct values of y . There is a reduction from the problem of tracking persistent items to that of identifying HDHs, as follows. Consider the identification of α -persistent items on a stream S of (d, t) pairs, of length N . Let n denote the total number of timeslots in S . Consider a stream S' of (x, y) pairs where for each element $(d, t) \in S$, there is an element $(x = d, y = t)$ in S' . Then, the $\frac{n\alpha}{N}$ -HDHs in S' are the set of α -persistent items in S . There are two significant issues with using such a reduction for solving our problem using an algorithm

(such as in [110, 20]) for HDH identification. (1)The first one is that for HDH identification, the threshold $\frac{n\alpha}{N}$ should be known beforehand. Though n , the number of timeslots is usually known before the stream is observed, the number of packets N is not known beforehand, so the prior algorithms for HDHs cannot be directly used. (2)Next, even if we were to modify the algorithms for HDHs to work with an “adaptive threshold”, that can change as the number of elements increases (which seems non-trivial), there is special structure in the data in the persistent items identification problem that can be used here. In the heavy distinct hitter problem on a stream of (x, y) values, there is no relative ordering required on the y values, and the same (x, y) tuple can re-occur at arbitrary positions in the stream. But in the persistent items problem on a stream of (d, t) tuples, the t values must be in a non-decreasing order (since they represent the times of observation at the stream processor). An important consequence of this difference is that the algorithms for HDH identification ([110, 20]) need to use “distinct counters” (such as in [57, 51]) to count the number of distinct y values associated with each value of x . Hence, the space complexity of their algorithms is the number of counters maintained multiplied by the space taken by an (approximate) distinct counter. Approximate distinct counting is inherently expensive space-wise, since it has been shown [68] that maintaining distinct counters with a relative error of ϵ requires $\Omega(1/\epsilon^2)$ space. Our algorithm does not need to use approximate distinct counters, making it simpler, more efficient, and easier to implement.

Prior work in Giroire *et al.* [59] used the following method to track persistent items in a stream of network traffic. For each distinct item in the stream, their method maintained (1)The number of timeslots in which the item has appeared in the stream so far, and (2)Whether or not the item has appeared in the current timeslot. This allowed them to exactly compute the number of timeslots that each item has appeared in, and hence exactly track the set of persistent items. However, the space taken by this scheme is proportional to the number of distinct items in the stream. The stream could have a very large number of distinct items (for example, IP sources, or destinations), and the memory overhead may render this infeasible on a typical network monitor or a router. Thus the challenge is to track the persistent items in a stream using a small workspace, and minimal processing per element. Further, all tracking must be done online, and the system does not have the luxury of making multiple passes through the

data.

2.1.1 Contributions

In this work, we present the first small-space approximation algorithm for tracking persistent items in a data stream, and an evaluation of the algorithm. Our contributions are as follows.

Space Lower Bound: We first consider the problem of exactly tracking all α -persistent items in a stream, for some user-defined $\alpha \in (0, 1]$. For this problem, we show that any algorithm that solves it must use $\Omega(|D| \log n \alpha)$ space, where $|D|$ is the number of distinct items in the stream, and n is the total number of slots, *even when the number of persistent items is much smaller than $|D|$.*

Approximate Tracking of Persistent Items: In light of the above lower bound, we define an approximate version of the problem. We are given two parameters, α - the threshold for persistence, and $\epsilon < \alpha$, an approximation (or “uncertainty”) parameter. The task is to report a set of items with the following properties: every item that is α -persistent is reported, and no item with persistence less than $(\alpha - \epsilon)$ is reported. We also formulate this problem for a “sliding window” of the most recently observed items of the stream.

Small Space Algorithm: For the above problem of approximate tracking of persistent items, we present a randomized algorithm that can approximately track the α -persistent items using space that is typically much smaller than the number of distinct items in the stream. The expected space complexity of the algorithm is $O\left(\frac{P}{\epsilon n}\right)$, where P is the sum of the persistence values of all items in the stream, and n is the total number of timeslots. The algorithm has a small probability of a false negative (i.e. an α -persistent item is missed). This probability can be made arbitrarily small, at the cost of additional space. Note that any algorithm will need space that is at least as large as the size of the output, i.e., the number of α -persistent items in the stream. The worst case scenario is when every item is α -persistent, forcing the algorithm to use space proportional to the number of distinct items! Fortunately, this situation does not seem to occur in practice and only a fraction of items are very persistent, and this helps our algorithm considerably. We also prove that if persistence of different items in a stream follow a power law distribution, then the space taken by our algorithm is $O\left(\frac{1}{\epsilon}\right)$.

Sliding Windows: In most network monitoring applications, the data set of interest is not the entire traffic stream, but only a window of the recent past (say, the n most recent timeslots). For instance, Giroire *et al* [59] used this sliding window model in their work on botnet traffic detection. Though the size of the data set has decreased when compared with the fixed window case, maintaining statistics over a sliding window is still a hard problem, since the data contained within a sliding window is often too large to be stored completely within the memory of the stream processor. This is a harder problem than the fixed window, since it has to deal with (old) elements falling off the window. We present an extension to our fixed window algorithm to handle the sliding window model. Interestingly, the expected space cost of our sliding window algorithm is within a factor of two of the space cost of the fixed window algorithm.

Experimental Evaluation: We evaluate our algorithm against three datasets: a large, real-world network traffic trace (which we call **HeaderTrace**) collected from an Internet backbone link, as well as two artificially created datasets, which we call **Synthetic1** and **Synthetic2** respectively, the latter having a skewness of persistence (17.17) which is three times that of the former (5.67). In other words, **Synthetic1** had a more *uniform* distribution of persistence than **Synthetic2** or **HeaderTrace**.

Our algorithm performed best on **HeaderTrace** and **Synthetic2**, and a little worse on **Synthetic1**. On **HeaderTrace**, our small-space algorithm uses upto 85% less space than the naive algorithm and typically incurs a false positive rate of less than 1% and a false negative rate of less than 4%. We also see that false positive rate never exceeds 3% for any parameter setting, while the false negative rate stays below 5% for all but the most aggressive thresholds for persistence. For **Synthetic2**, which had a skewness about twice that of **HeaderTrace**, the maximum FPR was 2.2%, the typical FNR being about 6%. The skewness of persistence for the **Synthetic1** dataset was about 60% of that of **HeaderTrace** (9.28). Although the maximum FNR for **Synthetic1** is 11.5% (the theoretical maximum FNR is 13%) and the maximum FPR is 15.6%, the typical FNR and FPR are both within 4%. The comparative performance on the three datasets shows that our algorithm in fact works better for datasets with high skewness of persistence and low mean persistence, which is very typical of real-life network traffic.

2.1.2 Roadmap

The rest of this paper is organized as follows. A precise statement of the problem is presented in Section 3.1.1, followed by a lower bound on the space cost of exactly tracking the persistent items in a stream. Our algorithms for the fixed and sliding windows models are presented in Section 2.3, followed by their analysis and correctness. Experimental results are described in Section 2.4. A detailed discussion of related work is presented in Section 2.5.

2.2 Problem Definition

Consider a world where time is divided into timeslots (or slots) that are numbered $1, 2, \dots$. Let S be a stream of elements of the form $S = \langle (d_1, t_1), (d_2, t_2), \dots \rangle$. Each element is a tuple (d_i, t_i) , where d_i is an item identifier (IP address, hostname, etc), and t_i is the time slot during which the element arrived. It is assumed that the t_i s are in non-decreasing order. All elements that have the same values of t_i are said to be in the same timeslot. Clearly, a timeslot consists of elements that form a contiguous subsequence of the observed stream.

The duration of a timeslot depends on the application on hand. In the botnet detection application [59], the duration of a timeslot was chosen to be between 1 hour and 24 hours, primarily because these were suspected to be the possible lengths of time between successive connections from the (infected) client to malicious destinations, for the botnets that they considered. Since then, there have been other botnet attacks that work on a much smaller timescale (see Section 2.4 for a discussion). In an eventual solution to botnet attack detection, we may need to consider running the algorithm simultaneously with different timeslot durations, to monitor multiple types of attacks.

We define a window S_ℓ^r to consist of all stream elements (d_i, t_i) whose timeslots are in the range $[\ell, r]$, i.e. $S_\ell^r = \{(d_i, t_i) \in S \mid \ell \leq t_i \leq r\}$. The size of window S_ℓ^r is defined as $(r - \ell + 1)$, i.e. the number of timeslots it encompasses. For a given window we define the persistence of an item in that window as follows:

Definition 2.2.1 *The persistence of an item d over a window S_ℓ^r , denoted $p_d(\ell, r)$, is defined*

as the number of distinct slots in $\{\ell, \ell + 1, \dots, r\}$ that d appeared in.

$$p_d(\ell, r) = |\{t | ((d, t) \in S) \wedge (\ell \leq t \leq r)\}|$$

Definition 2.2.2 An item d is said to be α -persistent in window S_ℓ^r if $p_d(\ell, r) \geq \alpha(r - \ell + 1)$.

In other words, d must have occurred in at least an α fraction of all slots within the window.

We state two versions of the problem, the first version for a fixed window, and the second version for a sliding window. In practice, the sliding window version is more useful.

2.2.1 Exact Tracking of Persistent Items

Problem 1 Identifying Persistent Items Over a Fixed Window: *Devise a space-efficient algorithm that takes as input a prespecified window $W = S_1^n$ and a persistence threshold α , and at the end of observing the stream, returns the set of all items that are α -persistent. In other words, the algorithm will report every item that is α -persistent in W and will not report any item that is not α -persistent.*

A straightforward algorithm for this problem would track every distinct item in the stream, and for each distinct item, count the number of slots (from 0 to $n - 1$) during which the item appeared. For a single item, its persistence can be tracked in a constant number of bytes (assuming that the item identifier and slot number can be stored in constant space), by maintaining a counter for the number of timeslots the item has appeared in so far, in addition to one bit of state for whether or not the item has appeared in the current timeslot. The total space consumed by the naive algorithm is of the order of the number of distinct items in the stream. In general, this would be a large number and the space overhead may make it infeasible for this algorithm to be deployed within a network router.

Space Lower Bound for Exact Tracking: We now show that any algorithm that solves Problem 1 exactly must require $\Omega(m)$ space in the worst case, where m is the number of distinct items in the input. Importantly, $\Omega(m)$ space is needed even if the number of persistent items is much smaller than m .

Lemma 2.2.1 *Any algorithm that can exactly solve Problem 1 must use $\Omega(m \log(n\alpha + 1))$ bits of space in the worst case, where m is the number of distinct items in the input.*

Proof: Without loss of generality, suppose that the m distinct items that appear in the stream are labeled $1, 2, 3, \dots, m$. Consider the state of the stream after observing $k = (n - \alpha n)$ timeslots. For i from 1 to m , let n_i denote the number of timeslots among $1, 2, \dots, k$ during which item i has appeared. Consider the vector $u = \langle n_1, n_2, \dots, n_m \rangle$. Consider the following set V of possible assignments to u , where each component in u is chosen from the range $0, 1, 2, \dots, \alpha n$. The size of V is $(1 + n\alpha)^m$. We show that any algorithm that solves Problem 1 must distinguish between two distinct vectors in V , and hence must have a different state of its memory for two input streams that result in different assignments to u .

We use proof by contradiction. Suppose the above was not true, and there were two input streams A and B which, at the end of k slots, resulted in vectors $v_A, v_B \in V$ respectively. Suppose $v_A \neq v_B$ but the states of the algorithm's memory were the same after observing the two inputs. Now, v_A and v_B must differ in at least one coordinate. Without loss of generality, suppose they differed in coordinate 1, so $n_1(A) \neq n_1(B)$, and without loss of generality suppose $n_1(A) < n_1(B)$. Consider the rest of the stream, from slot $n - n\alpha$ onwards. Suppose these slots had $n\alpha - n_1(B)$ slots in which item 1 occurred. Clearly, appending this stream to stream A results in a stream with n slots where the persistence of item 1 is $n_1(A) + (n\alpha - n_1(B)) = n\alpha - (n_1(B) - n_1(A)) < n\alpha$, and appending this same stream to stream B results in a stream with n slots where the persistence of item 1 is $n\alpha$. Thus, item 1 must be reported as α -persistent in the latter case, and not in the former case. But this is not possible, since the algorithm has the same memory state for both A and B , and sees the same substream henceforth, leading to a contradiction.

To distinguish between any two vectors in V , the algorithm needs at least $\log |V|$ bits of memory. Since the size of V is $(n\alpha + 1)^m$, the lower bound is $\Omega(m \log(n\alpha + 1))$ bits. ■

2.2.2 Approximate Tracking of Persistent Items

In light of the above lower bound on the space cost of exact tracking of persistent items, we define a relaxed version of the problem. Here, in addition to the persistence threshold α , the user provides two additional parameters, $\epsilon \in [0, 1]$, an “uncertainty parameter”, and $\delta \in [0, 1]$, an error probability.

Problem 2 Approximate Tracking of Persistent Items over a Fixed Window: *Given a fixed window $W = S_1^n$, persistence threshold α , approximation parameter ϵ , and error probability δ , devise a small space algorithm that returns a set of items with the following properties.*

- A. *Every Persistent Item is reported with high probability. If $p_d(1, n) \geq \alpha$, then d is returned as being persistent with a probability at least $1 - \delta$.*
- B. *Items that are far from persistent are not reported. If $p_d(1, n) < (\alpha - \epsilon) \cdot n$, then d is not reported*

Sliding Windows. The sliding window version of the problem requires that we continuously monitor the window of the n most recent timeslots in the stream.

Problem 3 Approximately identifying Persistent Items over a Sliding Window: *The problem of approximately tracking persistent items over a sliding window is the same as the above Problem 2, except that the window of interest, W , is the set of the n most recent timeslots in the stream, and changes continuously with time.*

The fixed window version is a special case of the sliding window, where the window is equal to the entire stream. The space lower bound for fixed window obviously applies to the sliding window version, hence it is also necessary to consider an approximate version of the problem for sliding windows, if we are to achieve a small space solution.

2.3 An Algorithm for Approximate Tracking of Persistent Items

We present algorithms for approximate tracking of persistent items in a stream. We first present the algorithm for tracking persistent items over a fixed window, followed by a proof of correctness and analysis of complexity. We then present the algorithm for sliding window.

2.3.1 Fixed Window

Intuition. The goal is to track the persistence of as few items in the stream as possible, and hence minimize the workspace used by the algorithm. Ideally, we track (and hence, use space for) only the α -persistent items in the stream, and not the rest. But this is impossible, since we do not know in advance which items are α -persistent.

The strategy is to set up a hash-based “filter”. Each stream element is sent through this filter, and if it is selected by the filter, then the persistence of the corresponding item is tracked in future timeslots. The filter behaves in such a way that if the same item reappears in the same timeslot, then its chances of being selected by the filter are not enhanced, but if the same item reappears in different timeslots, then its chances of passing the filter get progressively better. For achieving the above, the filter for an item is selected to be dependent on the output of a hash function whose inputs are both the item identifier as well as the timeslot within which it appeared.

Let h denote a hash function that takes two inputs, and whose output is a random real number in the range $[0, 1]$. For item d arriving in slot t , the item passes through the filter if $h(d, t) < \tau$, for some pre-selected threshold τ . The value of τ is chosen to be small enough that an item with a small value of persistence is not likely to cross this filter; in particular, transient items which only occur in a constant number of timeslots will almost certainly not make it. Note that if the same item d reappears in the same timeslot t , then the hash output $h(d, t)$ is the same as before, hence the probability of the item passing the filter does not increase.

After an item has passed the filter, the persistence of this item in the remaining timeslots is tracked exactly, since this requires only a constant amount of additional space (per item). Finally, the persistence of an item is estimated as the number of slots that it has appeared in since it started being tracked (this is known exactly), plus an estimate of the number of slots it had to appear in before we started tracking it. An item is returned as α -persistent if its estimated persistence is greater than a threshold T (decided by the analysis). Note that there may be items which are being tracked because they passed the filter, but are not returned as α -persistent, since the estimate of their persistence did not exceed T .

The higher the threshold τ , the greater is the accuracy in our estimate of the persistence, but this comes at the cost of higher memory consumption since more items will now pass the filter. Setting the value of τ gives us a way to tradeoff accuracy versus space.

Formal Description. Let $D(S)$ denote the set of distinct items in the stream S , and suppose that the timeslots of interest are $1, 2, \dots, n$. The stream processor tracks only a subset of $D(S)$, and maintains a data structure that we call a “sketch”, which summarizes the stream elements seen so far. Let \mathcal{S} denote the sketch data structure maintained by the algorithm.

\mathcal{S} is a set of tuples of the form (d, n_d, t_d) , where d is an item that has appeared in the stream, n_d is the number of slots in which d has appeared, since we started tracking it, and t_d is the most recent timeslot during which d has appeared. For each item d , if d is being tracked, then there is a tuple of the form (d, \cdot, \cdot) belonging in \mathcal{S} ; if d is not being tracked, then there is no such tuple in \mathcal{S} . For each item d , there can never be more than one tuple of the form (d, \cdot, \cdot) in \mathcal{S} at a time. We say $d \in \mathcal{S}$ to mean “there is a tuple (d, \cdot, \cdot) belonging to \mathcal{S} ”. Similarly, we say $d \notin \mathcal{S}$.

The inputs to the algorithm are the persistence threshold α , the total number of slots n , approximation parameter ϵ , and error probability δ . The algorithm selects a hash function $h(d, t)$ where d is an item, and t is the timeslot number. It is assumed that $h(d, t)$ is a uniform random real number in $(0, 1)$, and that the outputs of h on different inputs are mutually independent; when presented with the same input (d, t) , the hash function returns the same output. We note that it is possible to work with weaker assumptions of hash functions whose range is a finite set of integers, but we assume the current model for simplicity and ease of exposition.

Before any element arrives, Algorithm 8 Sketch-Initialize is invoked to initialize the data structures. When an element (d, t) arrives, Algorithm 9 is invoked to update the \mathcal{S} data structure. When there is a query for persistent items in the stream, Algorithm 3 Detect-Persistent-Items is called to process the query and will return a list of all items deemed persistent.

Algorithm 1: Sketch-Initialize($m, n, \alpha, \epsilon, \delta$)

Input: Size of domain m ; Total number of slots n ; persistence threshold α ; parameter ϵ ; error probability δ

- 1 Initialize the hash function $h : ([1, m] \times [1, n]) \rightarrow (0, 1)$;
 - 2 $\mathcal{S} \leftarrow \phi$; $\tau \leftarrow \frac{2}{\epsilon n}$; $T \leftarrow \alpha n - \frac{\epsilon n}{2}$
-

Algorithm 2: Sketch-Update(d, t)

Input: d is an item; t is the timeslot of arrival

- 1 **if** $d \in \mathcal{S}$ **then**
 - 2 **if** $t_d < t$ **then**
 - 3 /* d appeared in a new slot */
 - 4 $n_d \leftarrow n_d + 1$; $t_d \leftarrow t$;
 - 5 **end**
 - 6 **else**
 - 7 **if** $h(d, t) < \tau$ **then**
 - 8 /* Start tracking item d from now onwards */
 - 9 $\mathcal{S} \leftarrow \mathcal{S} \cup (d, 1, t)$;
 - 10 **end**
 - 11 **end**
-

Algorithm 3: Detect-Persistent-Items

- 1 **foreach** tuple $(d, n_d, t_d) \in \mathcal{S}$ **do**
 - 2 $\hat{p}_d \leftarrow n_d + \frac{1}{\tau}$
 - 3 **if** $\hat{p}_d \geq T$ **then**
 - 4 Report d as a persistent item
 - 5 **end**
 - 6 **end**
-

2.3.1.1 Analysis of the Fixed Window Algorithm

We present the proof of correctness and analysis of space complexity. Consider an item d , with absolute persistence $p_d = p_d(1, n)$. For parameter q , $0 < q \leq 1$, let $G(q)$ denote the geometric random variable with parameter q , i.e., the number of Bernoulli trials till a success (including the trial when the success occurred), where the different trials are all independent, and the success probability is q in each trial.

For each item d that appeared in the stream, there are two possibilities: (1) either d is tracked by the algorithm from some timeslot t onwards, or (2) d is not tracked by the algorithm, because none of the tuples (d, t) were selected by the filter.

In each distinct slot where d appears, the probability of d being sampled into the sketch is τ . If $G(\tau) > p_d$, then this will lead to case (2) above, and d will fail to make it into the sketch \mathcal{S} . On the other hand, if $G(\tau) \leq p_d$, this will lead to case (1), and d will be inserted into the sketch at some timeslot in Algorithm 9, and the counter $n_d = p_d - G(\tau) + 1$.

Lemma 2.3.1 False Negative: *If an item d has $p_d \geq \alpha n$, then the probability that this item will not be reported as α -persistent by Algorithm 3 is no more than e^{-2} .*

Proof: From Algorithm 3, the item will not be reported if $\hat{p}_d < T$, i.e., $n_d + \frac{1}{\tau} < T$. Using $\tau = \frac{2}{\epsilon n}$ and $T = \alpha n - \frac{\epsilon n}{2}$, we get:

$$\begin{aligned} \Pr[\text{False Negative}] &= \Pr[p_d - G(\tau) + 1 + \frac{1}{\tau} < T] \\ &= \Pr[G(\tau) > 1 + \frac{1}{\tau} + p_d - T] \\ &= \Pr[G(\tau) > 1 + \frac{1}{\tau} + \frac{\epsilon n}{2} + (p_d - \alpha n)] \\ &\leq \Pr[G(\tau) > \frac{2}{\tau}] \end{aligned}$$

In the last step, we have used the fact $p_d \geq \alpha n$, and $\frac{1}{\tau} = \frac{\epsilon n}{2}$. Using the fact $\Pr[G(p) > t] = (1 - p)^t$, we get

$$\Pr[\text{False Negative}] \leq (1 - \tau)^{\frac{2}{\tau}} \leq e^{-2}$$

In the last step, we have used the inequality $1 - x \leq e^{-x}$. ■

Lemma 2.3.2 *Items that are far from persistent are not reported:* If an item d has $p_d < (\alpha - \epsilon)n$, then d will not be reported by Algorithm 3 as an α -persistent item.

Proof: For such an item, the value of n_d at the end of observation is $n_d = p_d - G(\tau) + 1$. Let f denote the probability that d is reported as α -persistent. We have:

$$\begin{aligned}
f &= \Pr[n_d + \frac{1}{\tau} \geq T] \\
&= \Pr[p_d - G(\tau) + 1 + \frac{1}{\tau} \geq \alpha n - \frac{1}{\tau}] \\
&= \Pr[G(\tau) \leq (p_d - \alpha n) + 1 + \frac{2}{\tau}] \\
&= \Pr[G(\tau) \leq p_d - (\alpha - \epsilon)n + 1] \\
&\leq \Pr[G(\tau) \leq 0] = 0
\end{aligned}$$

■

Lemma 2.3.3 *The expected space taken by the \mathcal{S} is $O\left(\frac{1}{\epsilon n} \sum_{d \in D(S)} p_d\right)$, where $D(S)$ is the set of all distinct items in stream S . We assume that storing a tuple (d, n_d, t_d) takes a constant amount of space.*

Proof: The space taken by \mathcal{S} is a random variable, since the decision of whether or not to allocate space to an item is a randomized decision. For item d , let random variable Z_d be defined as follows. $Z_d = 1$ if the algorithm tracks d , i.e $d \in \mathcal{S}$, and $Z_d = 0$ otherwise.

Let $Z = \sum_{d \in D(S)} Z_d$. If we assume that the space required for storing a single tuple (d, \cdot, \cdot) in \mathcal{S} is a constant number of bytes, say c , then the space used by \mathcal{S} is cZ bytes. Now, for the random variable Z , by linearity of expectation, we get:

$$E[Z] = E\left[\sum_{d \in D(S)} Z_d\right] = \sum_{d \in D(S)} E[Z_d] = \sum_{d \in D(S)} \Pr[Z_d = 1] \quad (2.1)$$

$$\Pr[Z_d = 0] = (1 - \tau)^{p_d} \quad (2.2)$$

Using Taylor's expansion,

$$\begin{aligned}
e^{-2\tau} &\leq 1 - 2\tau + 4\tau^2/2 \\
&\leq 1 - 2\tau + \tau = 1 - \tau \quad (\text{assuming } \tau \leq 1/2)
\end{aligned}$$

Using in Equation 2.2, we get:

$$\Pr[Z_d = 0] \geq (e^{-2\tau})^{p_d} = e^{-2\tau p_d}$$

Thus,

$$\begin{aligned} \Pr[Z_d = 1] &= 1 - \Pr[Z_d = 0] \leq (1 - e^{-2\tau p_d}) \\ &\leq (1 - (1 - 2\tau p_d)) \text{ (using } e^{-x} > 1 - x \text{)} \\ &= 2\tau p_d \end{aligned}$$

Using in Equation 2.1, we get:

$$E[Z] \leq \sum_{d \in D(S)} 2\tau p_d = 2\tau \sum_{d \in D(S)} p_d = \frac{4}{\epsilon n} \sum_{d \in D(S)} p_d$$

■

Discussion: The expression for the space complexity shows that the expected space required for an item d is proportional to p_d/n . Note that p_d can range from 1 till n , but in a typical stream, the persistence of most items can be expected to be small, with only a few items having a large persistence. Thus, in the typical case, for example, with a Zipfian distribution of packet frequencies and persistence, the space taken by the sketch will be much smaller than the number of distinct items in the input.

Space Complexity for Specific Distributions. Let $P = \sum_{d \in D(S)} p_d$ denote the sum of the persistence values of all items in the stream. We now show that if the persistence values of the different items followed a Zipfian distribution, then $P = O(n)$, leading to a constant space complexity, independent of the number of distinct items in the input.

Lemma 2.3.4 *If the persistence of different items in $D(S)$ followed a Zipfian distribution, then the space complexity of the sketch is $O(\frac{1}{\epsilon})$.*

Proof: Let ρ_k be the persistence of the k th most persistent item for $k \in 1, 2, \dots, |D(S)|$. With a Zipfian distribution, $\rho_k = \frac{c}{k^\beta}$, for some $c > 0$ and $\beta > 1$. Since the persistence of an item is bounded by n , $\rho_1 = c \leq n$. Let $\zeta(\cdot)$ be the Reimann Zeta function.

$$\sum_{d \in D(S)} p_d = \sum_{k \leq |D(S)|} \frac{c}{k^\beta} \leq c \sum_{k=1}^{\infty} \frac{1}{k^\beta} = c\zeta(\beta) \leq n\zeta(\beta)$$

Thus, from Lemma 2.3.3, we have $E[Z] \leq \frac{4}{\epsilon n} n\zeta(\beta) = \frac{4\zeta(\beta)}{\epsilon}$

By the Maclaurin-Cauchy test, we know for $\beta > 1$, the series represented by $\zeta(\beta)$ converges, and is usually a small constant, which proves the lemma. For example, if $\beta = 1.5$, then $\zeta(1.5) = 2.6$. For this case, we get: $\sum_{d \in D(S)} p_d \leq 2.6n$, and thus, from Lemma 2.3.3, $E[Z] < \frac{11}{\epsilon}$.

■

Theorem 2.3.1 *The above algorithms 9 and 3 can be used in an algorithm for tracking persistent items in a fixed window with the following properties:*

- A. Each α -persistent item is reported with probability at least $1 - \delta$.
- B. No item d such that $p_d < (\alpha - \epsilon)n$ is reported.
- C. The space complexity of the algorithm is $O\left(\frac{P \log(1/\delta)}{\epsilon n}\right)$, where $P = \sum_{d \in D(S)} p_d$.
- D. The processing time per stream element is $O(\log \frac{1}{\delta})$.

Proof: Algorithms 9 and 3 achieve most of the above properties. From Lemma 2.3.1, we get that the probability of a persistent item not being reported is no more than e^{-2} . The only task now is to bring down the probability of a false negative to δ .

To achieve this, we run $(1/2) \ln \frac{1}{\delta}$ instances of Algorithm 9 in parallel, and return the union of the items reported by all the instances. For an item that is persistent, it is not reported only if it is missed by every instance. The probability that this happens is no more than $(e^{-2})^{(1/2) \ln \frac{1}{\delta}}$, which is δ . For an item d whose persistence is less than $(\alpha - \epsilon)n$, from Lemma 2.3.2, we see that the item is not returned by any instance, and hence will not be present in aggregated result, proving property B.

Property C follows from Lemma 2.3.3, adding a multiplying factor of $O(\log \frac{1}{\delta})$. For the time complexity (property D), we note that Algorithm 9 can be made to run in constant expected time if the sketch \mathcal{S} is organized as a hash table with the item identifier as the key. ■

2.3.2 Sliding Windows

In this setting, we are interested only in the substream of elements that belong to the n most recent timeslots. If c is the current timeslot, then the window of interest is S_{c-n+1}^c . Note that n here does not represent the number of timeslots in the stream, but the number of timeslots in the window. We now present an algorithm solving Problem 3. The intuition for the sliding window algorithm is as follows.

Suppose we started a new fixed window data structure for each new timeslot. This would suffice, since any sliding window query in the future will be covered by one of these fixed window data structures. For now, suppose that \mathcal{S}_t was the fixed window data structure that we start from time t onwards (this will serve the window S_t^{t+n-1}). At first glance, it seems like this would be too much space, since the cost would be n times the space for a single fixed window data structure.

The space can be reduced through the following observations: (1) when we start a fixed window data structure at a particular timeslot t , say, only a few of the items (approximately a τ fraction of the items) that arrive in timeslot t will be selected into this data structure; (2) for those items d that were not selected into \mathcal{S}_t in timeslot t , the tuple for d in \mathcal{S}_t can be shared with the tuple for d in \mathcal{S}_{t+1} ; (3) further, when the current timeslot is t , we can afford to discard \mathcal{S}_r for $r \leq (t - n)$, since these data structures will never be used in a future query.

Thus, the sketch used by our algorithm at time c is effectively $\cup_{i=c-n+1}^c \mathcal{S}_i$, where \mathcal{S}_i is the fixed window sketch starting at timeslot i . Through observation (2), we reduce the space by having a single tuple for d in $\mathcal{S}_i, \mathcal{S}_{i+1}, \dots, \mathcal{S}_j$ such that j is the first timeslot in $i, i + 1, \dots, j$ where d was selected into the sketch.

The formal description of the algorithm for the Sliding Window model is presented in Algorithms 4, 5, 6, and 7. The sketch \mathcal{S} is a set of tuples of the form $(d, t, n_{d,t}, t_{d,t})$, where d is an item identifier, t is the timeslot when this tuple was created, $n_{d,t}$ is the number of timeslots since t when d has reappeared, and $t_{d,t}$ is some state that we maintain to eliminate counting reoccurrences of d within the same timeslot. In the following discussion, we say “ (d, t) belongs in the sketch”, or “ $(d, t) \in \mathcal{S}$ ”, if there is a 4-tuple of the form (d, t, \cdot, \cdot) in the sketch. In our

sketch, for any item d and timeslot t , there can be at most one tuple of the form (d, t, \cdot, \cdot) .

Algorithm 4: Sliding-Window-Sketch-Initialize $(m, n, N, \alpha, \epsilon, \delta)$

Input: Size of domain m ; window size n ; maximum number of slots N ; persistence threshold α ; parameter ϵ ; error probability δ

- 1 Initialize the hash function $h : ([1, m] \times [1, N]) \rightarrow (0, 1)$;
 - 2 $\mathcal{S} \leftarrow \phi; \tau \leftarrow \frac{2}{\epsilon n}; T \leftarrow (\alpha - \frac{\epsilon}{2})n$
-

Algorithm 5: Sliding-Window-Sketch-Update (d, t)

Input: d is an item; t is the timeslot of arrival

- 1 **if** $(d, t) \in \mathcal{S}$ **then**
 - 2 | return
 - 3 **end**
 - // Consider starting a new tuple, tracking d from slot t onwards.
 - 4 **if** $h(d, t) < \tau$ **then**
 - 5 | $\mathcal{S} \leftarrow \mathcal{S} \cup (d, t, 1, t)$
 - 6 **end**
 - 7 **foreach** t' such that $(d, t') \in \mathcal{S}$ **do**
 - 8 | Let $(d, t', n_{d,t'}, t_{d,t'})$ be the tuple corresponding to (d, t')
 - // Incorporate (d, t) into this tuple if not been done yet
 - 9 | **if** $t_{d,t'} < t$ **then**
 - 10 | | // d has not been seen in slot t by this tuple
 - | $n_{d,t'} \leftarrow n_{d,t'} + 1; t_{d,t'} \leftarrow t$
 - 11 | **end**
 - 12 **end**
-

During the initialization phase of the algorithm, \mathcal{S} is initialized to empty, τ to $\frac{2}{\epsilon n}$, and T to $\alpha n - \frac{\epsilon n}{2}$. When we want to add an element (d, t) to the sketch, there are two possible cases. First, if there is an entry in the sketch of the form (d, t, \cdot, \cdot) , then this element can be safely ignored, since the same combination of item and timeslot has been observed earlier. Otherwise, if (d, t) hashes to an appropriately small value (less than τ), then a new entry is created for tracking d , starting from time t onwards, that will serve to answer queries on certain windows that include t within them. Simultaneously, (d, t) is used to update each of the tuples in \mathcal{S} that track d . Whenever time advances, and the window slides forward from t to $t + 1$, all entries (d, t', \cdot, \cdot) in \mathcal{S} such that $t' \leq (t - n)$ are discarded, because stream windows of current and future interest will not be served by this entry. Let $p_d^t = p_d(t - n + 1, t)$ denote the persistence of d over the window $[t - n + 1, t]$.

Algorithm 6: Actions taken when timeslot changes from $c - 1$ to c

// Discard old items
 1 Discard items $(d, t, \cdot, \cdot) \in \mathcal{S}$ where $t \leq (c - n)$

Algorithm 7: Sliding-Window-Detect-Persistent-Items(c)

Input: c is the current timeslot. The window of interest is $[c - n + 1, c]$.

1 Let \mathcal{S}_{cur} be all tuples $(d, t', n_{d,t'}, t_{d,t'})$ in \mathcal{S} such that both the following conditions are true: (A) $t' \geq (c - n + 1)$ and (B) There is no t'' such that $(d, t'') \in \mathcal{S}$ and $(c - n + 1) \leq t'' < t'$.

2 **foreach** tuple $(d, \cdot, n_d, t_d) \in \mathcal{S}_{cur}$ **do**

3 $\hat{p}_d^c \leftarrow n_d + \frac{1}{\tau}$

4 **if** $\hat{p}_d^c \geq T$ **then**

5 | Report d as a persistent item in the window

6 **end**

7 **end**

2.3.2.1 Correctness and Complexity

For a pair (d, t) where d is an item identifier and t is a time slot, (d, t) is said to be stored in \mathcal{S} at time c if there exists a tuple (d, t, \cdot, \cdot) in \mathcal{S} at time c .

Lemma 2.3.5 **Items that are far from persistent in the window are not reported:**

At time c , if an item d has $p_d^c < (\alpha - \epsilon)n$, then d will not be reported as persistent in the window in Algorithm 7.

Proof: Consider such an item d , where $p_d^c < (\alpha - \epsilon)n$. We analyze the instances when d was processed by Algorithm 5. If d was never stored in the sketch from time $c - n + 1$ onwards, then there will not exist a tuple (d, t', \cdot, \cdot) in \mathcal{S} at time c , and d will not be reported by Algorithm 7.

Suppose at time c , there existed a tuple (d, t', n_d, t_d) in \mathcal{S} , such that $t' \geq (c - n + 1)$. This tuple was inserted into the sketch at time t' . From Algorithm 5, it can be seen that n_d is equal to the number of occurrences of d in timeslots $t', t' + 1, t' + 2, \dots, c$. This number cannot be more than p_d^c , and hence $n_d \leq p_d^c < (\alpha - \epsilon)n$.

In Algorithm 7, for item d , it must be true that:

$$\hat{p}_d^c = n_d + \frac{1}{\tau} < (\alpha - \epsilon)n + \frac{\epsilon n}{2} = \alpha n - \frac{\epsilon n}{2} = T$$

Since $\hat{p}_d^c < T$, d will not be reported as persistent. ■

Lemma 2.3.6 Sliding Window False Negative: *At time c , if an item d has $p_d^c \geq \alpha n$, then the probability that this item will not be reported as α -persistent in the current window by Algorithm 7 is no more than e^{-2} .*

Proof: Suppose that d was sampled into the sketch later than time $(c - n)$, i.e., there exists a tuple (d, t, n_d, \cdot) such that $t > (c - n)$. In such a case, Algorithm 7 selects the tuple (d, t', n_d, t_d) such that (A) $t' > (c - n)$ and (B) there is no tuple (d, t'', \cdot, \cdot) in \mathcal{S} such that $t'' < t'$. In other words, t' is the earliest timeslot in $[c - n + 1, c]$ when a sketch for d was initialized. Thus, it follows that from time $c - n + 1$ onwards (inclusive), d was not selected into the sketch till time t' . The number of times that d needs to occur in slots $c - n + 1, c - n + 2, \dots$ till it is sampled into \mathcal{S} is $G(\tau)$ (the geometric random variable with parameter τ). The counter n_d keeps track of the number of times d occurred in different timeslots starting from slot t' (inclusive). Since d occurred in the window in a total of p_d^c distinct slots, $n_d = p_d^c - G(\tau) + 1$.

$$\Pr[\text{False Negative}] = \Pr[p_d^c - G(\tau) + 1 + \frac{1}{\tau} < T]$$

In the proof of Lemma 2.3.1, it is shown that the above probability is no more than e^{-2} if $p_d^c \geq \alpha n$, and the lemma follows. ■

2.3.3 Space Complexity

The following result is useful for the space complexity.

Lemma 2.3.7 *A tuple (d, t) is stored in \mathcal{S} at time c if and only if both the following conditions are true:*

A. $t > (c - n)$

B. $h(d, t) < \tau$

Proof: Suppose (d, t) is stored in \mathcal{S} at time c . From Algorithm 6, it is clear that $t > (c - n)$, since otherwise (d, t) would have been discarded from the sketch. This proves condition A.

Also, in Algorithm 5, if $h(d, t) \geq \tau$, then (d, t) would never have been inserted into the sketch. Thus, it must be true that $h(d, t) < \tau$, proving condition B.

Now, suppose that both A and B were true. Then, it is clear that in Algorithm 5, (d, t) will be inserted into the sketch when it first appears. Further, this tuple will never be discarded from the sketch in Algorithm 6 since our current timeslot c satisfies $c < (t + n)$. ■

Lemma 2.3.8 Space Complexity: *Let Z_c denote the number of tuples in \mathcal{S} at time c , and D denote the set of all distinct items that appeared during timeslots $c - n + 1$ till c . Then,*

$$E[Z_c] = \frac{2}{\epsilon n} \sum_{d \in D} p_d^c$$

Proof: First, it can be verified that in Algorithm 5, if the same tuple (d, t) occurs multiple times, then the effect on the sketch is the same as if (d, t) occurred only once in the stream. Thus we can ignore repeated arrivals of the same tuple (d, t) .

For each tuple (d, t) that arrived, let random variable $Z_{d,t}^c$ be defined as follows. $Z_{d,t}^c$ is 1 if tuple (d, t) is stored in \mathcal{S} at time c . Let $D(S)$ denote the set of all distinct tuples (d, t) in the stream so far.

We have

$$Z_c = \sum_{(d,t) \in D(S)} Z_{d,t}^c$$

From Lemma 2.3.7, we have that $Z_{d,t}^c = 0$ if $t \leq (c - n)$. Thus, we can rewrite the above as:

$$Z_c = \sum_{\{(d,t)|t>(c-n)\}} Z_{d,t}^c \quad (2.3)$$

To compute the expectation of Z_c , we use linearity of expectation:

$$E[Z_c] = E \left[\sum_{\{(d,t)|t>(c-n)\}} Z_{d,t}^c \right] = \sum_{\{(d,t)|t>(c-n)\}} E[Z_{d,t}^c]$$

For a tuple (d, t) such that $t > (c - n)$, $Z_{d,t}^c$ is equal to 1 if it was sampled into the sketch at time t i.e., if $h(d, t) < \tau$. The probability of this event is $\tau = \frac{2}{\epsilon n}$. Let D denote the set of all distinct items that appeared in the stream during a timeslot i such that $(c - n) < i \leq c$.

$$E[Z_c] = \sum_{\{(d,t)|t>(c-n)\}} \tau = \sum_{d \in D} (p_d^c \cdot \tau) = \frac{2}{\epsilon n} \sum_{d \in D} p_d^c$$

■

2.4 Evaluation

We evaluated our small space algorithm and contrasted its performance with that of a naive (exact) algorithm, by running the two on the following three (one real, two synthetic) datasets described below. The goal of our experiments is to show how the performance of our algorithm varies with the skewness of persistence of the items appearing in a stream.

Dataset design:

- **HeaderTrace:** This is a real-world traffic trace dataset. The trace used had 885 million packets collected during a 3-hour period from a large Internet backbone link (source: CAIDA [24]). The data consists of timestamped packet headers, with the source and destination addresses, in addition to other attributes. From this packet header trace, we extracted a sequence of (destination IP address, timestamp) pairs which forms the input data stream. We divided the entire trace into slots of 30 seconds (to obtain a trace of 360 slots). The sliding window length was set to 100 slots.
- **Synthetic1:** This is a synthetic dataset that comprised of 1,024,680,418 (timeslot, itemID) tuples. The item-identifiers were from the universe $\{1, \dots, 4000000\}$, and the trace was simulated for a period of 30 days, the length of each timeslot being 15 minutes. Hence, there were $30 \cdot 24 \cdot 60 / 15 = 2880$ distinct timeslots. We split the universe of size 4000000 in 10 disjoint groups, and defined a list F of 10 fractions (the constraint being $\sum_{i=1}^{10} F_i = 1$) where F_i represented what fraction of the universe belongs to group i . We also kept a list P of 10 fractions, where P_i indicated the persistence of F_i over the whole trace of 2880 slots. The values of F_i 's and P_i 's are all listed in Table 2.1. As an example, for $i = 1$, $F_i = 0.01$ and $P_i = 0.95$, which implies group 1 comprised of 1% of the items of the universe (i.e., 40000 items), each of which will occur in $2880 \cdot 0.95 = 2736$ slots on expectation. Note that, the way we assigned the values of F_i 's and P_i 's mimics the real-life fact that the distribution of persistence is very skewed, so more than 50% of the items in the universe occur in less than 3 slots (on expectation). In practice, we put an item in group i in slot $j \in \{1, \dots, 2880\}$ with probability P_i . Also, before generating the actual tuples, we created a random permutation of the universe $\{1, \dots, 4000000\}$ by

FisherYates shuffle [47].

- **Synthetic2:** This is a synthetic dataset that comprised of 123,408,469 (timeslot, itemID) tuples. Like Synthetic1, for this also, the item-identifiers were from the universe $\{1, \dots, 4000000\}$, and the trace was simulated for 2880 distinct timeslots. It differs from Synthetic1 in the values of F_i 's and P_i 's, and the difference is evident in Table 2.1. Note that, for Synthetic2, 86% of the items in universe have a persistence of 0.001 only, whereas for Synthetic1, 55% of the items in universe have that persistence. On the other hand, for Synthetic1, 1% of the items in universe have a persistence of 0.95, whereas for Synthetic2, 0.1% of the items have that high persistence. This explains why the skewness of Synthetic2 is about thrice that of Synthetic1, and the mean persistence of Synthetic2 is about $\frac{1}{9}^{th}$ of that of Synthetic1.

Table 2.1: Distribution of persistence for all datasets

Partition of universe	Synthetic1		Synthetic2		HeaderTrace
	F_i	P_i	F_i	P_i	
$i = 1$	0.01	0.95	0.001	0.95	
$i = 2$	0.02	0.75	0.002	0.75	
$i = 3$	0.03	0.55	0.003	0.55	
$i = 4$	0.04	0.35	0.004	0.35	
$i = 5$	0.05	0.25	0.005	0.25	
$i = 6$	0.06	0.15	0.006	0.15	
$i = 7$	0.07	0.1	0.007	0.1	
$i = 8$	0.08	0.05	0.01	0.05	
$i = 9$	0.09	0.01	0.1	0.01	
$i = 10$	0.55	0.001	0.862	0.001	
$\sum_{i=1}^{10} F_i$	1		1		
Size of universe	4,000,000		4,000,000		2,047,953
Exp. packets	1,024,704,000		123,402,240		
Actual packets	1,024,680,418		123,408,469		885,055,227
Mean persistence	0.089		0.01		0.0177
Third moment of persistence (m_3)	0.0105		0.0014		0.0043
Variance of persistence (m_2)	0.015		0.0019		0.006
Skewness of persistence ($m_3/m_2^{3/2}$)	5.67		17.17		9.28

There is no obvious choice on what should be a suitable duration of the timeslot, since prior research has shown that the delay between successive botnet-related communications to

the same destination can range from a few minutes to a few days. A duration of a few minutes is reasonable, since many botnets have multiple events occurring within this time frame. For example, Li *et al* [85] observed periodic botnet-related events about every half an hour. Rajab *et al* [98] reported that the average “staying time” for bots that they monitored was about 25 minutes, and 90% of them lasted less than 50 minutes. Over a 24-hour window, the BRAT project [106] reported probes by 8 fast-flux botnets which showed periodicity, the periods being in the range of 1-10 minutes. Porras *et al* [97] showed that for iKeeB, the iPhone-based botnet, a compromised iPhone runs a shell script once every 5 minutes. For the **HeaderTrace** dataset, we finally decided on a duration of 30 seconds so that our 3 hour trace led to a sufficient number of slots, and for the **Synthetic1** and **Synthetic2** datasets, we chose the length to be 15 minutes. This helped us evaluate the scalability of our algorithm with increasing number of timeslots, and also to experiment with different slot-lengths. With the above setting of parameters, for the **HeaderTrace** and the synthetic datasets, we had reasonably large number of timeslots (360 and 2880 respectively) as well as a large number of packets per timeslot.

The algorithms were implemented in C++ using the STL extensions. For the hash functions in the small space algorithm (Algorithm 5), we used an endian-neutral implementation of the *Murmur Hash* algorithm [16], which is generally considered to generate high quality hash outputs.

We obtained the ground truth about the persistence of individual items (IP addresses for **HeaderTrace**) by running the naive algorithm over the input data streams. Note that, although for the synthetic datasets, we determined which item will have how much persistence, the actual data was generated by a probabilistic process, so we still needed to collect the *actual* persistence values of the items. In the process, for the **HeaderTrace** dataset, we discovered that a large fraction of the windows did not contain many persistent items. On such windows, our algorithm will run in a space-efficient manner, but we did not consider these windows since there would not be enough data for a fair comparison.

To simplify the presentation, on **HeaderTrace**, we focus on 11 specific “query” windows: these are [1, 100], [26, 125], [51, 150], . . . , [251, 350]. On both the synthetic datasets, the query windows are [1, 288], [289, 576], . . . , [2593, 2880], so on these two, the time-duration of the query

window was $288 \cdot 15 / 60 = 72$ hours. We use window $[a, b]$ to denote the window of all timeslots starting from a till b (both endpoints included).

On **HeaderTrace**, we found that the cumulative distribution of the persistence values in the dataset was highly skewed, for every query window that we tried. We present the CDF of persistence for three out of the 11 query windows: $[1, 100]$, $[101, 200]$ and $[201, 300]$ in Figure 2.1, but all the 11 query windows showed similar pattern. For example, in the $[101, 200]$ window, more than 50% IP addresses occur in 1 slot only, and 95% of the IP addresses occur in 20 or less slots. This confirms the utility of an algorithm like ours, which requires less space when items have lower average persistence. We made the distribution of **Synthetic1** less skewed (Figure 2.2) than that of **HeaderTrace** (skewnesses are respectively 5.67 and 9.28, as shown in Table 2.1), to construct **Synthetic1** as an adversarial input dataset, and found the results for **HeaderTrace** better than those for **Synthetic1**. In Figure 2.2, we present the CDF for only one query window as the distributions are identical across all windows, because of the way the dataset is generated. Then again, we constructed **Synthetic2** as the dataset with highest skew (17.17, as in Table 2.1) of all three, and thus it shows some improvement over **Synthetic1**, as we explain later.

Metrics: The following metrics were used. For parameter α , an item that is not α -persistent is called “transient”.

The **False Negative Rate (FNR)** is defined as the ratio of the number of items that were α -persistent, but were not reported by the small space algorithm, to the total number of α -persistent items in the window.

The **False Positive Rate (FPR)** is defined as the ratio of the number of transient items that were reported as persistent by the algorithm, to the total number of transient items.

The **Space Compression (SC)** is defined as the ratio of the number of tuples stored by the naive algorithm to the number of tuples stored by the small space algorithm.

The **Physical Space Compression (PSC)** is defined as the maximum resident set size of the naive algorithm to that of the small-space algorithm.

The notion of Space Compression (SC) is a logical one, and for the sliding window version of the problem (Problem 3), we were interested in the number of tuples of the form (d, t, \cdot, \cdot) ,

as referred to in Algorithms 5 to 7.

In the actual implementation, for each distinct item d , we maintained a sorted list (of variable size) of $(t', n_{d,t'})$ tuples, ordered by t' , where $n_{d,t'}$ indicates in how many distinct slots d has appeared since its appearance in slot t' . The sorted list helped us to check by binary-search if an item d has occurred in a given slot t' . When d appears in a slot t' it has not appeared in before, the tuple $(t', n_{d,t'})$ is initialized only if $h(d, t') < \tau$. Note that $t_{d,t'}$ - the last timeslot d has appeared in since its appearance in t' , does not depend on t' , and hence we maintained a single copy of this variable for each item d .

For computing the Physical Space Compression (PSC), for each combination of α and ϵ , we actually created a new process so that the resident set is created afresh. We expect the Physical Space Compression for (α, ϵ') to be higher than that for (α, ϵ) when $\epsilon' > \epsilon$ (because τ is lower for ϵ'), but we found that because of the way memory allocation algorithms work, if the algorithm runs first for (α, ϵ) and then for (α, ϵ') (using the same process), then, the memory allocated for (α, ϵ) is enough to accomodate the algorithm for (α, ϵ') , and the space-saving due to (α, ϵ') does not get reflected.

Note that both the numerator and the denominator of each metric depend on the query window $[c - n + 1, c]$ (n is the window length). To measure the ratios, we ran the small-space algorithm on the query windows defined previously and in each window, recorded all the items that were marked as persistent by the algorithm. The only source of randomness in each run is the output of the Murmur Hash function and we ran each simulation thrice using different seeds (we saw very minor variation in the results when different seeds were used.) Thus, for each parameter setting we had 11×3 data points for **HeaderTrace**, and 10×3 data points for both the synthetic datasets, and in each we recorded the false positives, the false negatives, and the number of tuples that were tracked. The ratios computed (by comparing to the naive algorithm) are then averaged across all the runs.

Observations along metrics:

For every value of α , the **False Negative Rate** (Figure 2.4a for **HeaderTrace**, Figure 2.5a for **Synthetic1** and Figure 2.6a for **Synthetic2**) increases as ϵ increases, which is expected.

However, although Lemma 2.3.6 bounds the False Negative Rate to $\frac{1}{e^2} = 13\%$, the algorithm

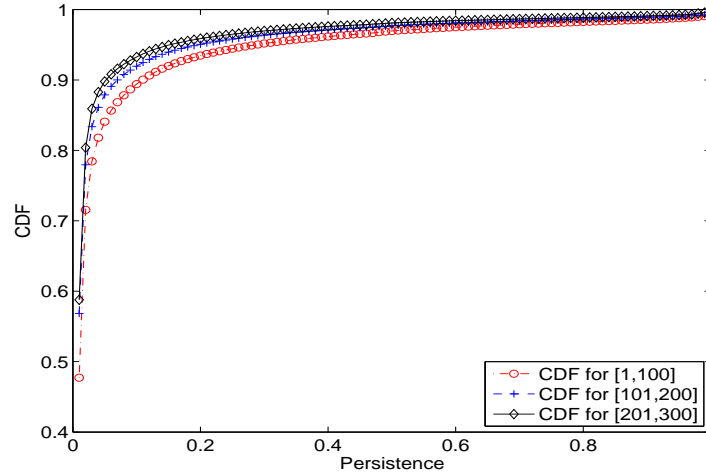


Figure 2.1: CDF of persistence values from 3 windows for the **HeaderTrace** dataset

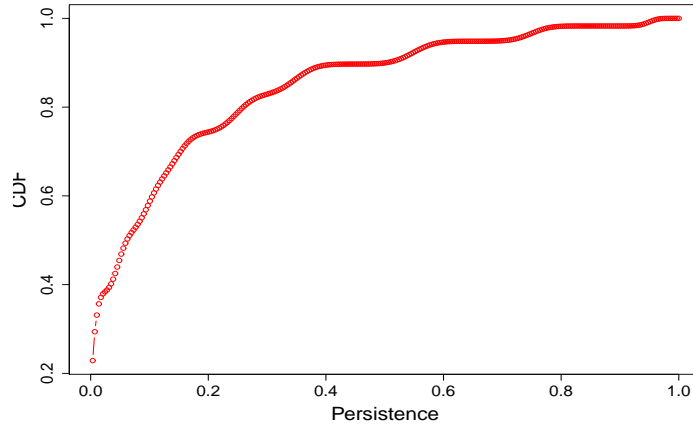


Figure 2.2: CDF of persistence values from the [1,288] window for the **Synthetic1** dataset

performed much better in practice - we found that even for $\alpha = 0.3$ and $\epsilon = 0.21$, the FNR was as low as 2% for **HeaderTrace** and $\sim 3.5\%$ for both the synthetic datasets. Note that $\frac{\epsilon}{\alpha}$ is a relative measure of error tolerance in α , which in this case is as high as 70%. The highest FNR we ever got was less than 10% for **HeaderTrace**, less than 12% for **Synthetic1** and 12.7% for **Synthetic2**. However, for all the three datasets, this was for the highest setting of α ($\alpha = 0.9$) - the number of false negatives for this were higher than for the other settings, for similar values of ϵ . One possible reason is that for $\alpha = 0.9$, an item that was 0.9-persistent had persistence very close to $0.9n$. Whereas, many of the items that were 0.3-persistent had persistence values that were much larger than $0.3n$. Items that have persistence values close to

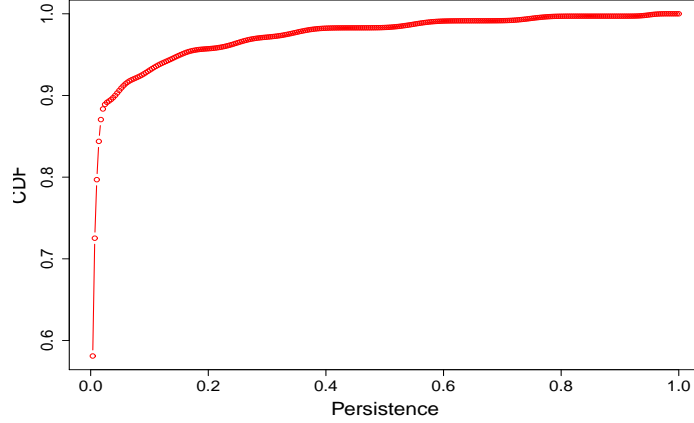


Figure 2.3: CDF of persistence values from the $[1,288]$ window for the **Synthetic2** dataset

the threshold, but higher than it, have a greater chance of not being reported than items whose persistence values are far above the threshold. Hence, the false negative ratio for $\alpha = 0.9$ is a little higher.

The **False Positive Rate**, similar to the False Negative Rate, shows (Figure 2.4b for **HeaderTrace**, Figure 2.5b for **Synthetic1** and Figure 2.6b for **Synthetic2**) an increasing trend as ϵ increases. The maximum FPR was 2.69% for **HeaderTrace** and 2.2% for **Synthetic2** (both for $\alpha = 0.3$ and $\epsilon = 0.21$). Moreover, all of Figures 2.4b, 2.5b and 2.6b show that for the same value of ϵ , the FPR is lower for higher values of α . The possible reason is that when α is very high (e.g. 0.9), most items have persistence much lower than αn (as is evident from the CDFs in Figures 2.1 and 2.2), hence are very unlikely to cross the threshold T in Algorithm 7.

The (Logical) **Space Compression** increases *linearly* with ϵ (Figures 2.4c for **HeaderTrace**, 2.5c for **Synthetic1** and 2.6c for **Synthetic2**), and we found the Space Compression is close to $\frac{1}{\tau} = \frac{\epsilon n}{2}$, for all values of α and all three datasets. This is expected since the naive algorithm creates a new tuple for an item everytime it appears in a different slot - where the small-space algorithm creates a tuple with probability τ only. For $\alpha = 0.9$, with $\epsilon = 0.63$, the logical space compression was as high as 32 for **HeaderTrace**, and as high as 91 for **Synthetic1** and ~ 100 for **Synthetic2**. The higher Space Compression for the synthetic datasets compared to **HeaderTrace** is justified by the larger value of the window length n (288 as opposed to 100).

For higher values of α , we could achieve better Space Compression as the tolerance ϵ could be

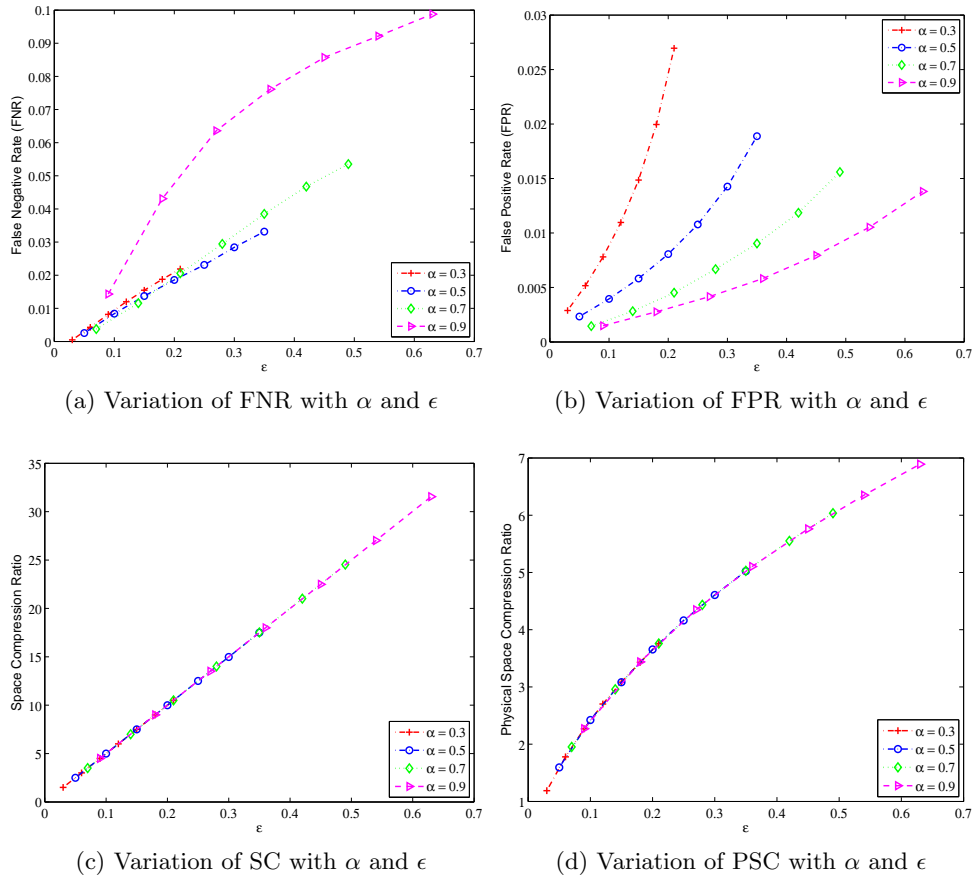


Figure 2.4: Trade-off between accuracy and space for the small-space algorithm over sliding windows for the **HeaderTrace** dataset. Each point in each plot is an average from 33 data points - 3 runs over 11 query windows each. Note that the Y-axis is different for each plot. Also, for each value of α , the values of ϵ range from 0.1α to 0.7α .

made higher while keeping the false positives and the false negatives small enough.

Like its logical counterpart, the **Physical Space Compression** also increases with ϵ (Figure 2.4d for **HeaderTrace**, 2.5d for **Synthetic1** and 2.6d for **Synthetic2**), and for each distinct value of α , the Physical Space Compression grows almost linearly with ϵ . For higher values of α , we could achieve better Space Compression as the tolerance ϵ could be made higher. While the size of the **HeaderTrace** dataset was 58 GB, the maximum resident set size of the naive algorithm went upto 3 GB (at the query window [251,350]), whereas for typical parameters like $\alpha = 0.5$ and $\epsilon = 0.35$, the small-space algorithm took less than $\frac{1}{5}^{th}$ (600 MB) memory (on average) compared to the naive algorithm. For **Synthetic1**, the dataset size was 12 GB, the maximum resident set size of the naive algorithm went upto 1.8 GB, whereas for $\alpha = 0.5$

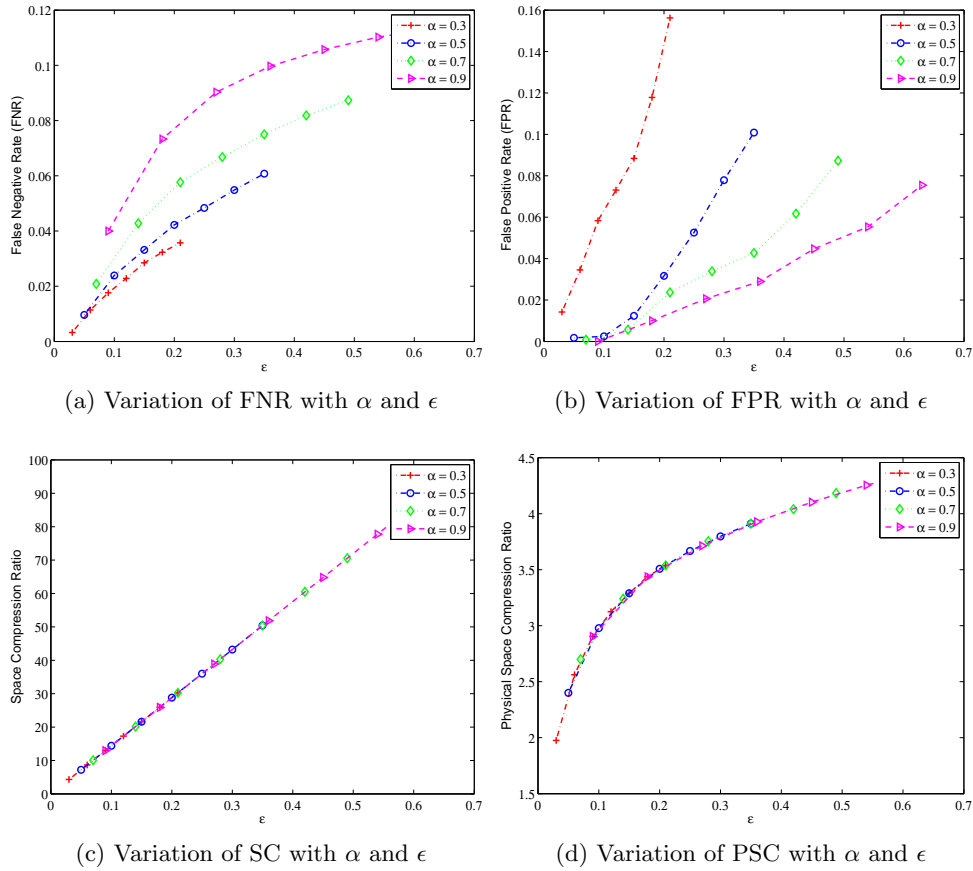


Figure 2.5: Trade-off between accuracy and space for the small-space algorithm over sliding windows for the **Synthetic1** dataset. Each point in each plot is an average from 30 data points - 3 runs over 10 query windows each. The Y-axis is different for each plot. For each value of α , the values of ϵ range from 0.1α to 0.7α .

and $\epsilon = 0.35$, the small-space algorithm took space between 350 and 500 MB. For **Synthetic2**, the dataset size was 1.5 GB, the maximum resident set size of the naive algorithm went up to 736 MB, whereas for $\alpha = 0.5$ and $\epsilon = 0.35$, the small-space algorithm took space between 140 and 200 MB.

Variation with ϵ , seed and query window: Figures 2.7a through 2.9c take a closer look at some of the absolute numbers (actual memory used, number of true and false positives, number of true and false negatives) rather than ratios for the **Synthetic1** dataset, and show how they vary with ϵ , the seed value of the random number generator and the different query windows. Figure 2.7a shows how the physical memory (in KB) varies with ϵ for $\alpha = 0.5$ and the query window [2593, 2880]. Since the memory taken by the naive algorithm does not depend on α or

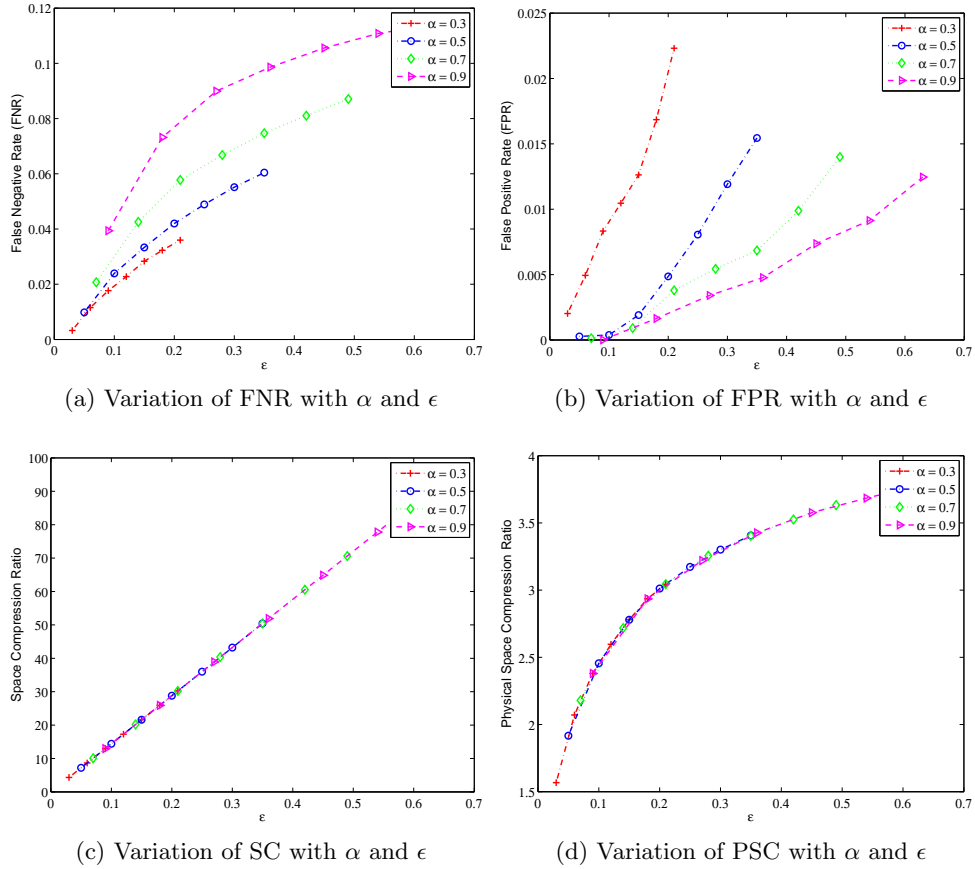


Figure 2.6: Trade-off between accuracy and space for the small-space algorithm over sliding windows for the **Synthetic2** dataset. Other details are same as **Synthetic1**.

ϵ , it is constant throughout at 1.86 GB, and the memory taken by the small space algorithm falls from 800 MB to 500 MB as ϵ increases from 0.05 to 0.35.

In Figure 2.7b, the actual number of persistent items in the window [2593, 2880] is constant at 235,353; and we can see that with increasing ϵ , the number of true positives reduces only a little, remaining very close to the number of actual persistent items throughout. The number of false positives is as low as $\sim 3,600$ when $\epsilon = 0.05$, and for typical values of ϵ (e.g., 0.15) that will probably be used in practice for a problem like this), the number of false positives is $\sim 25k$.

In Figure 2.7c, the actual number of transient items in the window [2593, 2880] is constant at $\sim 2.1m$; and we can see that with increasing ϵ , the number of true negatives reduces only a little, remaining very close to the number of actual transient items throughout. The number of false negatives is as low as $\sim 2,200$ when $\epsilon = 0.05$, and even when ϵ is as high as 0.35, the number

of false negatives increases only to $\sim 14k$. Note that, for many practical applications, it is important to keep the number/rate of false negatives much lower compared to the number/rate of false positives, and comparing Figure 2.7b and 2.7c shows that our algorithm meets that criterion.

Figure 2.8a shows how the physical memory (in KB) varies with the seed of the random number generator for $\alpha = 0.5$, $\epsilon = 0.15$ and the query window [2593, 2880]. Since the memory taken by the naive algorithm does not depend on the seed of the random number generator, it is constant throughout at 1.86 GB, and the memory taken by the small space algorithm also remains practically constant at $\sim 593MB$, which justifies averaging the actual memory footprint over the 3 different seed values.

In Figure 2.8b, the actual number of persistent items in the window [2593, 2880] is constant at 235,353; and we can see that with change in the seed, the number of true positives remains practically constant at $\sim 227k$, and so does the number of false positives at $\sim 26k$.

In Figure 2.8c, the actual number of transient items in the window [2593, 2880] is constant at $\sim 2.1m$; and we can see that with change in the seed, the number of true negatives remains practically constant at $\sim 2.07m$, and so does the number of false negatives at $\sim 7.8k$.

Figure 2.9a shows how the physical memory (in KB) varies with the query window for $\alpha = 0.5$ and $\epsilon = 0.15$, the seed of the random number generator being 10. Unlike Figure 2.7a or Figure 2.8a, the physical memory depends on the number of distinct items in the window, and although we generated the items uniformly across the slot range [1, 2880], we see it increased gradually with increasing slot number. However, while the space taken by the naive algorithm varied from 1.5 GB to 1.86 GB, the space taken by the small-space algorithm varied from ~ 400 MB to $\sim 600MB$.

Figure 2.9b shows how the number of persistent items varies with the query window for $\alpha = 0.5$ and $\epsilon = 0.15$, the seed of the random number generator being 10. Since for the **Synthetic1** dataset, each persistent item was distributed uniformly across the slot range [1, 2880], the actual number of persistent items across the different windows was almost constant at $\sim 235k$; and we can see that with change in the query window, the number of true positives also remains practically constant at $\sim 227k$, and so does the number of false positives at $\sim 26k$.

Figure 2.9c shows how the number of transient items varies with the query window for $\alpha = 0.5$ and $\epsilon = 0.15$, the seed of the random number generator being 10. For the **Synthetic1** dataset, like the persistent items, each transient item was also distributed uniformly across the slot range $[1, 2880]$, hence the actual number of transient items across the different query windows is practically constant at $\sim 2.1\text{m}$; and we can see that with change in the query window, the number of true negatives remains practically constant at $\sim 2.07\text{m}$, and so does the number of false negatives at $\sim 7.8\text{k}$.

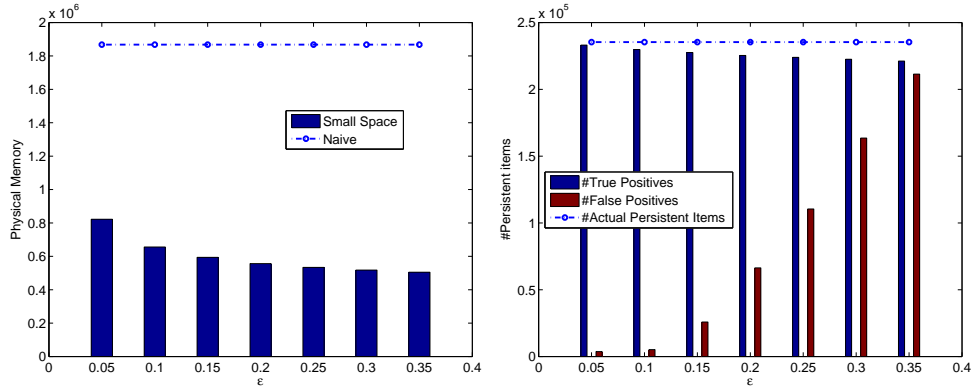
The small variation in the actual memory used, the number of true and false positives and the number of true and false negatives by the small-space algorithm, as shown in Figures 2.8a through 2.9c justifies our averaging of these quantities across the different seed values and query windows.

Comparison among three datasets: The FPR for **HeaderTrace** was much lower than that for **Synthetic1** at comparable points, e.g., at $\alpha = 0.3, \epsilon = 0.21$ and in comparable query windows, for **Synthetic1**, the FPR is 16%, whereas for **HeaderTrace**, the FPR is 2.7%. For **Synthetic1**, at an identical query window, there are $\sim 300\text{k}$ false positives out of $\sim 1931\text{k}$ transient items, and for **HeaderTrace**, there are $\sim 21\text{k}$ false positives out of $\sim 802\text{k}$ transient items. The lower FPR for **HeaderTrace** probably arises out of the fact that the CDF curve (Figure 2.1) grows more steeply than the CDF curve for **Synthetic1** (Figure 2.2), so the fraction of items whose persistence come anywhere close to 0.3 is much less. To demonstrate the difference among the three datasets, we present the 93rd percentile value of persistence for each. For **HeaderTrace**, there are total $\sim 829\text{k}$ distinct items in the window $[251, 350]$, but $\sim 770\text{k}$ of these items occur in 10 or less slots out of 100, i.e., the 93rd percentile value of persistence is 10%. As a comparison, for **Synthetic1**, there are total $\sim 2330\text{k}$ distinct items in the window $[2593, 2880]$, but $\sim 2167\text{k}$ of these items occur in 161 or less slots out of 288 ($161/288 = 56\%$), i.e., the 93rd percentile value of persistence is 56%. But then again, for **Synthetic2**, the FPR improves significantly over **Synthetic1** - for $\alpha = 0.3$ and $\epsilon = 0.21$, **Synthetic2** gives an FPR of 2.2% in pretty much all query windows - so the FPR becomes comparable to that of **HeaderTrace**. To compare with **Synthetic1**, there are $\sim 1394\text{k}$ distinct items in the window $[2593, 2880]$, but $\sim 1296\text{k}$ of them occur in 28 or less slots out of 288, i.e., the 93rd percentile

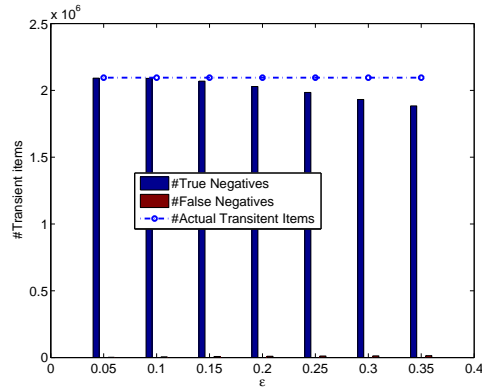
value of persistence is 9.7% (see Figure 2.3). The reason for lower FPR for **Synthetic2** is similar to that for **HeaderTrace**. This shows that the FPR improves with skewness of the data, and our algorithm in fact performs well for datasets with realistic skewness.

The physical memory compression ratio is less for **Synthetic1** than for **HeaderTrace** for similar reasons - many transient items find room into the sketch for having persistence close to the threshold. Although the skew for **Synthetic2** is more than that of **Synthetic1**, it has similar values of logical and physical memory compressions since the proportion of items from the universe that have similar persistence values bear similar ratios to each other, e.g., for **Synthetic1**, 1% of the items have persistence 0.95 and 2% have persistence 0.75; whereas for **Synthetic2**, 0.1% of the items have persistence 0.95 and 0.2% have persistence 0.75 (first two rows of Table 2.1).

In Lemma 2.3.8, we showed that for a given value of ϵ and length of sliding window (n), the expected number of tuples in the sketch is proportional to the sum of the persistence values of all items appearing in the window ($\sum_{d \in D} p_d^\epsilon$). Hence, the physical memory taken should also vary with the sum of the persistence values. We present the following example to demonstrate this: in the $[1, 288]$ window, the number of distinct items for **Synthetic1** and **Synthetic2** are respectively $\sim 2.33\text{m}$ and $\sim 1.39\text{m}$, and the sum of persistence values for **Synthetic1** and **Synthetic2** respectively are about 20.47×10^4 and 1.4×10^4 . For $\alpha = 0.5$ and $\epsilon = 0.35$, the memory footprints by the small-space algorithm for this combination of parameters for **Synthetic1** and **Synthetic2** are respectively 353 MB and 142 MB, so **Synthetic1** takes about 2.5 times more memory than **Synthetic2**.

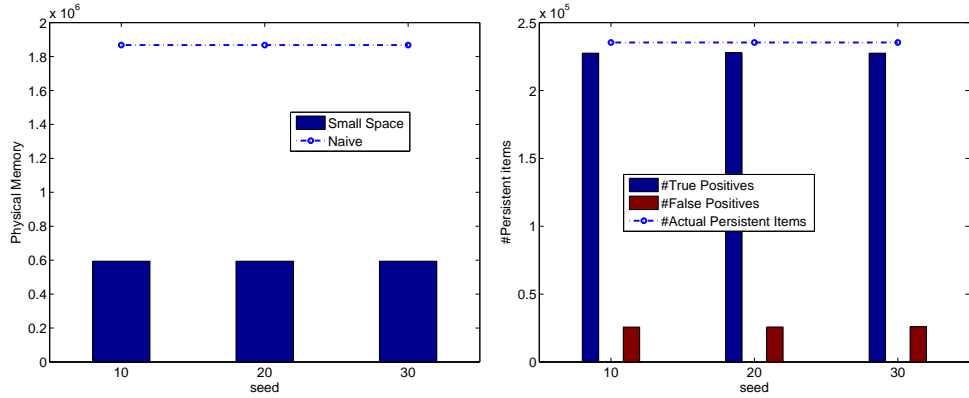


(a) Variation of actual memory used with ϵ (b) Variation of number of true positives and false positives with ϵ

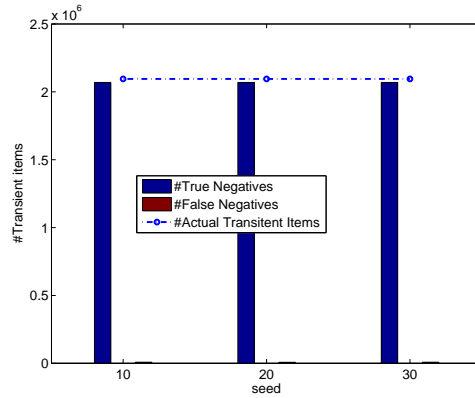


(c) Variation of number of true negatives and false negatives with ϵ

Figure 2.7: The variation of the physical memory taken, the number of true positives, false positives, true negatives and false negatives with ϵ for the **Synthetic1** dataset. All the plots are for $\alpha = 0.5$ and the query window $[2593, 2880]$. So, each point in each plot is an average from 3 data points corresponding to the 3 different seed values (10, 20, 30). Note that the horizontal lines in the three plots represent respectively the actual memory taken by the naive algorithm, the actual number of persistent items and the actual number of transient items, all measured in the same query window, and hence does not vary with ϵ . The Y-axis is different for each plot. The values of ϵ range from 0.1α to 0.7α .

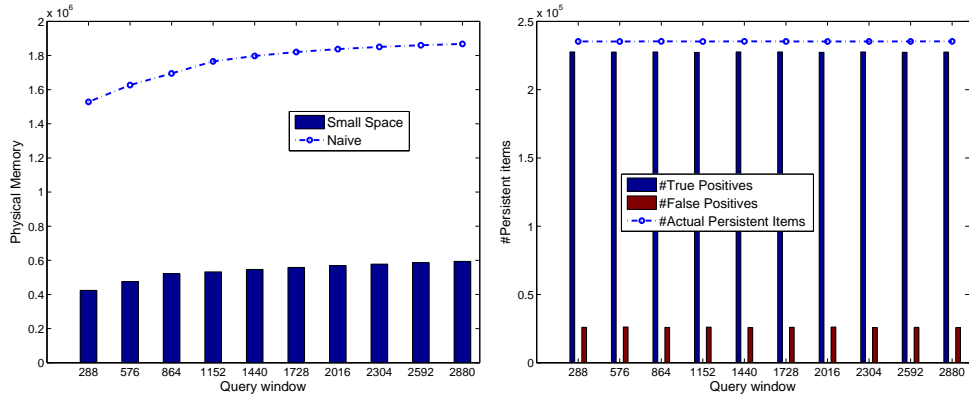


(a) Variation of actual memory used with seed (b) Variation of number of true positives and false positives with seed

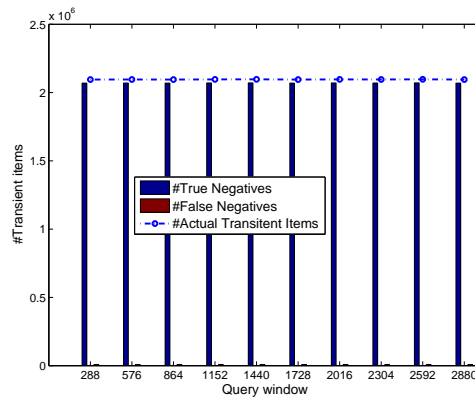


(c) Variation of number of true negatives and false negatives with seed

Figure 2.8: The variation of the physical memory taken, the number of true positives, false positives, true negatives and false negatives with the seed of the random number generator for the **Synthetic1** dataset. All the plots are for $\alpha = 0.5, \epsilon = 0.15$ and the query window $[2593, 2880]$. Note that the horizontal lines in the three plots represent respectively the actual memory taken by the naive algorithm, the actual number of persistent items and the actual number of transient items, all measured in the same query window, and hence does not vary with the seed. The Y-axis is different for each plot. The values of the seed used are 10, 20 and 30.



(a) Variation of actual memory used with query window (b) Variation of number of true positives and false positives with query window



(c) Variation of number of true negatives and false negatives with query window

Figure 2.9: The variation of the physical memory taken, the number of true positives, false positives, true negatives and false negatives with the query window for the **Synthetic1** dataset. All the plots are for $\alpha = 0.5, \epsilon = 0.15$ and seed = 10. Note that the horizontal lines in the three plots represent respectively the actual memory taken by the naive algorithm, the actual number of persistent items and the actual number of transient items - the first one shows slight increase with the progress of time (increasing query window number) but the other two are practically constant. The Y-axis is different for each plot. The query windows are $[1, 288], [289, 576], \dots, [2593, 2880]$ and the values on the X-axis are the endpoints of the query windows.

2.5 Related Work

A large body of literature on network anomaly detection has focused on detecting volume-based anomalies, i.e., tracking IPs which send or receive an unusually large volume of traffic over an interval of time. While volume-based anomaly detection is relevant for Denial-of-Service type attacks like SYN flood [108], UDP flood [109], Ping flood or P2P attacks, there are many “stealthy” attacks [54], which can bypass the radar by never sending traffic in large volume, yet remaining active over long windows in time, and probing the target network/host once in a while. For example, port scans [102] look for open ports on remote hosts that have applications with known vulnerabilities deployed on those ports; bots installed on compromised hosts in a botnet keep on communicating with the C&C server, etc. Our work differs from these in that persistent items may not result in large volumes of traffic and may escape detection by a volume-based system.

It is interesting to compare how algorithmic techniques for identifying heavy-hitters (or “frequent items”) may work for the problem of identifying persistent items. Broadly, the techniques in the literature can be classified into “counter-based”, “quantile algorithms”, “sketches”, or “random sampling-based” (see [31]). Counter-based techniques such as the Misra-Gries algorithm [89], and the “Space-Saving” algorithm [88] rely on maintaining per-item counters for counting the number of occurrences of each item that has been currently identified as being frequent; these counters are occasionally decremented to ensure that the space taken by the data structure is small. The difficulty in using this technique for our problem is that it is not easy to ensure that re-occurrences of the same item within a timeslot have no effect on the system state. For example, in the Misra-Gries algorithm, if there is a decrement of the counters between two occurrences of an item within the same timeslot, it seems hard to ensure that the second occurrence has no effect on the system state, especially given that the increment due to the first occurrence may have disappeared from the system (due to the decrement). The same argument is true for Lossy Counting too [87]. Quantile-based algorithms such as Greenwald and Khanna, or [62], the q-digest [100] view the space of all items as being a bijection with the set of integers, and associate counts with different ranges in this space of all items. In

the q-digest algorithm, there are no decrements to these counters, so one may use “distinct counters” such as those by Flajolet-Martin [51], or Gibbons and Tirthapura [57], or Kane, Nelson, and Woodruff[70], instead of regular counters. Such an approach based on maintaining distinct counters would not only be more complex than our approach, but also likely have a greater space complexity, since maintaining distinct counters with a relative error of ϵ requires $\Omega(1/\epsilon^2)$ space [68]. The sketch approach, such as count-sketch [28] or count-min sketch [36] also maintains multiple counters, each of which is the sum of many random variables. Replacing each such counter with a distinct counter leads to its own set of difficulties, one of which is the space complexity of distinct counting, explained above, and the other being the fact that each distinct counter is only approximate (exact distinct counting necessarily requires large space [13]), while the analyses in [28] and [36] rely on the different counters in the data structure being exact.

Finally, our algorithm is inspired by the random sampling approaches based on the “sample and count” scheme of Alon *et al.* [13, 12] and the “sticky sampling” algorithm of Manku and Motwani [87]. Both these algorithms use the following idea: “sample a random element in the stream, and track reoccurrences of this element exactly”. In these works, the idea was applied to a different context than ours – sample and count was applied to track the size of a self-join in limited storage, and sticky sampling was used in the identification of heavy hitters using limited space. Our algorithm has the following technical differences when compared with the above works. The sampling of an item is done using a hash function that is based on the item identifier and the timeslot in which it arrived in. This hash-based sampling avoids giving greater sampling probability to an item if it occurs multiple times within the same timeslot. Further, reoccurrences are tracked in such a way that we do not overcount if the same item appears again in the same timeslot. In addition, we show how to handle sliding windows using nearly the same space, while the above works do not address the context of sliding windows. A distinguishing aspect of our work on sliding windows is that while the extension to sliding windows often requires asymptotically greater space than for the infinite window case (for example, see Arasu and Manku [17]), in our case the space complexity increases only by a factor of two.

Persistence is exploited to detect botnet traffic in [59], using an algorithm that tracked the state of every distinct item that arrived within the sliding window. Hence the memory used is of the order of the number of distinct items times the window size, which is potentially very high. In contrast, our algorithm tracks persistent items using much smaller space, while giving up some accuracy.

There has been much work in estimating various properties of the frequency distribution of stream items, including the frequency moments of a stream [13, 112, 70], heavy-hitters [87, 50, 36, 84], and the entropy [83, 27, 94]. Unlike the set of persistent items, all the above properties depend only on the frequency distribution of items in the stream – they are unaffected by re-ordering of the stream elements, or by changing the times at which the elements arrive. In contrast, the set of persistent items in a stream is affected by the time and order in which elements arrive.

In a recent work on a temporal property of a stream, Chen *et al* [30] addressed the problem of tracking long-duration flows from network streams. They identified flows for which the difference of timestamps between the first and the last packet in the flow exceed some threshold d . A flow might continue for a long duration and yet the total number of bytes sent in the flow may not be high enough to be detected by the heavy-hitter algorithms; whereas some other flow of shorter duration might qualify as a heavy-hitter because it sends many more bytes. Clearly, a long-lived flow is not necessarily persistent.

2.6 Conclusion

We formulated the problem of detecting *persistent* items in a data stream. Our lower bound result shows that an exact algorithm for the problem, which reports *all* persistent items, would need a prohibitively high memory, and is therefore impractical. Subsequently, we presented an approximate formulation of the problem that explores a tradeoff between space and accuracy in identifying persistent items. Allocating more memory leads to more accurate answers and this allows operators to tune their systems appropriately depending on the amount of resources available.

By running simulations of both the naive (exact) and small space algorithms on a real as

well as two synthetic traffic datasets with different skewness, we demonstrate that our algorithm works very well in practice: for the real trace, it uses upto 85% *less* space than the naive (exact) algorithm and incurs a false positive rate (and false negative rate) of less than 1% (and 4% respectively) for typical values of the parameters. We also see that false positive rate never exceeds 3% for any parameter setting, while the false negative rate stays below 5% for all but the most aggressive thresholds for persistence. For the synthetic trace with low skewness, the small-space algorithm uses upto 80% less space than the naive one, the false positive rate is less than 2% and the false negative rate is about 4% for typical parameter values (e.g., $\alpha = 0.5$ and $\epsilon = 0.15$). The maximum false positive rate is less than 3% for the real trace and the synthetic trace with higher skewness. The empirical false positive and false negative rates, for most parameters, are much better than the analytical bounds: and our experiment across the three different datasets shows that the false positive rate improves for data with higher skewness.

CHAPTER 3. Identifying Correlated Heavy-Hitters over a Data Stream

In this chapter, we consider online mining of correlated heavy-hitters (CHH) from a data stream. Given a stream of two-dimensional data, a correlated aggregate query first extracts a substream by applying a predicate along a *primary* dimension, and then computes aggregates along a *secondary* dimension. We consider queries of the following form: “In a stream S of (x, y) tuples, on the substream H of all x values that are heavy-hitters, maintain those y values that occur frequently with the x values in H ”. This query arises naturally in situations where we need to track not only the identity of frequently occurring items in a stream, but also additional information associated with these items along other dimensions. Prior work on heavy-hitters in streams have focused solely on identifying the heavy-hitters on a single dimensional stream, and these yield little information about correlated heavy-hitters. We formulate an approximate version of the CHH problem, and present an algorithm for approximately tracking CHHs on a data stream. The algorithm is easy to implement and uses workspace which is orders of magnitude smaller than the stream itself. We present provable guarantees on the maximum error estimates, as well as experimental results that demonstrate the space-accuracy trade-off on a large stream of IP packet headers from a backbone network link.

3.1 Introduction

Correlated aggregates ([15, 55, 37]) reveal interesting interactions among the different attributes of a multi-dimensional dataset. They are useful when we are interested in finding an aggregate on an attribute over a subset, where the subset is defined by a selection predicate on a different attribute of the same dataset. On a stored database, a correlated aggregate can be computed by considering one dimension at a time, using multiple passes through the data.

However, for streaming data, we often do not have the luxury of making multiple passes over the dataset, the data may be too large to store and it is desirable to have an algorithm that works in a single pass through the data. Moreover, even the substream derived by applying the query predicate along the primary dimension can be too large to store, let alone the whole dataset.

We consider the identification of correlated heavy-hitters (CHHs) from massive data streams. We first define the notion of a heavy-hitter on a data stream (this is considered in prior work, such as [87, 89, 29, 35]), and then define our notion of correlated heavy-hitters. Given a sequence of single-dimensional records (a_1, a_2, \dots, a_N) , where $a_i \in \{1, \dots, m\}$, the frequency of an item i is defined as $|\{a_j | a_j = i\}|$. Given a user-input threshold $\phi \in (0, 1)$, any data item i whose frequency is at least ϕN is termed as a ϕ -heavy-hitter. We first consider the following problem of exact identification of CHHs.

Problem 4 Exact Identification of Correlated Heavy Hitters. *Given a data stream S of (x, y) tuples of length N (x and y will henceforth be referred to as the “primary” and the “secondary” dimensions, respectively), and two user-defined thresholds ϕ_1 and ϕ_2 , where $0 < \phi_1 < 1$ and $0 < \phi_2 < 1$, identify all (d, s) tuples such that:*

$$f_d = |\{(x, y) \in S : (x = d)\}| > \phi_1 N$$

and

$$f_{d,s} = |\{(x, y) \in S : (x = d) \wedge (y = s)\}| > \phi_2 f_d$$

The above aggregate can be understood as follows. The elements d are heavy-hitters in the traditional sense, on the stream formed by projecting along the primary dimension. For each heavy-hitter d along the primary dimension, there is logically a (uni-dimensional) substream S_d , consisting of all values along the secondary dimension, where the primary dimension equals d . We require the tracking of all tuples (d, s) such that s is a heavy-hitter in S_d .

Many stream mining and monitoring problems on two-dimensional streams need the CHH aggregate, and cannot be answered by independent aggregation along single dimensions. For example, consider a network monitoring application, where a stream of (destination IP address,

source IP address) pairs is being observed. The network monitor maybe interested not only in tracking those destination IP addresses that receive a large fraction of traffic (heavy-hitter destinations), but also in tracking those source IP addresses that send a large volume of traffic to these heavy-hitter destinations. This cannot be done by independently tracking heavy-hitters along the primary and the secondary dimensions. Note that in this application, we are interested not only in the identity of the heavy-hitters, but also additional information on the substream induced by the heavy-hitters.

In another example, in a stream of (server IP address, port number) tuples, identifying the heavy-hitter server IP addresses will tell us which servers are popular, and identifying frequent port numbers (independently) will tell us which applications are popular; but a network manager maybe interested in knowing which applications are popular among the heavily loaded servers, which can be retrieved using a CHH query. Such correlation queries are used for network optimization and anomaly detection [39].

Another application is the recommendation system of a typical online shopping site, which shows a buyer a list of the items frequently bought with the ones she has decided to buy. Our algorithm can optimize the performance of such a system by parsing the transaction logs and identifying the items that were bought commonly with the frequently purchased items. If such information is stored in a cache with a small lookup time, then for most buyers, the recommendation system can save the time to perform a query on the disk-resident data.

Similar to the above examples, in many stream monitoring applications, it is important to track the heavy-hitters in the stream, but this monitoring should go beyond simple identification of heavy-hitters, or tracking their frequencies, as is considered in most prior formulations of heavy-hitter tracking such as [34, 87, 89, 29, 49]. In this work we initiate the study of tracking additional properties of heavy-hitters by considering tracking of correlated heavy hitters.

3.1.1 Approximate CHH

It is easy to prove that exact identification of heavy-hitters in a single dimension is impossible using limited space, and one pass through the input. Hence, the CHH problem is also impossible to solve in limited space, using a single pass through the input. Due to this, we

consider the following approximate version of the problem. We introduce additional approximation parameters, ϵ_1 and ϵ_2 ($0 < \epsilon_1 \leq \frac{\phi_1}{2}$, $0 < \epsilon_2 < \phi_2$), which stand for the approximation errors along the primary and the secondary dimensions, respectively. We seek an algorithm that provides the following guarantees.

Problem 5 Approximate Identification of Correlated Heavy-Hitters. *Given a data stream S of (d, s) tuples of length N , thresholds ϕ_1 and ϕ_2 :*

1. *Report any value d such that $f_d > \phi_1 N$ as a heavy-hitter along the primary dimension.*
2. *No value d such that $f_d < (\phi_1 - \epsilon_1)N$, should be reported as a heavy-hitter along the primary dimension.*
3. *For any value d reported above, report any value s along the secondary dimension such that $f_{d,s} > \phi_2 f_d$ as a CHH.*
4. *For any value d reported above, no value s along the secondary dimension such that $f_{d,s} < (\phi_2 - \epsilon_2)f_d$ should be reported as a CHH occurring along with d .*

With this problem formulation, false positives are possible, but false negatives are not. In other words, if a pair (d, s) is a CHH according to the definition in Problem 4, then it is a CHH according to the definition in Problem 5, and will be returned by the algorithm. But an algorithm for Problem 5 may return a pair (s, d) that are not exact CHHs, but whose frequencies are close to the required thresholds.

3.1.2 Contributions

Our contributions are as follows.

- We introduce the notion of exact and approximate CHHs, and present a small-space approximation algorithm for identifying approximate CHHs in a single pass. Prior literature on correlated aggregates have mostly focused on the correlated sum, and these techniques are not applicable for CHH. Our algorithm for approximate CHH identification is based on a nested application of the Misra-Gries algorithm [89].

- We provide a provable guarantee on the approximation error. We show that there are no false negatives, and the error in the false positives is controlled. When greater memory is available, this error can be reduced. The space taken by the algorithm as well as the approximation error of the algorithm depend on the sizes of two different data structures within the algorithm. The total space taken by the sketch is minimized through solving a constrained optimization problem that minimizes the total space taken subject to providing the user-desired error guarantees.
- We present results from our simulations on a stream of more than 1.4 billion (50 GB trace) anonymized packet headers from an OC48 link (collected by CAIDA [25]). We compared the performance of our small-space algorithm with a slow, but exact algorithm that goes through the input data in multiple passes. Our experiments revealed that even with a space budget of a few megabytes, the average error statistics of our algorithm were very small, showing that it is a viable algorithm in practice.

Along each dimension our algorithm maintains frequency estimates of mostly those values (or pairs of values) that occur frequently. For example, for every destination that sends a significant fraction of traffic on a link, we maintain mostly the sources that occur frequently along with this destination. Note that the set of heavy-hitters along the primary dimension can change as the stream elements arrive, and this influences the set of CHHs along the secondary dimension. For example, if an erstwhile heavy-hitter destination d no longer qualifies as a heavy-hitter with increase in N (and hence gets rejected from the sketch), then a source s occurring with d should also be discarded from the sketch. This interplay between different dimensions has to be handled carefully during algorithm design.

Roadmap: The rest of this paper is organized as follows. In Section 3.2, we present related work, followed by the Algorithm in Section 3.3, and its proof of correctness in Section 3.4. The analysis of the space complexity and the setting of certain important parameters of the algorithm is discussed in Section 3.5, followed by experimental results in Section 4.5.

3.2 Related Work

In the data streaming literature, there is a significant body of work on correlated aggregates ([15, 55, 37]), as well as on the identification of heavy hitters ([87, 89, 29, 35]). See [32] for a recent overview of work on heavy-hitter identification. None of these works consider correlated heavy-hitters.

Estan *et al.* [48] and Zhang *et al.* [118] have independently studied the problem of identifying heavy-hitters from multi-dimensional packet streams, but they both define a multidimensional tuple as a heavy-hitter if it occurs more than ϕN times in the stream, N being the stream size – the interplay across different dimensions is not considered.

Gehrke *et al* [55] addressed correlated aggregates where the aggregate along the primary dimension was an extremum (min or max) or the average, and the aggregate along the secondary dimension was sum or count. For example, given a stream S of (x, y) tuples, their algorithm could approximately answer queries of the following form: “Return the sum of y -values from S where the corresponding x -values are more than twice the minimum of all x -values”. They maintained a data structure called *adaptive histograms*, but these did not come with provable guarantees on performance. Ananthakrishna *et al* [15] presented algorithms with provable error bounds for correlated sum and count. Given a stream S of (x, y) tuples, their algorithms could answer, approximately, prefix-range queries of the following form: “Return the sum (or count) of y -values from S where the corresponding x -values are at most x^* ”. Here, x^* is a value that is presented at query time. Their solution was based on the quantile summary of [62]. Cormode, Tirthapura, and Xu [37] presented algorithms for maintaining the more general case of *time-decayed* correlated aggregates, where the stream elements were weighted based on the time of arrival. This work also addressed the “sum” aggregate. A recent work by Tirthapura and Woodruff [103] presents a general method for estimating correlated aggregates for a class of aggregates that satisfy a certain set of conditions.

The above works differ from our work in the following respect. In our case the constraint on the x dimension is of the form “the frequency of the x value selected is large”, while the constraints in the previous works [55, 15, 37, 103] are of the form $x > x^*$ or $x < x^*$. These

two types of constraints require significantly different techniques. For instance in the case of the constraint of the form $x > x^*$, we know that the interval of x that is of interest always has the right endpoint equal to the maximum x value; and the techniques in [103, 37] make use of this by storing summaries over a nested sequence of intervals with the right endpoint equal to the maximum x value. But the x values of interest in our case do not necessarily even form a contiguous subsequence of the universe.

The heavy-hitter literature has usually focused on the following problem. Given a sequence of elements $A = (a_1, a_2, \dots, a_N)$ and a user-input threshold $\phi \in (0, 1)$, find data items that occur more than ϕN times in A . Misra and Gries [89] presented a deterministic algorithm for this problem, with space complexity being $O(\frac{1}{\phi})$, time complexity for updating the sketch with the arrival of each element being $O(\log \frac{1}{\phi})$, and query time complexity being $O(\frac{1}{\phi})$. For exact identification of heavy-hitters, their algorithm works in two passes. For approximate heavy-hitters, their algorithm used only one pass through the sequence, and had the following approximation guarantee. Assume user-input threshold ϕ and approximation error $\epsilon < \phi$. Note that for an online algorithm, N is the number of elements received so far.

- All items whose frequencies exceed ϕN are output. i.e. there are no false negatives.
- No item with frequency less than $(\phi - \epsilon)N$ is output.

Demaine *et al* [43] and Karp *et al* [71] improved the sketch update time per element of the Misra-Gries algorithm from $O(\log \frac{1}{\phi})$ to $O(1)$, using an advanced data structure combining a hashtable, a linked list and a set of doubly-linked lists. Manku and Motwani [87] presented a deterministic “Lossy Counting” algorithm that offered the same approximation guarantees as the one-pass approximate Misra-Gries algorithm; but their algorithm required $O(\frac{1}{\epsilon} \log(\epsilon N))$ space in the worst case. For our problem, we chose to extend the Misra-Gries algorithm as it takes asymptotically less space than [87].

3.3 Algorithm

Our algorithm is based on a nested application of an algorithm for identifying frequent items from an one-dimensional stream, due to Misra and Gries [89]. We first describe the

Misra-Gries algorithm (henceforth called the MG algorithm). Suppose we are given an input stream a_1, a_2, \dots , and an error threshold $\epsilon, 0 < \epsilon < 1$. The algorithm maintains a data structure \mathcal{D} that contains at most $\frac{1}{\epsilon}$ (key, count) pairs. On receiving an item a_i , it is first checked if a tuple (a_i, \cdot) already exists in \mathcal{D} . If it does, a_i 's count is incremented by 1; otherwise, the pair $(a_i, 1)$ is added to \mathcal{D} . Now, if adding a new pair to \mathcal{D} makes $|\mathcal{D}|$ exceed $\frac{1}{\epsilon}$, then for each (key, count) pair in \mathcal{D} , the count is decremented by one; and any key whose count falls to zero is discarded. This ensures at least the key which was most recently added (with a count of one) would get discarded, so the size of \mathcal{D} , after processing all pairs, would come down to $\frac{1}{\epsilon}$ or less. Thus, the space requirement of this algorithm is $O(\frac{1}{\epsilon})$. The data structure \mathcal{D} can be implemented using hashables or height-balanced binary search trees. At the end of one pass through the data, the MG algorithm maintains the frequencies of keys in the stream with an error of no more than ϵn , where n is the size of the stream. The MG algorithm can be used in exact identification of heavy hitters from a data stream using two passes through the data.

In the scenario of limited memory, the MG algorithm can be used to solve problem 4 in three passes through the data, as follows. We first describe a four pass algorithm. In the first two passes, heavy-hitters along the primary dimension are identified, using memory $O(1/\phi_1)$. Note that this is asymptotically the minimum possible memory requirement of any algorithm for identifying heavy-hitters, since the size of output can be $\Omega(\frac{1}{\phi_1})$. In the next two passes, heavy-hitters along the secondary dimension are identified for each heavy-hitter along the primary dimension. This takes space $O(\frac{1}{\phi_2})$ for each heavy-hitter along the primary dimension. The total space cost is $O(\frac{1}{\phi_1\phi_2})$, which is optimal, since the output could be $\Omega(\frac{1}{\phi_1\phi_2})$ elements. The above algorithm can be converted into a *three* pass exact algorithm by combining the second and third passes.

The high-level idea behind our single-pass algorithm for Problem 5 is as follows. The MG algorithm for an one-dimensional stream, can be viewed as maintaining a small space “sketch” of data that (approximately) maintains the frequencies of each distinct item d along the primary dimension; of course, these frequency estimates are useful only for items that have very high frequencies. For each distinct item d along the primary dimension, apart from maintaining its frequency estimate \hat{f}_d , our algorithm maintains an embedded MG sketch of the substream

S_d induced by d , i.e. $S_d = \{(x, y) | ((x, y) \in S) \wedge (x = d)\}$. The embedded sketch is a set of tuples of the form $(s, \hat{f}_{d,s})$, where s is an item that occurs in S_d , and $\hat{f}_{d,s}$ is an estimate of the frequency of the pair (d, s) in S (or equivalently, the frequency of s in S_d). While the actions on \hat{f}_d (increment, decrement, discard) depend on how d and the other items appear in S , the actions on $\hat{f}_{d,s}$ depend on the items appearing in S_d . Further, the sizes of the tables that are maintained have an important effect on both the correctness and the space complexity of the algorithm.

We now present a more detailed description. The algorithm maintains a table H , which is a set of tuples (d, \hat{f}_d, H_d) , where d is a value along the primary dimension, \hat{f}_d is the estimated frequency of d in the stream, and H_d is another table that stores the values of the secondary attribute that occur with d . H_d stores its content in the form of (key, count) pairs, where the keys are values (s) along the secondary attribute and the counts are the frequencies of s in S_d , denoted as $\hat{f}_{d,s}$, alongwith d .

The maximum number of tuples in H is s_1 , and the maximum number of tuples in each H_d is s_2 . The values of s_1 and s_2 depend on the parameters $\phi_1, \phi_2, \epsilon_1, \epsilon_2$, and are decided at the start of the algorithm. Since s_1 and s_2 effect the space complexity of the algorithm, as well as the correctness guarantees provided by it, their values are set based on an optimization procedure, as described in Section 3.5.

The formal description is presented in Algorithms 8, 9 and 10. Before a stream element is received, Algorithm 8 Sketch-Initialize is invoked to initialize the data structures. Algorithm 9 Sketch-Update is invoked to update the data structure as each stream tuple (x, y) arrives. Algorithm 10 Report-CHH is used to answer queries when a user asks for the CHHs in the stream so far.

On receiving an element (x, y) of the stream, the following three scenarios may arise. We explain the action taken in each.

1. If x is present in H , and y is present in H_x , then both \hat{f}_x and $\hat{f}_{x,y}$ are incremented.
2. If x is present in H , but y is not in H_x , then y is added to H_x with a count of 1. If this addition causes $|H_x|$ to exceed its space budget s_2 , then for each (key, count) pair in H_x ,

the count is decremented by 1 (similar to the MG algorithm). If the count of any key falls to zero, the key is dropped from H_x . Note that after this operation, the size of H_x will be at most s_2 .

3. If x is not present in H , then an entry is created for x in H by setting \hat{f}_x to 1, and by initializing H_x with the pair $(y, 1)$. If adding this entry causes $|H|$ to exceed s_1 , then for each $d \in H$, f_d is decremented by 1. If the decrement causes \hat{f}_d to be zero, then we simply discard the entry for d from H .

Otherwise, when f_d is decremented, the algorithm keeps the sum of the $\hat{f}_{d,s}$ counts within H_d equal to f_d ; the detailed correctness is proved in Section 3.5. To achieve this, an arbitrary key s is selected from H_d such that $\hat{f}_{d,s} > 0$, and $\hat{f}_{d,s}$ is decremented by 1. If $\hat{f}_{d,s}$ falls to zero, s is discarded from H_d .

Algorithm 8: Sketch-Initialize($\phi_1, \phi_2, \epsilon_1, \epsilon_2$)

Input: Threshold for primary dimension ϕ_1 ; Threshold for secondary dimension ϕ_2 ;
Tolerance for primary dimension ϵ_1 ; Tolerance for secondary dimension ϵ_2

- 1 $H \leftarrow \Phi$
 - 2 Set s_1 and s_2 as described in Section 3.5.
-

Algorithm 9: Sketch-Update(x, y)**Input:** Element along primary dimension x ; Element along secondary dimension y

```

1 if  $x \in H$  then
2    $\hat{f}_x \leftarrow \hat{f}_x + 1$ ;
3   if  $y \in H_x$  then
4     /* Both  $x$  and  $y$  are present */
5     Increment  $\hat{f}_{x,y}$  in  $H_x$  by 1;
6   else
7     /*  $x \in H$ , but  $y \notin H_x$  */
8     Add the tuple  $(y, 1)$  to  $H_x$ ;
9     if  $|H_x| > s_2$  then
10      foreach  $(s, \hat{f}_{d,s}) \in H_x$  do
11         $\hat{f}_{d,s} \leftarrow \hat{f}_{d,s} - 1$ ;
12        if  $\hat{f}_{d,s} = 0$  then
13          discard  $(s, \hat{f}_{d,s})$  from  $H_x$ ;
14        end
15      end
16 end
17 else
18   /* Neither of  $x$  or  $y$  is present */
19    $H_x \leftarrow \Phi$ ; Add  $(y, 1)$  to  $H_x$ ;  $\hat{f}_x \leftarrow 1$ ;
20   if  $|H| > s_1$  then
21     foreach  $d \in H$  do
22        $\hat{f}_d \leftarrow \hat{f}_d - 1$ ;
23       if there exists  $s$  such that  $\hat{f}_{d,s} > 0$  then
24         Choose an arbitrary  $(s, \hat{f}_{d,s}) \in H_d$  such that  $\hat{f}_{d,s} > 0$ ;
25          $\hat{f}_{d,s} \leftarrow \hat{f}_{d,s} - 1$ ;
26         if  $\hat{f}_{d,s} = 0$  then
27           discard  $(s, \hat{f}_{d,s})$  from  $H_d$ ;
28         end
29       end
30       if  $\hat{f}_d = 0$  then
31         Discard  $(d, H_d)$  from  $H$ ;
32       end
33 end

```

Algorithm 10: Report-CHH(N)**Input:** Size of the stream N

```

1 foreach  $d \in H$  do
2   if  $\hat{f}_d \geq (\phi_1 - \frac{1}{s_1})N$  then
3     Report  $d$  as a frequent value of the primary attribute;
4     foreach  $(s, \hat{f}_{d,s}) \in H_d$  do
5       if  $\hat{f}_{d,s} \geq (\phi_2 - \frac{1}{s_2})\hat{f}_d - \frac{N}{s_1}$  then
6         Report  $s$  as a CHH occurring with  $d$ ;
7       end
8     end
9   end
10 end

```

3.4 Correctness

In this section, we show the correctness of the algorithm, subject to the following constraints on s_1 and s_2 . In Section 3.5, we assign values to s_1 and s_2 in such a manner that the space taken by the data structure is minimized.

Constraint 3.4.1

$$\frac{1}{s_1} \leq \epsilon_1$$

Constraint 3.4.2

$$\frac{1}{s_2} + \frac{1 + \phi_2}{s_1(\phi_1 - \epsilon_1)} \leq \epsilon_2$$

Consider the state of the data structure after a stream S of length N has been observed. Consider a value d of the primary attribute, and s of the secondary attribute. Let f_d and $f_{d,s}$ be defined as in Section 3.1. Our analysis focuses on the values of variables \hat{f}_d and $\hat{f}_{d,s}$, which are updated in Algorithms 9 and used in Algorithm 10. For convenience, if d is not present in H then we define $\hat{f}_d = 0$. Similarly, if d is not present in H , or if (d, s) is not present in H_d , then we define $\hat{f}_{d,s} = 0$.

Lemma 3.4.1

$$\hat{f}_d \geq f_d - \frac{N}{s_1}$$

Proof: The total number of increments in the s_1 counters that keep track of the counts of the different values of the primary attribute is N . Each time there is a decrement to \hat{f}_d (in Line 20 of Algorithm 9), $s_1 + 1$ different counters are decremented. The total number of decrements, however, cannot be more than the total number of increments, and hence is at most N . So the number of times the block of lines 19-31 in Algorithm 9 gets executed is at most $\frac{N}{s_1+1} < \frac{N}{s_1}$. We also know that \hat{f}_d is incremented exactly f_d times, hence the final value of \hat{f}_d is greater than $f_d - \frac{N}{s_1}$. ■

Lemma 3.4.2 *Assume that Constraint is true. If $f_d > \phi_1 N$, then d is reported by Algorithm 10 as a frequent item. Further, if $f_d < (\phi_1 - \epsilon_1)N$, then d is not reported as a frequent item.*

Proof: Suppose $f_d \geq \phi_1 N$. From Lemma 3.4.1, $\hat{f}_d \geq f_d - \epsilon_1 N \geq \phi_1 N - \epsilon_1 N$. Hence Algorithm 10 will report d (see Lines 2 and 3). Next, suppose that $f_d < (\phi_1 - \epsilon_1)N$. Since $\hat{f}_d \leq f_d$, Algorithm 10 will not report d as a frequent item. ■

Lemma 3.4.3

$$\sum_{(s, \cdot) \in H_d} \hat{f}_{d,s} \leq \hat{f}_d$$

Proof: Let $\Sigma_d = \sum_{(s, \cdot) \in H_d} \hat{f}_{d,s}$. Let $C(n)$ denote the condition $\Sigma_d \leq \hat{f}_d$ after n stream elements have been observed. We prove $C(n)$ by induction on n . The base case is when $n = 0$, and in this case, $\hat{f}_{d,s} = \hat{f}_d = 0$ for all d, s , and $C(0)$ is trivially true. For the inductive step, assume that $C(k)$ is true, for $k \geq 0$. Consider a new element that arrives, say (x, y) , and consider Algorithm 9 applied on this element. We consider four possible cases.

(I) If $x = d$, and $d \in H$, then \hat{f}_d is incremented by 1, and it can be verified (Lines 3-11) that Σ_d increases by at most 1 (and may even decrease). Thus $C(k+1)$ is true.

(II) If $x = d$, and $d \notin H$, then initially, \hat{f}_d and Σ_d are both 1 (line 17). If $|H| \leq s_1$, then both \hat{f}_d and Σ_d remain 1, and $C(k+1)$ is true. Suppose $|H| > s_1$, then both \hat{f}_d and Σ_d will go down to 0, since H_d will be discarded from H . Thus $C(k+1)$ is true.

(III) If $x \neq d$, and $x \in H$, then neither \hat{f}_d nor Σ_d change.

(IV) Finally, if $x \neq d$ and $x \notin H$, then it is possible that \hat{f}_d is decremented (line 20). In this case, if $\Sigma_d > 0$, then Σ_d is also decremented (line 22), and $C(k+1)$ is satisfied. If $\Sigma_d = 0$, then $C(k+1)$ is trivially satisfied since $\hat{f}_d \geq 0$. ■

Lemma 3.4.4 *Subject to Constraint , $\hat{f}_{d,s} \geq f_{d,s} - \epsilon_2 f_d - \epsilon_1 N$.*

Proof: Note that each time the tuple (d, s) occurs in the stream, $\hat{f}_{d,s}$ is incremented in Algorithm 9. But $\hat{f}_{d,s}$ can be less than $f_{d,s}$ because of decrements in Lines 9 or 23 in Algorithm 9. We consider these two cases separately.

Let $\Sigma_d = \sum_{(s, \cdot) \in H_d} \hat{f}_{d,s}$. For decrements in Line 9, we observe that each time this line is executed, Σ_d reduces by $s_2 + 1$. From Lemma 3.4.3, we know that $\Sigma_d \leq \hat{f}_d \leq f_d$. Thus the total number of times $\hat{f}_{d,s}$ is decremented due to Line 9 is no more than $\frac{f_d}{s_2+1}$. From Constraint 3.4.2, we know $\frac{1}{s_2} < \epsilon_2$, and $\frac{f_d}{s_2+1} < \epsilon_2 f_d$.

For decrements in Line 23, we observe that $\hat{f}_{d,s}$ is decremented in Line 23 no more than the number of decrements to \hat{f}_d , which was bounded by $\frac{N}{s_1}$ in Lemma 3.4.1. From Constraint 3.4.4, this is no more than $\epsilon_1 N$. ■

Lemma 3.4.5 *For any value d that gets reported in line 3 of Algorithm 10, any value s of the secondary attribute that occurs with d such that $f_{d,s} > \phi_2 f_d$, will be identified by line 6 of Algorithm 10 as a CHH occurring alongwith d .*

Proof: From Lemma 3.4.4,

$$\begin{aligned} \hat{f}_{d,s} &\geq f_{d,s} - \epsilon_2 f_d - \epsilon_1 N \\ &> \phi_2 f_d - \epsilon_2 f_d - \epsilon_1 N \\ &= (\phi_2 - \epsilon_2) f_d - \epsilon_1 N \\ &\geq (\phi_2 - \epsilon_2) \hat{f}_d - \epsilon_1 N \end{aligned}$$

where we have used $f_d \geq \hat{f}_d$. The lemma follows since $(\phi_2 - \epsilon_2) \hat{f}_d - \epsilon_1 N$ is the threshold used in line 5 of Algorithm 10 to report a value of the secondary attribute as a CHH. ■

Lemma 3.4.6 *Under Constraints 3.4.4 and 3.4.2, for any value of d that is reported as a heavy-hitter along the primary dimension, then for a value s' along the secondary dimension, if $f_{d,s'} < (\phi_2 - \epsilon_2)f_d$, then the pair (d, s') will not be reported as a CHH.*

Proof: We will prove the contrapositive of the above statement. Consider a value s such that (d, s) is reported as a CHH. Then, we show that $f_{d,s} \geq (\phi_2 - \epsilon_2)f_d$. If (d, s) is reported, then it must be true that $\hat{f}_{d,s} \geq (\phi_2 - \frac{1}{s_2})\hat{f}_d - \frac{N}{s_1}$ (Algorithm 10, line 5). Using $f_{d,s} \geq \hat{f}_{d,s}$, and $\hat{f}_d \geq f_d - \frac{N}{s_1}$, we get:

$$\begin{aligned}
f_{d,s} &\geq \hat{f}_{d,s} \\
&\geq \left(\phi_2 - \frac{1}{s_2}\right)\hat{f}_d - \frac{N}{s_1} \\
&\geq \left(\phi_2 - \frac{1}{s_2}\right)\left(f_d - \frac{N}{s_1}\right) - \frac{N}{s_1} \\
&= \left(\phi_2 - \frac{1}{s_2}\right)f_d - \frac{N}{s_1}\left(1 + \phi_2 - \frac{1}{s_2}\right) \\
&\geq \left(\phi_2 - \frac{1}{s_2}\right)f_d - \frac{f_d}{(\phi_1 - \epsilon_1)s_1}\left(1 + \phi_2 - \frac{1}{s_2}\right) \\
&\quad \text{(since } d \text{ gets reported, by Lemma 3.4.2, } f_d \geq (\phi_1 - \epsilon_1)N \Rightarrow N \leq \frac{f_d}{\phi_1 - \epsilon_1}\text{)} \\
&= \left(\phi_2 - \frac{1}{s_2} - \frac{1}{(\phi_1 - \epsilon_1)s_1}\left(1 + \phi_2 - \frac{1}{s_2}\right)\right)f_d \\
&\geq f_d(\phi_2 - \epsilon_2)\text{(using Constraint 3.4.2)}
\end{aligned}$$

■

Lemmas 3.4.6, 3.4.5, and 3.4.2 together yield the following.

Theorem 3.4.1 *If Constraints 3.4.4 and 3.4.2 are satisfied, then Algorithms 8, 9 and 10 satisfy all the four requirements of Problem 5.*

3.5 Analysis

In this section, we analyze the space complexity of the algorithm. In Theorem 3.4.1, we showed that the Algorithms 9 and 10 solve the Approximate CHH detection problem, as long as constraints 3.4.4 and 3.4.2 are satisfied.

Space Complexity in terms of s_1 and s_2 . In our algorithm, we maintain at most s_2 counters for each of the (at most) s_1 distinct values of the primary attribute in H . Hence, the size of our sketch is $O(s_1 + s_1 s_2) = O(s_1 s_2)$. We now focus on the following question. *What is the setting of s_1 and s_2 so that the space complexity of the sketch is minimized while meeting the constraints required for correctness.?*

Lemma 3.5.1 *Let $\alpha = \left(\frac{1+\phi_2}{\phi_1-\epsilon_1}\right)$. Subject to constraints 3.4.4 and 3.4.2, the space of the data structure is minimized by the following settings of s_1 and s_2 .*

- If $\epsilon_1 \geq \frac{\epsilon_2}{2\alpha}$, then $s_1 = \frac{2\alpha}{\epsilon}$ and $s_2 = \frac{2}{\epsilon_2}$. In this case, the space complexity is $O\left(\frac{1}{(\phi_1-\epsilon_1)\epsilon_2^2}\right)$.
- If $\epsilon_1 < \frac{\epsilon_2}{2\alpha}$, then $s_1 = \frac{1}{\epsilon_1}$, and $s_2 = \frac{1}{\epsilon_2-\alpha\epsilon_1}$. In this case, the space complexity is $O\left(\frac{1}{\epsilon_1\epsilon_2}\right)$.

Proof: Let $\sigma_1 = \frac{1}{s_1}$, $\sigma_2 = \frac{1}{s_2}$. The problem is now to maximize $\sigma_1\sigma_2$. Constraints 3.4.4 and 3.4.2 can be rewritten as follows.

- **Constraint 1:** $\sigma_1 \leq \epsilon_1$
- **Constraint 2:** $\alpha\sigma_1 + \sigma_2 \leq \epsilon_2$

First, we note that any assignment $(\sigma_1, \sigma_2) = (x, y)$ that maximizes $\sigma_1\sigma_2$ must be tight on Constraint 2, i.e. $\alpha x + y = \epsilon_2$. This can be proved by contradiction. Suppose not, and $\alpha x + y < \epsilon_2$, and xy is the maximum possible. Now, there is a solution $\sigma_1 = x$, and $\sigma_2 = y'$, such that $y < y'$, and Constraints 1 and 2 are still satisfied. Further, $xy' > xy$, showing that the solution (x, y) is not optimal.

Thus, we have:

$$\sigma_2 = \epsilon_2 - \alpha\sigma_1 \tag{3.1}$$

Thus the problem has reduced to: **Maximize** $f(\sigma_1) = \sigma_1(\epsilon_2 - \alpha\sigma_1)$ **subject to** $\sigma_1 \leq \epsilon_1$.

Consider

$$f'(\sigma_1) = \epsilon_2 - 2\alpha\sigma_1$$

We consider two cases.

- **Case I:** $\epsilon_1 \geq \frac{\epsilon_2}{2\alpha}$.

Setting $f'(\sigma_1) = 0$, we find that the function reaches a fixed point at $\sigma_1 = \frac{\epsilon_2}{2\alpha}$. At this point, $f''(\sigma_1) = -2\alpha$, which is negative. Hence $f(\sigma_1)$ is maximized at $\sigma_1 = \frac{\epsilon_2}{2\alpha}$. We note that this value of σ_1 does not violate Constraint 1, and hence this is a feasible solution. In this case, the optimal settings are: $\sigma_1 = \frac{\epsilon_2}{2\alpha}$ and $\sigma_2 = \frac{\epsilon_2}{2}$. Thus $s_1 = \frac{2\alpha}{\epsilon}$ and $s_2 = \frac{2}{\epsilon_2}$. The space complexity is $O(\frac{1}{\sigma_1\sigma_2}) = O(\frac{4\alpha}{\epsilon_2^2})$.

- **Case II:** $\epsilon_1 < \frac{\epsilon_2}{2\alpha}$

The function $f(\sigma_1)$ is increasing for σ_1 from 0 to $\frac{\epsilon_2}{2\alpha}$. Hence this will be maximized at the point $\sigma_1 = \epsilon_1$. Thus, in this case the optimal settings are $\sigma_1 = \epsilon_1$, and $\sigma_2 = \epsilon_2 - \alpha\epsilon_1$. Thus, $s_1 = \frac{1}{\epsilon_1}$, and $s_2 = \frac{1}{\epsilon_2 - \alpha\epsilon_1}$. The space complexity is: $O(\frac{1}{\epsilon_1(\epsilon_2 - \alpha\epsilon_1)})$.

We note that since $\epsilon_2 > 2\alpha\epsilon_1$, we have $(\epsilon_2 - \alpha\epsilon_1) > \frac{\epsilon_2}{2}$, and hence the space complexity is $O(\frac{1}{\epsilon_1\epsilon_2})$.

■

Lemma 3.5.2 *The time taken to update the sketch on receiving each element of the stream is $O(\max(s_1, s_2))$.*

Proof: In processing an element (x, y) of the stream by Algorithm 9, the following three scenarios may arise.

1. x is present in H , and y is present in H_x . We implemented the tables as hash tables, hence the time taken to look up and increment \hat{f}_x from H and $\hat{f}_{x,y}$ from H_x is $O(1)$.
2. x is present in H , but y is not in H_x . If the size of H_x exceeds its space budget s_2 , then, the time taken to decrement the frequencies of all the stored values of the secondary attribute is $O(s_2)$.
3. x is not present in H . If the size of H exceeds its space budget s_1 , then the time taken to decrement the frequencies of all the stored values of the primary attribute is $O(s_1)$.

The time complexity to update the sketch on receiving each element is the maximum of these three, which establishes the claim. ■

3.6 Experiments

We simulated our algorithm in C++, using the APIs offered by the Standard Template Library [10], on anonymized packet header traces collected by CAIDA [25] in both directions of an OC48 link. We conducted the experiments over two machines: one had Cygwin 5.1 (on Windows XP) with 3 GHz Pentium dual-core processor and 2 GB RAM, and the other had Red Hat Linux 5.3 (kernel version 2.6.18) with a 2.4 GHz Pentium 4 processor and 1 GB RAM. We used windump [11] on the first machine in conjunction with our custom Java application to extract the source IP address, the destination IP address, the source port number and the destination port number from the .pcap files.

Objective: The goal of the simulation was threefold: first, to learn about typical frequency distributions in real two-dimensional network traffic streams; second, to illustrate the reduction in space and time cost achievable by the small-space algorithm in practice; and finally, to demonstrate how the space budget (and hence, the allocated memory) influences the accuracy of our algorithm in practice.

For the *first* objective, we ran a naive algorithm on a smaller dataset of 248 million (destination IP, source IP) tuples, where all the distinct destination IPs were stored, and for each distinct destination IP, all the distinct source IPs were stored. We identified (exactly) the frequent values along both the dimensions for $\phi = 0.001$ and $\psi = 0.001$. Only 43 of the 1.2 million distinct destination IPs were reported as heavy-hitters. For the secondary dimension, we ranked the heavy-hitter destination IPs based on the number of distinct source IPs they co-occurred with, and the number of distinct source IPs for the top eight are shown in Figure 3.1. All these heavy-hitter destination IPs co-occurred with 9,000-18,000 *distinct* source IPs, whereas, for all of them, the number of co-occurring *heavy-hitter* source IPs was in the range 20-200 (note that the Y-axis in Figure 3.1 is in log scale). This shows that the distribution of the primary attribute values, as well as that of the secondary attribute values for a given value of the primary attribute, are very skewed, and hence call for the design of small-space

approximation algorithms like ours.

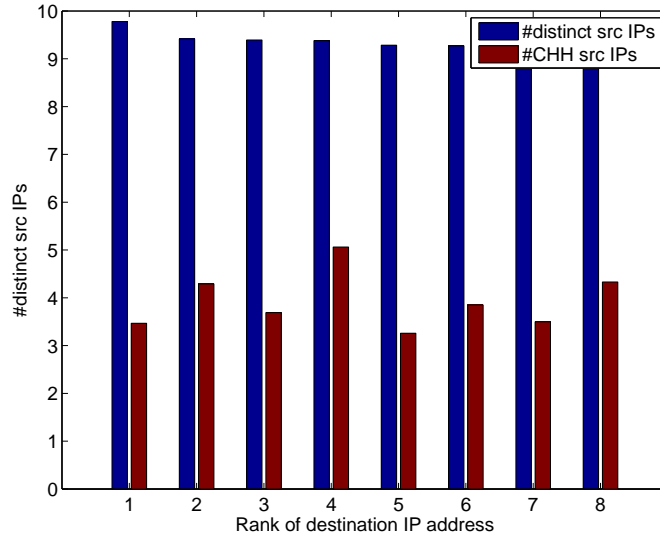


Figure 3.1: On the X-axis are the ranks of the eight (heavy-hitter) destination IPs, that co-appear with maximum number of distinct source IPs. For each destination IP, the Y-axis shows 1) the number of distinct source IPs co-occurring with it, 2) the number of heavy-hitter destination IPs co-appearing with it. Note that the Y-axis is logarithmic.

The *second* objective was accomplished by comparing the space and time costs of the naive algorithm as above (on the same dataset), with those of the small-space algorithm, run with $s_1 = 3000$ and $s_2 = 2000$ (Figure 3.2). We defined the space cost as the distinct number of (dstIP, srcIP) tuples stored ($\sum_d |H_d|$), which is 34 times higher for the naive algorithm compared to the small-space one. Also, the naive algorithm took more than twice as much time to run the small-space one.

For the *third* objective, we tested the small-space algorithm on two datasets (with different values of s_1 and s_2): one with 1.4 billion (destination IP, source IP) tuples, and the other with 20.7 million (destination port, destination IP) tuples - we will refer to these two datasets as “IPPair” and “PortIP” respectively. To test the accuracy of our small-space algorithm, we derived the “ground truth”, i.e., a list of the *actual* heavy-hitters along both the dimensions along with their *exact* frequencies, by employing a four-pass variant of the Misra-Gries algorithm (as discussed in Section 3.1.1).

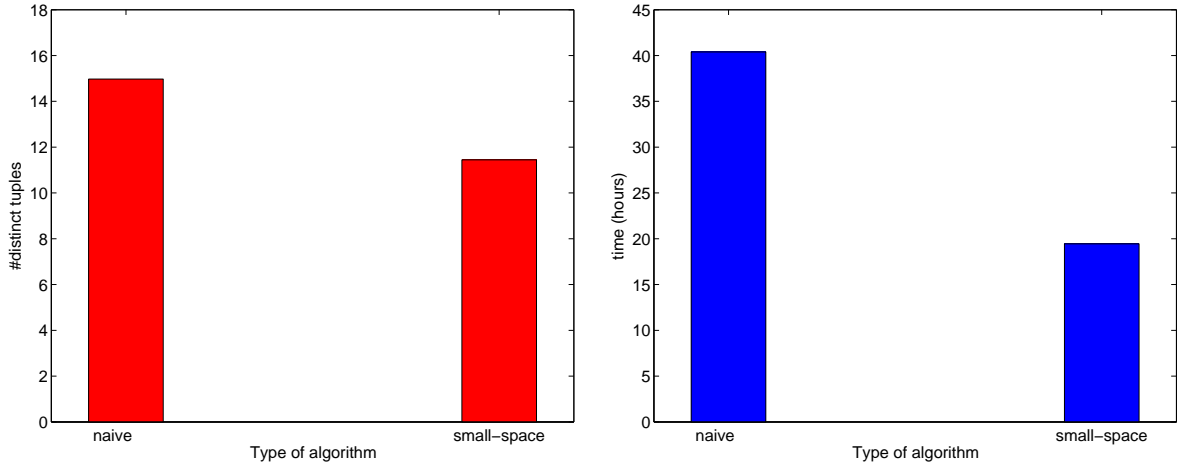


Figure 3.2: Comparison of space (left) and time (right) costs of the naive and the small-space algorithms. The space is the total number of distinct tuples stored, summed over all distinct destination IP addresses. The time is the number of hours to process the 248 million records. Note that the Y-axis for the left graph is logarithmic.

Observations: We define the error statistic in estimating the frequency of a heavy-hitter value d of the primary attribute as $\frac{f_d - \hat{f}_d}{N}$, and in Figures 3.3 and 3.5, for each value of s_1 , we plot the maximum and the average of this error statistic over all the heavy-hitter values of the primary attribute. We observed that both the maximum and the average fell sharply as s_1 increased. Even by using a space budget (s_1) as low as 1000, the maximum error statistic was only 0.09% for “IPPair” and 0.04% for “PortIP”.

The graphs in Figures 3.4 and 3.6 show the results of running our small-space algorithm with different values of s_1 as well as s_2 . We define the error statistic in estimating the frequency of a CHH s (that occurs along with a heavy-hitter primary attribute d) as $\frac{f_{d,s} - \hat{f}_{d,s}}{f_d}$, and for each combination of s_1 and s_2 , we plot the theoretical maximum, the experimental maximum and the average of this error statistic over all CHH attributes. Here also, we observed that both the maximum and the average fall sharply as s_1 increases. However, for a fixed value of s_1 , as we increased the value of s_2 , the maximum did not change at all (for either of “IPPair” and “PortIP”), and the average did not reduce too much - this becomes evident if we compare the readings of the three sub-figures in Figures 3.4 or 3.6, which differ in their values of s_2 , for

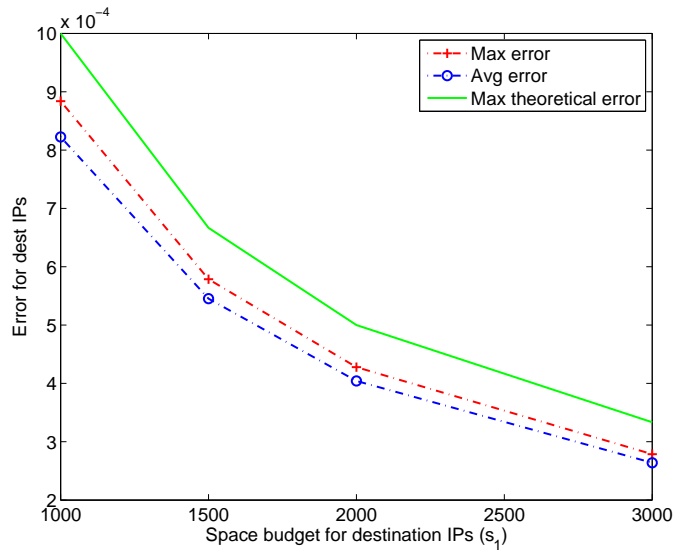


Figure 3.3: Error statistic in estimating the frequencies of the heavy-hitter destination IPs in “IPPair”. The graph shows the theoretical maximum ($\frac{1}{s_1}$), the experimental maximum and the experimental average.

identical values of s_1 . The possible reason is the number of CHHs being very low compared to the number of distinct values of the secondary attribute occurring with a heavy-hitter primary attribute, as we have pointed out in Figure 3.1 for “IPPair”. However, this is good because it implies that in practice, setting s_2 as low as $\frac{1}{\psi}$ should be enough.

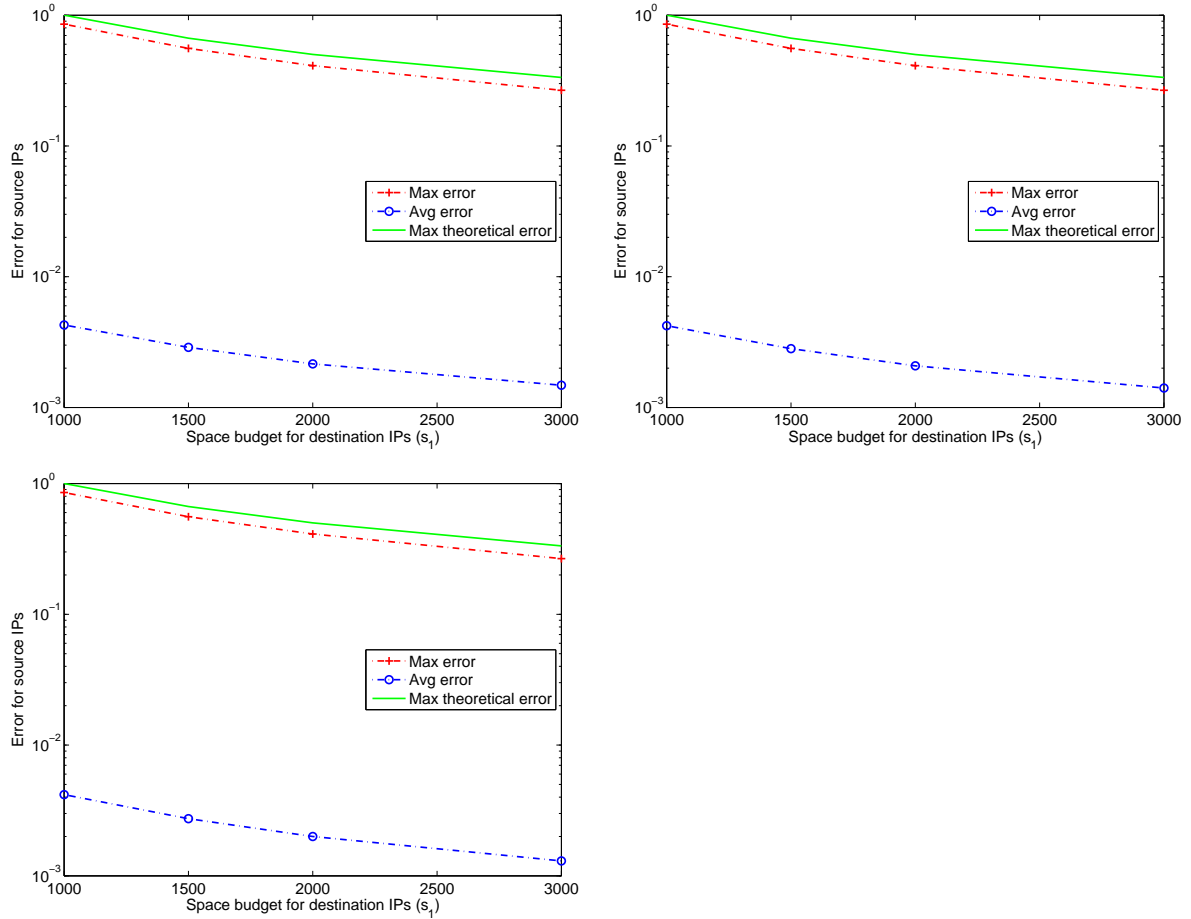


Figure 3.4: Error statistic in estimating the frequencies of the CHH source IPs in “IPPair”, for $s_2 = 1100, 1500$ and 2000 respectively. The graph shows the theoretical maxima $\left(\frac{1}{\phi s_1} + \frac{1}{s_2}\right)$, the experimental maxima and the experimental average.

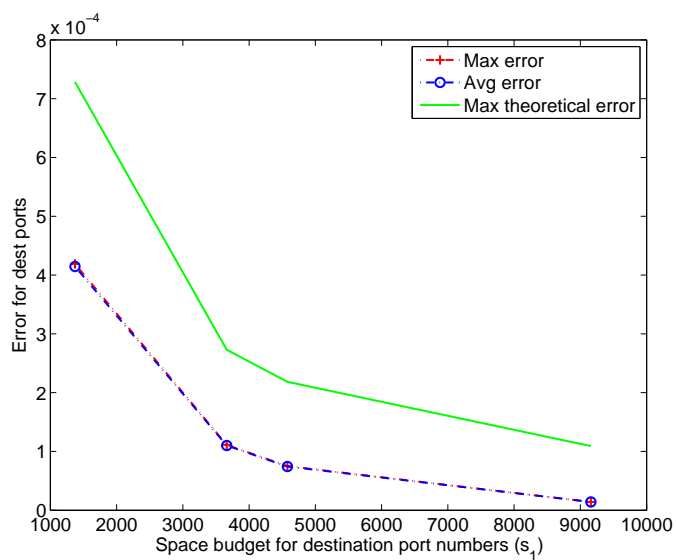


Figure 3.5: Error statistic in estimating the frequencies of the heavy-hitter destination ports from “PortIP”

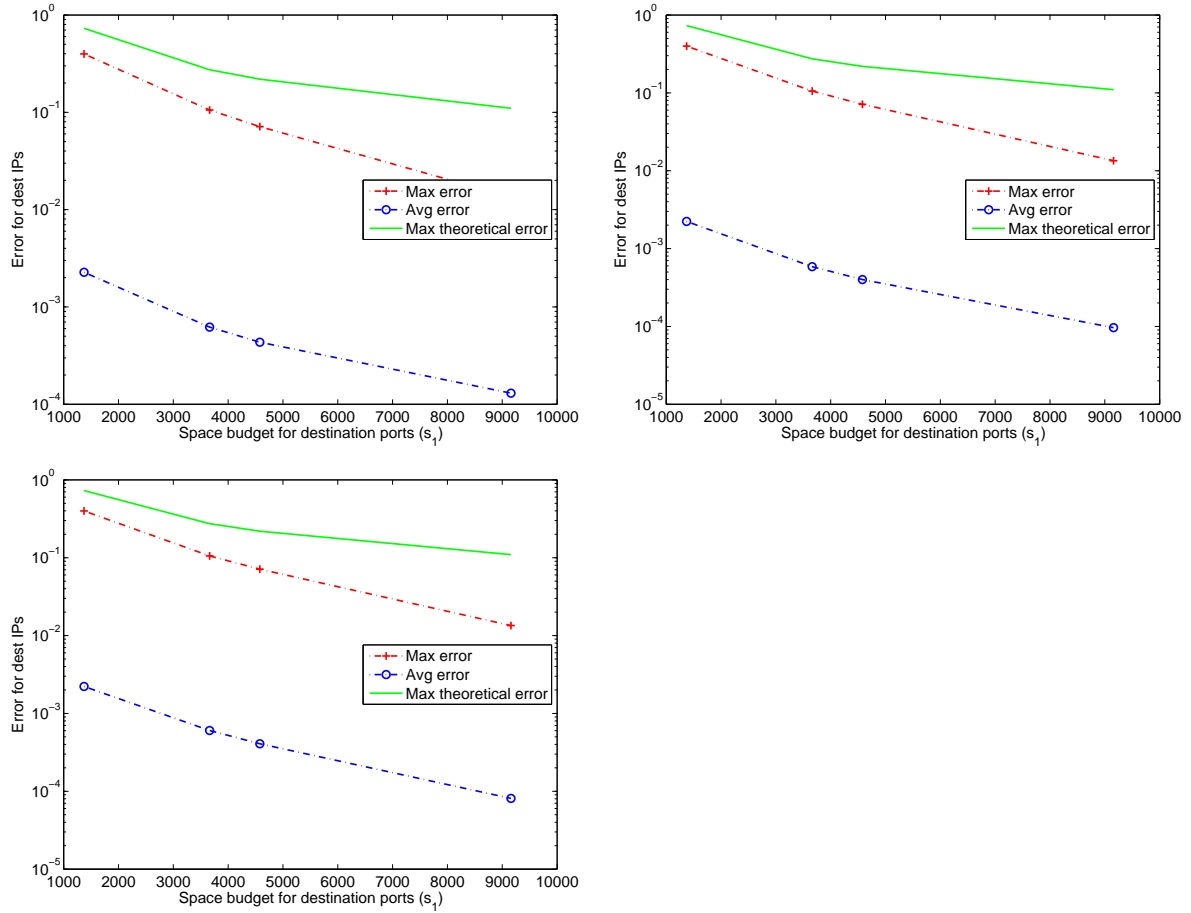


Figure 3.6: Error statistic in estimating the frequencies of the CHH destination IPs in “PortIP”. The three graphs are for $s_2 = 1100$, $s_2 = 1500$ and $s_2 = 2000$ respectively.

3.7 Conclusion and Future Work

For two-dimensional data streams, we presented a small-space approximation algorithm to identify the heavy-hitters along the secondary dimension from the substreams induced by the heavy-hitters along the primary. We theoretically studied the relationship between the maximum errors in the frequency estimates of the heavy-hitters and the space budgets; computed the minimum space requirement along the two dimensions for user-given error bounds; and tested our algorithm to show the space-accuracy tradeoff for both the dimensions.

Identifying the heavy-hitters along any one dimension allows us to split the original stream into several important substreams; and take a closer look at each one to identify the properties of the heavy-hitters. In future, we plan to work on computing other statistics of the heavy-hitters. For example, as we have already discussed in Section 4.5, our experiments with the naive algorithm (on both the datasets) revealed that the number of *distinct* secondary attribute values varied quite significantly across the different (heavy-hitter) values of the primary attribute. For any such data with high variance, estimating the variance in small space [19, 116] is an interesting problem in itself. Moreover, for data with high variance, the simple arithmetic mean is not an ideal central measure, so finding different quantiles, once again in small space, can be another problem worth studying.

CHAPTER 4. Identifying Frequent Items in a Network using Gossip

In this chapter, we present algorithms for identifying frequently occurring items in a large distributed data set. Our algorithms use gossip as the underlying communication mechanism, and do not rely on any central control, or on an underlying network structure, such as a spanning tree. Instead, nodes repeatedly select a random partner and exchange data with the partner. If this process continues for a (short) period of time, the desired results are computed, with probabilistic guarantees on the accuracy. Our algorithm for identifying frequent items is built by layering a novel small space “sketch” of data over a gossip-based data dissemination mechanism. We prove that the algorithm identifies the frequent items with high probability, and provide bounds on the time till convergence. To our knowledge, this is the first work on identifying frequent items using gossip.

4.1 Introduction

We are increasingly faced with data-intensive decentralized systems, such as large scale peer-to-peer networks, server farms with tens of thousands of machines, and large wireless sensor networks. With such large networks comes increasing unpredictability; the networks are constantly changing, due to nodes joining and leaving, or due to node and link failures. Gossip is a type of communication mechanism that is ideally suited for distributed computation on such unstable, large networks. Gossip-based distributed protocols do not assume any underlying structure in the network, such as a spanning tree, and hence, there is no overhead of sub-network formation and maintenance. A gossip protocol proceeds in many “rounds”. In each round, a node contacts a few randomly chosen nodes in the system and exchanges information with them. The randomization inherently provides robustness, and surprisingly, often leads to fast

convergence times. The use of gossip-based protocols for data dissemination and aggregation in distributed systems was first proposed by Demers *et al.* [44].

We consider the problem of identifying *frequent items* in a distributed data set, using gossip. Consider a large peer-to-peer network that is distributing content, such as news or software updates. Suppose that the nodes in the network wish to track the identities of the most frequently accessed items in the network. The relevant data for tracking this aggregate are the frequencies of accesses of different items. However, this data is distributed throughout the network – in fact, even the number of accesses to a single item may not be available at any single point in the network. Our algorithm can be used to track the most frequently accessed items in a low-overhead, decentralized manner, without having to aggregate the frequencies of accesses at any central location. Another application of tracking frequent items is in the detection of a distributed denial of service (DDoS) attack, where many malicious nodes may team up to simultaneously send excessive traffic towards a single victim (typically a web server), so that legitimate clients are denied service. Detecting a DDoS attack is equivalent to finding that the total number of accesses to some server has exceeded a threshold. A distributed frequent items algorithm can help by tracking the most frequently accessed web servers in a distributed manner, and noting if these frequencies are abnormally large. With a gossip-based algorithm this computation can proceed in a totally decentralized manner.

We consider two versions of the problem, one with a relative threshold on the frequency, and the other with an absolute threshold on the frequency. In the relative threshold version, the task is to identify all items whose frequency of occurrence is more than a certain fraction of the total size of the data, where the fraction (the relative threshold) is a user-defined parameter. In the absolute threshold version, the task is to identify all items whose frequency of occurrence is at least an absolute number (the absolute threshold), which is a user-defined parameter. In a distributed dynamic network, these two problems turn out to be rather different from each other.

Our algorithms work without explicitly tabulating the frequencies of different items at any single place in the network. Instead, the distributed data is represented by a small space “sketch” that is propagated and updated via gossip. A sketch is a space-efficient representation

of the input, which is specific to the aggregate being computed, and captures the essence of the data for our purposes. The space taken by the sketch can be tuned as a function of the desired accuracy. A complication with gossip is that since it is an unstructured form of communication, it is possible for the same data item to be inserted into the sketch multiple times as the sketch propagates. Due to this, a technical requirement on the sketch is that it should be able to handle duplicate insertions, i.e. it should be *duplicate-insensitive*. If the gossip proceeds long enough, the sketch can be used to identify all items whose frequency exceeds the user defined threshold. At the same time, items whose popularity is significantly below the threshold will be omitted (again, with high probability).

Contributions. The contributions of this work are as follows:

- We present randomized algorithms for identifying frequent items using gossip, for both the relative and absolute threshold versions of the problem.
- For each algorithm, we present a rigorous analysis of the correctness, time till convergence, and the communication overhead. Our analysis shows that these algorithms converge quickly, and can maintain frequent items in a network with a reasonable communication overhead.
- We present results from our simulations on synthetic data sets. We observed in our simulations that the convergence time and the communication overhead were both much lower than the theoretically guaranteed predictions.

To our knowledge, this is the first work on identifying frequent items in a distributed data set using gossip.

With a gossip protocol, communication is inherently randomized, and a node can never be certain that the results on hand are correct. However, the longer the protocol runs, the closer the results get to the correct answer, and we are able to quantify the time taken till the protocol converges to the correct answer, with high probability. Gossip algorithms are suitable for applications which can tolerate such relaxed consistency guarantees. Examples include a network monitoring application, which is running in the background and is maintaining

statistics about frequently requested data items, or the most frequently observed data in a distributed system. In such an application, an exact answer may not be required, and an approximate answer may suffice.

4.1.1 Related Work

Demers *et al.* [44] were the first to provide a formal treatment of gossip protocols (or “epidemic algorithms” as they called them) for data dissemination. Kempe and Kleinberg [74] analyzed the influence of the underlying gossip mechanism on the design of gossip-based protocols, and explored the limitations of uniform gossip in solving the *nearest resource location problem*. Kempe, Dobra and Gehrke [73] proposed algorithms for computing the sum, average, approximately uniform random sample and quantiles using uniform gossip. Their algorithm for quantiles are based on their algorithm for the sum – they choose a random element in the data, and count the number of elements that are greater and lesser than the chosen element, and recurse on smaller data sets until the quantile is found. Thus their algorithms need many instances of “sum” computations to converge before the quantile is found. A similar approach could potentially be used to find frequent items using gossip. In contrast, our algorithms are not based on repeated computation of the sum, and converge faster.

Much recent work [21, 22, 92] has focused on computing “separable functions” using gossip. A separable function is one that can be expressed as the sum of individual functions of the node inputs. For example, the function “count” is separable, and so is the function “sum”. However, the set of frequent items is not a separable function. Hence, these techniques do not apply to our problem. There is much other work on the computation of basic aggregates, we list a few representative ones here. Kashyap *et al.* [72] proposed algorithms for gossip with flexible tradeoffs between the number of rounds and the number of messages transmitted. Dimakis, Sarwate and Wainwright [45] consider the problem of computing the average over *random geometric graphs* with location-aware nodes, combining uniform gossip with greedy geographic routing. Deb, Medard and Choute [41] used network coding along with uniform gossip to speed up the dissemination of k messages in the network.

The problem of identifying frequent items in data has been extensively studied [89, 87, 71]

in the database, data streams and network monitoring communities (where frequent items are often called “heavy-hitters”). The early work in this is due to Misra and Gries [89], who proposed a deterministic algorithm to identify frequent items in a stream in small space, followed by Manku and Motwani [87], who gave randomized and deterministic algorithms for the same problem. The above were algorithms for a centralized setting.

In a distributed setting, Cao and Wang [26] proposed an algorithm to find the top- k items, where they first made a lower-bound estimate for the k^{th} value, and then used the estimate as a threshold to prune away items which should not qualify as top- k . Zhao *et al.* [119] proposed a sampling-based and a counting-sketch-based scheme to identify globally frequent items. Manjhi *et al.* [86] present an algorithm for finding frequent items on distributed streams, through a tree-based aggregation. Venkataraman *et al.* [110] present an algorithm for identifying “superspreaders” or “heavy distinct hitters” in a network data stream. Keralapura, Cormode and Ramamirtham [75] proposed an algorithm for continuously maintaining the frequent items over a network of nodes. The above algorithms sometimes assume the presence of a central node, or an underlying network structure such as a spanning tree [86, 75], and hence are not applicable where the underlying network does not guarantee reliability or robustness. Haridasan and van Renesse [65] proposed a gossip-based technique that allowed each node in a network to estimate the distribution of values held by other nodes, but their results did not offer any theoretical bounds.

Organization of the Paper. In Section 4.2, we state our system model and give a precise definition of the problem. We first present the algorithm and analysis for the relative threshold version in Section 4.3, and then the absolute threshold version in Section 4.4. In Section 4.5 we discuss the simulation results for both absolute and relative thresholds with an asynchronous time model. In Section 4.6, we discuss the extension of these results to synchronous gossip.

4.2 Model

We consider a distributed system with N nodes numbered from 1 to N . The number of nodes N is not necessarily known to any participating node, and this information is not used

by the algorithms. An “item” is an integer from the set $\{1, 2, \dots, m\}$. An “element” is a single occurrence of an item at any node. Each node i holds a multiset of items M_i , or equivalently, a set of elements. Let \mathcal{N}_i denote the size of M_i . Let $M = \bigcup_{i=1}^N M_i$ denote the set of all elements in the network.

For item $v \in [m]$, the frequency of v is denoted by f_v , and is defined as the number of occurrences of v in M . Note that f_v may not be available locally at any node, in fact determining f_v itself requires a distributed computation. The task is to identify those items v such that f_v is large. Let the total number of elements be defined as $\mathcal{N} = \sum_{i=1}^N \mathcal{N}_i$.

We consider the scenario of *uniform gossip*, which is the most commonly used model of gossip. Whenever a node i is chosen to transmit, it chooses the destination of its message to be a node selected uniformly at random from among all the current nodes in the system. The selection of the transmitting node is done by the distributed scheduler, described later in this section. We assume that the participating nodes execute the algorithms faithfully, and do not maliciously attempt to influence the results of the computation by sending spurious/incorrect messages.

Problem Definition. We consider two variants of the problem, depending on how the thresholds are defined.

- **Relative Threshold.** The user may be interested in identifying items whose relative frequency in the data set exceeds a given threshold. More precisely, given a relative threshold ϕ ($0 < \phi < 1$), approximation error ψ ($0 < \psi < \phi$), an item v is considered to be a frequent item if $f_v \geq \phi\mathcal{N}$, and v is considered an infrequent item if $f_v < (\phi - \psi)\mathcal{N}$. According to this definition, there may be no more than $1/\phi$ frequent items.
- **Absolute Threshold.** The user gives an absolute frequency threshold $k > 1$ and approximation error λ ($\lambda < k$). An item v is considered a frequent item if $f_v \geq k$, and v is an infrequent item if $f_v < k - \lambda$. Note that there may be up to \mathcal{N}/k frequent items according to this definition.

In a centralized setting, when all elements are being observed at the same location, the above formulations of relative and absolute thresholds are equivalent, since the number of elements \mathcal{N} can be computed easily, and any absolute threshold can be converted into a relative threshold, or vice versa. However, in a distributed setting, a threshold for relative frequency cannot be locally converted by a node into a threshold on the absolute frequency, since the user in a large distributed system may not know the number of nodes or the number of elements in the system accurately enough. Thus, we treat these two problems separately. The lack of knowledge of the network size N does not, though, prevent the system from choosing gossip partners uniformly at random. For example, Gkantsidis *et al.* [60] show how random walks can provide a good approximation to uniform sampling for networks where the gap between the first and the second eigenvalues of the transition matrix is constant.

Once the gossip has continued for long enough, the following probabilistic guarantees must hold, whether for absolute or relative thresholds. Let δ be a user-provided bound on the error probability ($0 < \delta < 1$).

- With probability at least $(1 - \delta)$, every node reports every frequent item.
- With probability at least $(1 - \delta)$, no node reports an infrequent item.

Note that we present randomized algorithms, where the probabilistic guarantees hold irrespective of the input.

Time Model. Time is divided into non-overlapping rounds. We consider two types of models, asynchronous and synchronous, depending on whether or not the nodes proceed at the same rate.

- In the *asynchronous* model, in each round, a single source node, chosen uniformly at random out of all N nodes, transmits to another randomly chosen receiver. Thus, in each round in the asynchronous model, there is only one message.
- In the *synchronous* model of communication, in each round, every node in the network sends a message to a receiver chosen uniformly at random from among all nodes. Thus,

in a single round of synchronous communication, N messages are exchanged among the nodes.

In Sections 4.3 and 4.4, we mostly focus on the asynchronous model. We discuss the extension of our results to the synchronous model in Section 4.6.

Performance Metrics. We evaluate the quality of our protocols via the following metrics: the *convergence time*, which is defined as the number of rounds of gossip till convergence, and the *communication complexity*, which is defined as the number of bytes exchanged till convergence.

4.3 Frequent Items with Relative Threshold

Given thresholds ϕ and ψ , where $\psi < \phi$, the goal is to identify all items v such that $f_v \geq \phi N$ and no item u such that $f_u < (\phi - \psi)N$.

We first describe the intuition. Our algorithm is based on random sampling. The idea is that if an item occurs frequently in the original data, it is likely to occur at approximately the same relative frequency in an appropriately sized random sample too. Hence, if we choose those items which have occurred frequently in the random sample, we are likely to choose the frequent items in the input also. To give guaranteed accuracy, we need a large enough sampling probability. However, this sampling probability cannot be decided in advance since the size of the dataset is not known beforehand. Hence, our algorithm works with an adaptive sampling probability, based on the idea of *min-wise independent permutations* [23]. For $i \in [N]$ and $\ell \in [\mathcal{N}_i]$, let the tuple (i, ℓ) denote the ℓ th element within M_i . Thus the tuple (i, ℓ) uniquely identifies an element within M , by first identifying a node id i , and then an element within M_i . Let m_i^ℓ denote the value of element (i, ℓ) . The algorithm assigns each element (i, ℓ) a weight w_i^ℓ , which is a random number in the unit interval $(0, 1)$.

The algorithm maintains a sketch S of $(i, \ell, m_i^\ell, w_i^\ell)$ tuples where (i, ℓ) identifies the element, m_i^ℓ is the value of the element, and w_i^ℓ is the weight. The sketch has no more than t elements, and only the tuples that have the smallest weights are included in S . The intuition is that if an item v has a large relative frequency, then v must occur frequently among the tuples with

the t smallest weights, and hence in the sketch. Maintaining these t elements with the smallest weights through gossip is easy, just as it is easy to maintain the smallest weight element through gossip. If we choose a large enough sketch size t , the likelihood of a frequent item appearing in the sketch a sufficient number of times is very high.

The algorithm for the asynchronous model is described in Figure 1. The threshold t is determined through the analysis to be $O(\frac{1}{\psi^2} \ln(\frac{1}{\delta}))$. There are three parts to this algorithm (and all others that we describe). The first part is the Initialization, where each node initializes its own sketch. The next part is the Gossip, where the nodes exchange sketches with each other according to the communication model. The algorithm only describes what happens during each round of gossip – it is implicit that such computations repeat forever. The third part is the Query, where a query for frequent items is answered using the sketch. The accuracy of the result improves as further rounds of gossip occur.

The sketch at any node can be stored by any data structure that implements an associative array with keys in a sorted order, the key here being w_i^ℓ , and the value being a combination of the other three. In our simulation, we created a Java object of class `Tuple` with $(i, \ell, m_i^\ell, w_i^\ell)$, and each node maintains the sketch as a `TreeSet` collection of these `Tuple` objects. The `TreeSet` Java class allows to store the objects in a user-defined order, so we kept the objects sorted as per the weight w_i^ℓ . The sorting order was defined by a class called `WeightComparator`, which implements the `Comparator` interface of Java. When two sketches (`TreeSet` objects) were merged in our simulation, the `TreeSet` class ensured that when an object is added to the sketch of the local node, the user-defined ordering on weights was preserved. The implementation of the `TreeSet` class ensures that the time taken to add, remove or check the existence of an item to a `TreeSet` collection is logarithmic in the number of objects in the collection. After merging two sketches, we checked whether the total number of objects exceeded the maximum size of the sketch (t), and if it did, we created a `SortedSet` object with the t objects with the lowest weights, and re-initialized the `TreeSet` object (the sketch) with this `SortedSet` object.

Algorithm 11: Gossip algorithm at node i for finding the frequently occurring items with a relative threshold

Input: Data sets M_i ; error probability δ , relative frequency threshold ϕ , approximation error $\psi < \phi$

```

// Initialization
1  $t \leftarrow \frac{128}{\psi^2} \ln(\frac{3}{\delta})$ 
2  $S_i \leftarrow \Phi$ 
3 foreach  $\ell = 1$  to  $N_i$  do
4   | Choose  $w_i^\ell$  as a uniformly distributed random number in  $(0, 1)$ 
5   | Set  $S_i \leftarrow S_i \cup \{(i, \ell, m_i^\ell, w_i^\ell)\}$ 
6 end
// Gossip
7 foreach round of gossip do
8   | if sketch  $S_j$  is received from node  $j$  then
9     |  $S_i \leftarrow S_i \cup S_j$ 
10    | if  $|S_i| > t$  then
11      | retain  $t$  elements of  $S_i$  with the smallest weights
12    | end
13  | end
14  | if node  $i$  is selected to transmit then
15    | select node  $j$  uniformly at random
16    | send  $S_i$  to  $j$ 
17  | end
18 end
// Query
19 when queried for the frequent items
20 foreach  $v \in \{1, \dots, m\}$  do
21   | if at least  $(\phi - \frac{\psi}{2})t$  (nodeID, elementID, value, weight) tuples exist in  $S_i$  with value  $v$ 
22     | then
23       | report  $v$  as a frequent item
24     | end
25 end

```

4.3.1 Analysis

Let \mathcal{W} denote the multi-set of weights $\bigcup_{i=1}^N \bigcup_{\ell=1}^{\mathcal{N}_i} \{w_i^\ell\}$. Clearly, $\mathcal{N} = |\mathcal{W}|$. Let τ denote the t^{th} smallest element in \mathcal{W} . Let M^t be the set of elements $\{(i, \ell)\}$ such that the $w_i^\ell \leq \tau$, where ties are broken arbitrarily. In other words, M^t is the set of t input elements which have been assigned the smallest weights.

The analysis can be divided into two parts. We first show that with high probability, each frequent item occurs with a sufficient frequency in M^t . Similarly, with high probability, the frequency in M^t of each infrequent item is small. As a result, if the sketch at a node equals M^t , then it can identify frequent items with a low probability of a false positive or a false negative. Note that this portion of the analysis is purely local, and has not yet dealt with the distributed algorithm directly.

Next we analyze the distributed gossip process, and prove that with high probability, the set M^t is disseminated to all nodes within $O(N \log N)$ rounds. Combining the analysis of the gossip with the results about false positives and false negatives, we obtain the main result about the correctness of the algorithm, Theorem 4.3.1.

4.3.2 Analysis of M^t

We first show that τ is sharply concentrated around $\frac{t}{\mathcal{N}}$.

Lemma 4.3.1 *If $t = \frac{128}{\psi^2} \ln(\frac{3}{\delta})$, then: (1) $\Pr[\tau < \frac{t}{\mathcal{N}}(1 - \frac{\psi}{4})] < \frac{\delta}{3}$ and (2) $\Pr[\tau > \frac{t}{\mathcal{N}}(1 + \frac{\psi}{4})] < \frac{\delta}{3}$*

Proof: Let X be a random variable equal to the number of elements in \mathcal{W} that are less than $\frac{t}{\mathcal{N}}(1 - \frac{\psi}{4})$. Since the weights are chosen independently of each other, X follows a binomial distribution with \mathcal{N} trials and probability of success in each trial $\frac{t}{\mathcal{N}}(1 - \frac{\psi}{4})$. This gives $E[X] = t(1 - \frac{\psi}{4})$. Using Chernoff bounds, we get

$$\begin{aligned}
 \Pr[\tau < \frac{t}{\mathcal{N}}(1 - \frac{\psi}{4})] &= \Pr[X \geq t] = \Pr\left[X \geq E[X] \left(\frac{1}{1 - \frac{\psi}{4}}\right)\right] \\
 &\leq \Pr\left[X \geq E[X](1 + \frac{\psi}{4})\right] \quad \left[\text{since } \frac{1}{1 - \frac{\psi}{4}} > 1 + \frac{\psi}{4}\right] \\
 &\leq e^{-\frac{E[X]\psi^2}{48}} = e^{-\frac{t(1 - \frac{\psi}{4})\psi^2}{48}}
 \end{aligned} \tag{4.1}$$

Using $t = \frac{128}{\psi^2} \ln\left(\frac{3}{\delta}\right)$,

$$\frac{t(1 - \frac{\psi}{4})\psi^2}{48} = \frac{8}{3} \ln\left(\frac{3}{\delta}\right) \left(1 - \frac{\psi}{4}\right) \geq \ln\left(\frac{3}{\delta}\right) \quad (4.2)$$

Note that $\frac{8}{3}(1 - \frac{\psi}{4}) \geq 1$ since $\psi \leq 1$. Substituting (4.2) in (4.1) yields:

$$\Pr[\tau < \frac{t}{\mathcal{N}}(1 - \frac{\psi}{4})] \leq e^{-\ln(\frac{3}{\delta})} = \frac{\delta}{3}$$

which completes the proof of the first part.

For the second part, let Y be a random variable equal to the number of elements in \mathcal{W} that are less than $\frac{t}{\mathcal{N}}(1 + \frac{\psi}{4})$. Y follows a binomial distribution with \mathcal{N} trials and probability of success in each trial equal to $\frac{t}{\mathcal{N}}(1 + \frac{\psi}{4})$. This gives $E[Y] = t(1 + \frac{\psi}{4})$.

$$\Pr\left[\tau > \frac{t}{\mathcal{N}}\left(1 + \frac{\psi}{4}\right)\right] = \Pr[Y < t] = \Pr\left[Y < E[Y] \left(\frac{1}{1 + \frac{\psi}{4}}\right)\right]$$

Note that $\frac{1}{1 + \frac{\psi}{4}} \leq 1 - \frac{\psi}{8}$ since $(1 + \frac{\psi}{4})(1 - \frac{\psi}{8}) = 1 + \frac{\psi}{8} - \frac{\psi^2}{32} \geq 1$. This yields

$$\begin{aligned} \Pr\left[Y < E[Y] \left(\frac{1}{1 + \frac{\psi}{4}}\right)\right] &\leq \Pr\left[Y < E[Y] \left(1 - \frac{\psi}{8}\right)\right] \\ &\leq e^{-\frac{E[Y]\psi^2}{64}(\frac{1}{2})} \text{ [Chernoff bound]} \end{aligned}$$

Substituting $t = \frac{128}{\psi^2} \ln\left(\frac{3}{\delta}\right)$, we get:

$$\frac{E[Y]\psi^2}{64} = t(1 + \frac{\psi}{4}) \frac{\psi^2}{64} = \frac{128}{64} \ln\left(\frac{3}{\delta}\right) \left(1 + \frac{\psi}{4}\right) \geq 2 \ln\left(\frac{3}{\delta}\right)$$

Thus,

$$\Pr\left[\tau > \frac{t}{\mathcal{N}}\left(1 + \frac{\psi}{4}\right)\right] \leq e^{-\ln(\frac{3}{\delta})} = \frac{\delta}{3}$$

■

We next prove results about the false negatives and false positives. In order to do so, we need the following corollaries of the Chernoff bound. Let X be any binomial random variable, i.e. $X = \sum_{i=1}^n X_i$ where the X_i are independent 0-1 random variables. The common form of the Chernoff bound expresses the tail probabilities of X as a function of the expectation

$\mu = E[X]$. In our cases, $E[X]$ is not known exactly, but a range $[\mu_L, \mu_H]$ is known such that $\mu_L \leq \mu \leq \mu_H$. In such a case, the following inequalities are useful.

Lemma 4.3.2 For any $0 < \delta \leq 1$,

$$\Pr[X \geq (1 + \delta)\mu_H] \leq \exp\left(\frac{-\mu_H\delta^2}{3}\right)$$

Proof: Let $M_X(\gamma)$ be the moment generating function of X .

$$\begin{aligned} \Pr[X \geq (1 + \delta)\mu_H] &= \Pr[e^{\gamma X} \geq e^{\gamma(1+\delta)\mu_H}] \text{ for any } \gamma > 0 \\ &\leq \frac{E[e^{\gamma X}]}{e^{\gamma(1+\delta)\mu_H}} \text{ [by Markov inequality]} \\ &= \frac{M_X(\gamma)}{e^{\gamma(1+\delta)\mu_H}} \\ &\leq \frac{e^{(e^\gamma-1)\mu}}{e^{\gamma(1+\delta)\mu_H}} \text{ [since } M_X(\gamma) \leq e^{(e^\gamma-1)\mu} \text{, as proved in [90], page 64]} \\ &\leq \frac{e^{(e^\gamma-1)\mu_H}}{e^{\gamma(1+\delta)\mu_H}} \text{ [since } \mu_H \geq \mu \text{, and } \gamma > 0 \Rightarrow (e^\gamma - 1) > 0 \Rightarrow e^{(e^\gamma-1)} > 1] \end{aligned}$$

Substituting $\gamma = \ln(1 + \delta) > 0$, we get (for any $\delta > 0$),

$$\Pr[X \geq (1 + \delta)\mu_H] \leq \left(\frac{e^\delta}{(1 + \delta)^{(1+\delta)}}\right)^{\mu_H} \quad (4.3)$$

It is proved in [90], page 65, that for any $0 < \delta \leq 1$,

$$\frac{e^\delta}{(1 + \delta)^{(1+\delta)}} \leq \exp\left(\frac{-\delta^2}{3}\right)$$

Combining this with inequality 4.3, the claim follows. ■

Lemma 4.3.3 For any $0 < \delta < 1$,

$$\Pr[X \leq (1 - \delta)\mu_L] \leq \exp\left(\frac{-\mu_L\delta^2}{2}\right)$$

Proof:

$$\begin{aligned} \Pr[X \leq (1 - \delta)\mu_L] &= \Pr[e^{\gamma X} \geq e^{\gamma(1-\delta)\mu_L}] \text{ for any } \gamma < 0 \\ &\leq \frac{E[e^{\gamma X}]}{e^{\gamma(1-\delta)\mu_L}} \text{ [by Markov inequality]} \\ &= \frac{M_X(\gamma)}{e^{\gamma(1-\delta)\mu_L}} \text{ [} M_X(\gamma) \text{ is the moment generating function of } X] \\ &\leq \frac{e^{(e^\gamma-1)\mu}}{e^{\gamma(1-\delta)\mu_L}} \text{ [since } M_X(\gamma) \leq e^{(e^\gamma-1)\mu}] \\ &\leq \frac{e^{(e^\gamma-1)\mu_L}}{e^{\gamma(1-\delta)\mu_L}} \text{ [since } \mu_L \leq \mu \text{, and } \gamma < 0 \Rightarrow (e^\gamma - 1) < 0 \Rightarrow 0 < e^{(e^\gamma-1)} < 1] \end{aligned}$$

Substituting $\gamma = \ln(1 - \delta) < 0$, we get, for any $0 < \delta < 1$,

$$\Pr[X \leq (1 - \delta)\mu_L] \leq \left(\frac{e^{-\delta}}{(1 - \delta)^{(1-\delta)}} \right)^{\mu_L} \quad (4.4)$$

It is proved in [90], page 66, that for any $0 \leq \delta < 1$,

$$\frac{e^{-\delta}}{(1 - \delta)^{(1-\delta)}} \leq \exp\left(\frac{\delta^2}{2}\right)$$

Combining this with inequality 4.4, the claim follows. ■

The following lemmas provide upper bounds on the probabilities of finding a false negative and a false positive respectively, in a centralized setting.

Lemma 4.3.4 *If v is a frequent item, i.e. $f_v \geq \phi\mathcal{N}$, then with probability at least $1 - \delta$, v occurs at least $(\phi - \frac{\psi}{2})t$ times in M^t .*

Proof: Let E_1 be the event that v occurs at least $(\phi - \frac{\psi}{2})t$ times in M^t . Let E_2 be the event $\tau \geq \frac{t}{\mathcal{N}}(1 - \frac{\psi}{4})$. Let Z be a random variable indicating the number of copies of v with weight $\frac{t}{\mathcal{N}}(1 - \frac{\psi}{4})$ or smaller. Let E_3 be the event that $Z \geq (\phi - \frac{\psi}{2})t$. We observe that if E_2 and E_3 are true, then E_1 is also true. From Lemma 4.3.1, we know that $\Pr[\bar{E}_2] < \frac{\delta}{3}$. Using these:

$$\begin{aligned} \Pr[E_1] &\geq \Pr[E_2 \wedge E_3] = 1 - \Pr[\bar{E}_2 \vee \bar{E}_3] \\ &\geq 1 - \Pr[\bar{E}_2] - \Pr[\bar{E}_3] \geq 1 - \frac{\delta}{3} - \Pr\left[Z < \left(\phi - \frac{\psi}{2}\right)t\right] \end{aligned} \quad (4.5)$$

To estimate $\Pr\left[Z < \left(\phi - \frac{\psi}{2}\right)t\right]$, note that Z follows a binomial distribution with $\phi\mathcal{N}$ or more trials and a probability of success of $\frac{t}{\mathcal{N}}(1 - \frac{\psi}{4})$. This makes

$$E[Z] \geq (\phi\mathcal{N}) \left(\frac{t}{\mathcal{N}} \left(1 - \frac{\psi}{4}\right) \right) = \phi t \left(1 - \frac{\psi}{4}\right) \geq \left(\phi - \frac{\psi}{4}\right)t$$

Applying Lemma 4.3.3 with $\mu_L = (\phi - \frac{\psi}{4})t$ (note that $\mu = E[Z] \geq \mu_L$), we get

$$\Pr\left[Z < \left(\phi - \frac{\psi}{2}\right)t\right] = \Pr\left[Z < \left(\phi - \frac{\psi}{4}\right)t \left(1 - \frac{\psi}{4\phi - \psi}\right)\right]$$

$$\begin{aligned}
&\leq \exp\left(\frac{-(\phi - \frac{\psi}{4})t(\frac{\psi}{4\phi - \psi})^2}{2}\right) \\
&\leq \exp\left(\frac{-4\ln(\frac{3}{\delta})}{\phi - \frac{\psi}{4}}\right) \quad [\text{Substituting } t = \frac{128}{\psi^2} \ln(\frac{3}{\delta})] \\
&< \exp\left(-4\ln(\frac{3}{\delta})\right) \quad [\text{Since } 0 < \phi - \frac{\psi}{4} < 1] \\
&= \left(\frac{\delta}{3}\right)^4 \tag{4.6}
\end{aligned}$$

Substituting (4.6) in (4.5):

$$\Pr[E_1] \geq 1 - \frac{\delta}{3} - \left(\frac{\delta}{3}\right)^4 \geq 1 - \delta$$

■

Lemma 4.3.5 *If u is an infrequent item, i.e. $f_u < (\phi - \psi)\mathcal{N}$, then, with probability at least $1 - \delta$, u occurs less than $(\phi - \frac{\psi}{2})t$ times in M^t .*

Proof: Let Y denote the number of copies of u with weight $\leq \frac{t}{\mathcal{N}}(1 + \frac{\psi}{4})$. Let E denote the event that u occurs less than $(\phi - \frac{\psi}{2})t$ times in M^t . As in the proof of Lemma 4.3.4, using Lemma 4.3.1, we get:

$$\begin{aligned}
\Pr[E] &\geq \Pr\left[\left(\tau \leq \frac{t}{\mathcal{N}}\left(1 + \frac{\psi}{4}\right)\right) \wedge \left(Y \leq \left(\phi - \frac{\psi}{2}\right)t\right)\right] \\
&\geq 1 - \Pr\left[\tau > \frac{t}{\mathcal{N}}\left(1 + \frac{\psi}{4}\right)\right] - \Pr\left[Y > \left(\phi - \frac{\psi}{2}\right)t\right] \\
&\geq 1 - \frac{\delta}{3} - \Pr\left[Y > \left(\phi - \frac{\psi}{2}\right)t\right] \tag{4.7}
\end{aligned}$$

To estimate $\Pr\left[Y > \left(\phi - \frac{\psi}{2}\right)t\right]$, note that Y follows a binomial distribution with $(\phi - \psi)\mathcal{N}$ or less trials and a probability of success of $\frac{t}{\mathcal{N}}(1 + \frac{\psi}{4})$. Thus,

$$E[Y] \leq (\phi - \psi)\mathcal{N} \left(\frac{t}{\mathcal{N}}\left(1 + \frac{\psi}{4}\right)\right) \leq t \left[\phi - \frac{3\psi}{4} - \frac{\psi^2}{4}\right] \leq t \left[\phi - \frac{3\psi}{4}\right]$$

Applying Lemma 4.3.2 with $\mu_H = (\phi - \frac{3\psi}{4})t$ (note that $\mu = E[Z] \leq \mu_H$), we get

$$\Pr\left[Y > \left(\phi - \frac{\psi}{2}\right)t\right] = \Pr\left[Y > \left(\phi - \frac{3\psi}{4}\right)t \left(1 + \frac{\psi}{4\phi - 3\psi}\right)\right]$$

$$\begin{aligned}
&\leq \exp\left(\frac{-(\phi - \frac{3\psi}{4})t(\frac{\psi}{4\phi - 3\psi})^2}{3}\right) \\
&\leq \exp\left(\frac{-8 \ln(\frac{3}{\delta})}{3(\phi - \frac{3\psi}{4})}\right) \quad [\text{Substituting } t = \frac{128}{\psi^2} \ln(\frac{3}{\delta})] \\
&< \exp\left(-\frac{8}{3} \ln\left(\frac{3}{\delta}\right)\right) \quad [\text{Since } 0 < \phi - \frac{3\psi}{4} < 1] \\
&= \left(\frac{\delta}{3}\right)^{\frac{8}{3}} \tag{4.8}
\end{aligned}$$

Substituting (4.8) in (4.7), we get

$$\Pr[E] \geq 1 - \frac{\delta}{3} - \left(\frac{\delta}{3}\right)^{\frac{8}{3}} \geq 1 - \delta$$

■

4.3.3 Analysis of Gossip.

Consider an item θ that is disseminated through gossip in the asynchronous model. At the start, θ is with one node, and in subsequent rounds it is disseminated to the other nodes. Let T^N be the number of rounds till θ is disseminated to all the N nodes.

Lemma 4.3.6 $E[T^N] = 2N \ln N + O(N)$.

Proof: Let ξ_i be the set of nodes that have θ after i rounds. Thus ξ_0 has only one node (the one that sampled θ during the initialization step). For $j = 1 \dots N - 1$, let random variable X_j be the number of rounds required to increase the number of nodes that have θ from j to $j + 1$.

$$T^N = \sum_{j=1}^{N-1} X_j$$

For $i \geq 1$, in round i , a new node receives θ if a gossip message is transmitted from node α to node β where $\alpha \in \xi_{i-1}$ and $\beta \notin \xi_{i-1}$. Thus X_j is a geometric random variable, i.e. the number of trials till the first “success”, where a success is defined as a message from node $\alpha \in \xi_{i-1}$ to a node $\beta \notin \xi_{i-1}$. The probability of a success is thus $\left(\frac{j}{N}\right) \left(1 - \frac{j}{N}\right) = \frac{j(N-j)}{N^2}$. A geometric random with probability of success p has expectation $1/p$, so we get $E[X_j] = \frac{N^2}{j(N-j)}$.

Using linearity of expectation, we get:

$$\begin{aligned}
E[T^N] &= \sum_{j=1}^{N-1} E[X_j] = \sum_{j=1}^{N-1} \frac{N^2}{j(N-j)} = N \sum_{j=1}^{N-1} \left(\frac{1}{j} + \frac{1}{N-j} \right) \\
&= 2N \sum_{j=1}^{N-1} \frac{1}{j} = 2NH_{N-1} = 2N \ln N + O(N)
\end{aligned}$$

where H_k denotes the k th Harmonic number. ■

Our proof for high-probability bounds on T^N uses a result about the *coupon collector* problem. Suppose there are coupons of Λ distinct types, labeled $\{1, 2, \dots, \Lambda\}$, and one has to draw coupons at random (with replacement) until at least one coupon of each type has been collected. Initially, it is very easy to select a type not yet chosen, but as more and more types get chosen, it becomes increasingly difficult to get a coupon of a type not yet chosen. We present a high probability bound on the number of trials to collect all Λ coupons.

Lemma 4.3.7 *Let the random variable \mathcal{C}_Λ denote the number of trials to collect at least one coupon of each of Λ types. Then,*

$$\Pr[\mathcal{C}_\Lambda > 3\Lambda \ln \Lambda] \leq \frac{1}{\Lambda^2}$$

Proof: Let E_i denote the event that the coupon with label $i \in \{1, 2, \dots, \Lambda\}$ did not get drawn at all in $3\Lambda \ln \Lambda$ trials. Then,

$$\begin{aligned}
\Pr[\mathcal{C}_\Lambda > 3\Lambda \ln \Lambda] &= \Pr\left(\bigcup_{i=1}^{\Lambda} E_i\right) \leq \sum_{i=1}^{\Lambda} \Pr(E_i) \text{ (union bound)} \\
&= \sum_{i=1}^{\Lambda} \left(1 - \frac{1}{\Lambda}\right)^{3\Lambda \ln \Lambda} \leq \sum_{i=1}^{\Lambda} \left(e^{-\frac{1}{\Lambda}}\right)^{3\Lambda \ln \Lambda} \leq \sum_{i=1}^{\Lambda} \frac{1}{\Lambda^3} = \frac{1}{\Lambda^2}
\end{aligned}$$
■

Lemma 4.3.8

$$\Pr[T^N > 12N \ln 2N] \leq \frac{1}{2N^2}$$

Proof: The dissemination of θ can be divided into two phases. The first phase starts with the first transmission and continues until the object has reached $\frac{N}{2}$ distinct nodes. The second

phase starts once it has reached $\frac{N}{2}$ nodes and continues until it reaches N nodes. Note that, in the first phase, it is more unlikely to find a source node that has θ and it is easy to find a destination that does not have θ . Once θ has reached $\frac{N}{2}$ nodes, the situation reverses. Let T_1 and T_2 be the number of rounds taken by the two phases, respectively.

More formally, let X_j be defined as in the proof of Lemma 4.3.6. Let $T_1 = \sum_{j=1}^{N/2} X_j$, and $T_2 = \sum_{j=\frac{N}{2}+1}^{N-1} X_j$. Clearly, we have $T^N = T_1 + T_2$.

To bound T^N , we note that if $T^N > 12N \ln 2N$, then at least one of T_1 or T_2 should be greater than $6N \ln 2N$. In Lemma 4.3.9, we show that T_1 and T_2 are bounded by $6N \ln 2N$, with high probability. Thus,

$$\begin{aligned} \Pr[T^N > 12N \ln 2N] &\leq \Pr[(T_1 > 6N \ln 2N) \cup (T_2 > 6N \ln 2N)] \\ &\leq \Pr[T_1 > 6N \ln 2N] + \Pr[T_2 > 6N \ln 2N] \text{ (union bound)} \\ &\leq \frac{1}{4N^2} + \frac{1}{4N^2} \text{ (Lemma 4.3.9)} \\ &= \frac{1}{2N^2} \end{aligned}$$

■

Lemma 4.3.9

$$\begin{aligned} \Pr[T_1 > 6N \ln 2N] &\leq \frac{1}{4N^2} \\ \Pr[T_2 > 6N \ln 2N] &\leq \frac{1}{4N^2} \end{aligned}$$

Proof: For $0 < p \leq 1$, let $G(p)$ denote a geometric random variable with parameter p . From the proof of Lemma 4.3.6, we know that $X_j = G\left(\frac{j(N-j)}{N^2}\right)$. Thus, T_1 is the sum of independent geometric random variables.

$$T_1 = \sum_{j=1}^{N/2} G\left(\frac{j(N-j)}{N^2}\right)$$

Consider the random variable \mathcal{C}_{2N} , the number of trials needed to collect $2N$ coupons.

$$\mathcal{C}_{2N} = \sum_{j=1}^{2N} G\left(\frac{2N-j+1}{2N}\right)$$

Let random variable \mathcal{C}' consist of the last few terms of \mathcal{C}_{2N} :

$$\mathcal{C}' = \sum_{j=1}^{N/2} G\left(\frac{j}{2N}\right)$$

Note that for $j = 1 \dots N/2$, we have $(N - j) \geq \frac{N}{2}$, and hence $\frac{j(N-j)}{N^2} \geq \frac{j}{2N}$. Thus we can write \mathcal{C}' and T_1 as follows:

$$\begin{aligned} \mathcal{C}' &= \sum_{j=1}^{N/2} G(y_j) \\ T_1 &= \sum_{j=1}^{N/2} G(x_j) \end{aligned}$$

such that $x_j \geq y_j$, for all $j = 1, \dots, \frac{N}{2}$. We also know that if $0 < y \leq x \leq 1$, then for any $\gamma > 0$, $\Pr[G(y) \geq \gamma] \geq \Pr[G(x) \geq \gamma]$. From Lemma 4.3.10, we have that \mathcal{C}' stochastically dominates T_1 , i.e. for each $\gamma \geq 0$, $\Pr[\mathcal{C}' \geq \gamma] \geq \Pr[T_1 \geq \gamma]$. Since $\mathcal{C}_{2N} \geq \mathcal{C}'$, we have:

$$\Pr[T_1 > 6N \ln 2N] \leq \Pr[\mathcal{C}' > 6N \ln 2N] \leq \Pr[\mathcal{C}_{2N} > 6N \ln 2N] \leq \frac{1}{4N^2}$$

where we have used Lemma 4.3.7. The proof for T_2 follows similarly. ■

Lemma 4.3.10 *Suppose Y and Z are random variables defined as follows. $Y = \sum_{i=1}^k Y_i$, and $Z = \sum_{i=1}^k Z_i$, where the Y_i s are mutually independent, the Z_i s are mutually independent, and for every $i = 1 \dots k$, Y_i stochastically dominates Z_i , i.e. for every $\gamma \geq 0$, $\Pr[Y_i \geq \gamma] \geq \Pr[Z_i \geq \gamma]$. Then, Y stochastically dominates Z , i.e. for each $\gamma \geq 0$, $\Pr[Y \geq \gamma] \geq \Pr[Z \geq \gamma]$.*

Proof: For $i = 1 \dots k$, let f_i and g_i be the cumulative distribution functions of Y_i and Z_i respectively.

$$f_i(\gamma) = \Pr[Y_i \leq \gamma]$$

$$g_i(\gamma) = \Pr[Z_i \leq \gamma]$$

Consider any $i \in \{1, 2, \dots, k\}$. We know that for each $\gamma \geq 0$, $f_i(\gamma) = 1 - \Pr[Y_i > \gamma] \leq 1 - \Pr[Z_i > \gamma] = g_i(\gamma)$. Thus, $f_i(\gamma) \leq g_i(\gamma)$.

We can view Y_i and Z_i as random variables in the same sample space as follows. For $i = 1 \dots k$, let U_i be a number chosen uniformly at random from $(0, 1)$. Let random variables $Y'_i = f_i^{-1}(U_i)$, and $Z'_i = g_i^{-1}(U_i)$. It is easy to see that for every outcome for U_i , $f_i^{-1}(U_i) \geq g_i^{-1}(U_i)$. Thus random variables Y'_i and Z'_i satisfy $Y'_i \geq Z'_i$. If the outcomes $U_i, i = 1, \dots, k$ are all independent, then the Y'_i 's are mutually independent, and the Z'_i 's are mutually independent. We observe that for every $\gamma \geq 0$,

$$\Pr[Y'_i \leq \gamma] = \Pr[f_i^{-1}(U_i) \leq \gamma] = \Pr[U_i \leq f_i(\gamma)] = f_i(\gamma) = \Pr[Y_i \leq \gamma]$$

Hence, Y'_i and Y_i have identical distributions. Similarly, Z'_i and Z_i have identical distributions. Now, consider Y' and Z' defined as follows:

$$Y' = \sum_{i=1}^k Y'_i$$

$$Z' = \sum_{i=1}^k Z'_i$$

Since the Y'_i 's are mutually independent, Y' has the same distribution as Y , and similarly Z' has the same distribution as Z . Further, for each outcome in the above sample space, $Y' \geq Z'$. This implies that for each $\gamma \geq 0$, $\Pr[Y \geq \gamma] \geq \Pr[Z \geq \gamma]$. ■

We now present a bound on the dissemination time of the smallest weights. Let \mathcal{T}^t denote the time taken for all items in M^t to be disseminated to all nodes.

Lemma 4.3.11 $\Pr[\mathcal{T}^t > 12N \ln 2N] \leq \frac{1}{2^N}$.

Proof: For $i = 1 \dots N$, let $M_i^t = M_i \cap M^t$ i.e. the set of all elements at node i which have been assigned weights among the smallest t weights. Note that in algorithm 11, all elements in M_i^t are transmitted together, i.e., in each round, either all elements in M_i^t are transmitted, or none of them are; thus, the upper bound on T^N also applies to the dissemination time of M_i^t . Let E_i denote the event that M_i^t is not disseminated to all nodes in $12N \ln 2N$ rounds. From Lemma 4.3.8, we have $\Pr[E_i] \leq \frac{1}{2^{N^2}}$.

$$\begin{aligned}
\Pr[\mathcal{T}^t > 12N \ln 2N] &= \Pr\left[\bigcup_{i=1}^N E_i\right] \\
&\leq \sum_{i=1}^N \Pr[E_i] \text{ (union bound)} \\
&\leq N \cdot \frac{1}{2N^2} = \frac{1}{2N}
\end{aligned}$$

■

We now present the main theorem on the correctness of algorithm 11.

Theorem 4.3.1 *Suppose algorithm 11 is run for $12N \ln 2N$ rounds. Then, with probability at least $1 - \delta$, an item of frequency ϕN or more in M will be identified as a frequent item at every node. Similarly, with probability at least $1 - \delta$, an item with frequency less than $(\phi - \psi)N$ will not be identified as a frequent item at any node.*

Proof: From Lemma 4.3.11, we have that all elements in M^t are disseminated to every node in the network after $12N \ln 2N$ rounds. The theorem follows from Lemmas 4.3.5 and 4.3.4. ■

Since the size of the sketch at any time during gossip is at most $t = \frac{128}{\psi^2} \ln(\frac{3}{\delta})$, the number of bytes exchanged in each round is $O(\frac{1}{\psi^2} \ln(\frac{1}{\delta}))$. Hence, we get the following result on the communication complexity, using Lemma 4.3.11.

Theorem 4.3.2 *The number of bytes exchanged by algorithm 11 till the frequent items are identified is at most $O(\frac{1}{\psi^2} \ln(\frac{1}{\delta})N \ln N)$, with probability $1 - O(\frac{1}{N})$.*

4.4 Frequent Items with an Absolute Threshold

We now present an algorithm in the asynchronous model for identifying items whose frequency is greater than a user-specified absolute threshold k . As in Section 4.3, let $M = \bigcup_{i=1}^N M_i$ denote the multi-set of all input values. The goal is to output all items v such that $f_v \geq k$ without outputting any item v such that $f_v < k - \lambda$.

The intuition is as follows. Similar to the algorithm for relative threshold, this algorithm is based on random sampling. Unlike the algorithm for relative threshold, the sampling probability can be statically decided by the nodes, based on k and λ . The elements of M are sampled in a distributed manner, and the sampled elements are disseminated using gossip. Intuitively, suppose we sample each element from M into a set S with probability $1/k$. For a frequent item v with $f_v \geq k$, we (roughly) expect one or more copies of v to be present in S . Similarly, for an infrequent item u with $f_u < k - \lambda$, we expect that no copy of u will be included in S . However, some infrequent items may get “lucky” and may be included in S and similarly, some frequent items may not make it to S . The probabilities of these events depend on the sample size.

To refine this sampling scheme, we sample with a probability that is slightly larger than $1/k$, say c/k for some parameter c . Finally, we select those items that occur at least r times within S , for some parameter $r < c$; the value of r will be determined by the analysis. The smaller the value of λ , the greater should be the sampling probability, since we need to make a more precise distinction between the frequencies of frequent and infrequent items. In the actual algorithm, we use a sampling probability of $\frac{12k}{\lambda^2} \ln \frac{2}{\delta}$ – note that this is $\Omega(\frac{1}{k})$ since $\lambda < k$ and hence $\frac{k}{\lambda^2} > \frac{1}{k}$.

The algorithm for the absolute threshold is shown as Algorithm 12. Through our analysis, we give a bound on the number of rounds after which frequent items are likely to be found at all nodes.

4.4.1 Analysis

We now analyze the correctness and the time complexity of algorithm 12. The plan is as follows. Let M^s be the set of elements in M that are sampled by the nodes, and hence get disseminated through gossip. We first show in Lemma 4.4.1 that within a small number of rounds, all elements in M^s are disseminated to all nodes. We then show in Lemma 4.4.2 that for each frequent item, M^s contains sufficient copies of the item (with high probability), thus showing that the probability of a false negative is small. Then, we show in Lemma 4.4.3 that for each infrequent item, M^s does not contain enough copies of the item to be identified as a frequent item at any node (with high probability), showing that the probability of a false

Algorithm 12: Gossip algorithm at node i for finding the frequently occurring items with an absolute threshold k

Input: Data sets M_i ; error probability δ , frequency threshold k , approximation error λ

```

// Initialization
1  $S_i \leftarrow \Phi$ 
2 foreach  $\ell = 1$  to  $N_i$  do
3   | Choose  $\rho$  as a uniformly distributed random number in  $(0, 1)$ 
4   | if  $\rho < \frac{12k}{\lambda^2} \ln \frac{2}{\delta}$  then
5   |   |  $S_i \leftarrow S_i \cup \{(i, \ell, m_i^\ell)\}$ 
6   |   end
7 end
// Gossip
8 foreach round of gossip do
9   | if sketch  $S_j$  is received from node  $j$  then
10  |   |  $S_i \leftarrow S_i \cup S_j$ 
11  |   end
12  | if node  $i$  is selected to transmit then
13  |   | select node  $j$  uniformly at random from  $\{1, \dots, N\}$ 
14  |   | send  $S_i$  to  $j$ 
15  |   end
16 end
// Query
17 when queried for the frequent items
18 foreach  $v \in \{1, \dots, m\}$  do
19  | if  $v$  occurs more than  $r = \frac{12k^2}{\lambda^2} (1 - \frac{\lambda}{2k}) \ln \frac{2}{\delta}$  times in  $S_i$  then
20  |   | report  $v$  as a frequent item
21  |   end
22 end

```

positive is small. Let \mathcal{T}^s denote the time taken for all items in M^s to be disseminated to all nodes.

Lemma 4.4.1

$$\Pr[\mathcal{T}^s > 12N \ln 2N] \leq \frac{1}{2N}$$

Proof: For a single element $\theta \in M^s$ that gets disseminated through gossip, the results of Lemmas 4.3.6 and 4.3.8 from the analysis for relative threshold hold, because the underlying gossip mechanism is same for both the algorithms. For $i = 1 \dots N$, let $M_i^s = M_i \cap M^s$, i.e., the set of all elements at node i which were sampled by node i , and hence included in the sketch at node i when it was initialized. Note that in algorithm 12, all elements in M_i^s are transmitted together, i.e., in each round, either all the elements in M_i^s are transmitted, or none of them are. Thus, the upper bound on T^N from Lemma 4.3.8 also applies to the dissemination time of M_i^s . Let E_i denote the event that M_i^s is not disseminated to all nodes in $12N \ln 2N$ rounds. From Lemma 4.3.8, we have $\Pr[E_i] \leq \frac{1}{2N^2}$.

$$\begin{aligned} \Pr[\mathcal{T}^s > 12N \ln 2N] &= \Pr\left[\bigcup_{i=1}^N E_i\right] \leq \sum_{i=1}^N \Pr[E_i] \\ &\leq N \cdot \frac{1}{2N^2} = \frac{1}{2N} \end{aligned}$$

■

Lemma 4.4.2 False Negative. *If v is an item with $f_v \geq k$, then with probability at least $1 - \delta$, v is returned as a frequent item by every node after $12N \ln 2N$ rounds.*

Proof: Let $r = \frac{12k^2}{\lambda^2} \left(1 - \frac{\lambda}{2k}\right) \ln \frac{2}{\delta}$. If v is such that $f_v \geq k$, then v is not reported by a node in the following two situations.

- Less than r copies of v are present in M^s .
- r or more copies of v were sampled into M^s during the initialization, but some copies did not make it to all nodes during the gossip.

Let E_1 denote the event that less than r copies of v are present in M^s . Let E_2 denote the event that after $12N \ln 2N$ rounds, all of M^s was not disseminated to all the nodes in the network. Let E denote the event that there was some node that did not report v as a frequent item.

$$\Pr[E] \leq \Pr[E_1 \cup E_2] \leq \Pr[E_1] + \Pr[E_2] \quad (4.9)$$

Consider some k copies of v in the input. Let X_v be a random variable that denotes the total number of these k copies of v that are in M^s . X_v is a binomial random variable with f_v trials and the probability of success in each trial being $\frac{12k}{\lambda^2} \ln \frac{2}{\delta}$. It follows that $E[X_v] = \frac{12k^2}{\lambda^2} \ln \frac{2}{\delta}$. Using Chernoff bounds:

$$\begin{aligned} \Pr[E_1] &= \Pr[X_v < r] = \Pr \left[X_v < \frac{12k^2}{\lambda^2} \left(1 - \frac{\lambda}{2k}\right) \ln \frac{2}{\delta} \right] \\ &= \Pr \left[X_v < E[X_v] \left(1 - \frac{\lambda}{2k}\right) \right] \\ &\leq e^{-\frac{3}{2} \ln \frac{2}{\delta}} = \left(\frac{\delta}{2}\right)^{\frac{3}{2}} < \frac{\delta}{2} \end{aligned}$$

From Lemma 4.4.1, we have $\Pr[E_2] < \frac{1}{N}$. Using the bounds on $\Pr[E_1]$ and $\Pr[E_2]$ in inequality 4.9, and assuming $N > \frac{2}{\delta}$, we get:

$$\Pr[E] \leq \frac{\delta}{2} + \frac{1}{N} < \delta$$

■

Lemma 4.4.3 False Positive. *If u is an item with $f_u \leq k - \lambda$, where $k^{\frac{3}{4}} \leq \lambda < k$, then the probability that u is returned by some node as a frequent item is no more than δ .*

Proof: A false positive can occur if both these events happen (1) r or more copies of u are present in M^s ; let E_1 denote this event and (2) all r copies reach some node in the network through gossip; let E_2 denote this event. Let E denote the event that a false positive occurred.

$$\Pr[E] = \Pr[E_1 \cap E_2] \leq \Pr[E_1]$$

Let X_u denote the number of copies of u that were sampled. Consider the “best case” scenario for a false positive, when $f_u = k - \lambda$. Then X_u is a binomial random variable with $E[X_u] = (k - \lambda) \frac{12k}{\lambda^2} \ln \frac{2}{\delta} = \frac{12k^2}{\lambda^2} (1 - \frac{\lambda}{k}) \ln \frac{2}{\delta}$. Using Chernoff bounds:

$$\begin{aligned} \Pr[E_1] = \Pr[X_u > r] &= \Pr \left[X_u > \frac{12k^2}{\lambda^2} \left(1 - \frac{\lambda}{2k}\right) \ln \frac{2}{\delta} \right] \\ &= \Pr \left[X_u > E[X_u] \left(1 + \frac{\frac{\lambda}{2k}}{1 - \frac{\lambda}{k}}\right) \right] \\ &\leq \exp \left(- \left(\ln \frac{2}{\delta} \right) \left(\frac{1}{1 - \frac{\lambda}{k}} \right) \right) \\ &= \left(\frac{\delta}{2} \right)^{\frac{1}{1 - \frac{\lambda}{k}}} < \frac{\delta}{2} \text{ [since } \frac{1}{1 - \frac{\lambda}{k}} > 1] \end{aligned}$$

■

Lemmas 4.4.2, 4.4.3 and 4.4.1 together lead to the following theorem about the correctness of the algorithm.

Theorem 4.4.1 *Suppose algorithm 12 is run for $12N \ln 2N$ rounds. Then, with probability at least $1 - \delta$, any item with k or more occurrences in M will be identified as a frequent item at every node. With probability at least $1 - \delta$, any item with less than $k - \lambda$ occurrences in M will not be identified as a frequent item at any node.*

We next analyze the communication complexity of gossip. Since each node initializes its sketch with the sampled elements, but accumulates more elements as the gossip proceeds, the sizes of the messages exchanged grow as the algorithm progresses. We note that the number of elements exchanged between two nodes in any round is no more than the total number of sampled elements, hence, the number of rounds of gossip required, times the maximum message size is an upper bound on the communication complexity. Let \mathcal{Y} denote the total number of bytes that need to be exchanged in the network until the frequent items have been identified.

Theorem 4.4.2 (Communication Complexity for Absolute Threshold) *With high probability, $\mathcal{Y} = O\left(\frac{N\mathcal{N}k}{\lambda^2} \ln \frac{1}{\delta} \ln N\right)$*

Proof: Let Z denote the number of elements sampled during initialization, i.e. $Z = |M^s|$. Z follows a binomial distribution with \mathcal{N} trials, and probability of success in each trial equal to $\frac{12k}{\lambda^2} \ln\left(\frac{2}{\delta}\right)$. Thus $E[Z] = \frac{12\mathcal{N}k}{\lambda^2} \ln\left(\frac{2}{\delta}\right)$. Using Chernoff bounds:

$$\begin{aligned} \Pr\left[Z > \frac{18\mathcal{N}k}{\lambda^2} \ln\left(\frac{2}{\delta}\right)\right] &= \Pr\left[Z > \left(1 + \frac{1}{2}\right) \frac{12\mathcal{N}k}{\lambda^2} \ln\left(\frac{2}{\delta}\right)\right] \\ &\leq e^{-\frac{\mathcal{N}k}{\lambda^2} \ln\left(\frac{2}{\delta}\right)} = \left(\frac{\delta}{2}\right)^{\frac{\mathcal{N}k}{\lambda^2}} \\ &< \frac{\delta}{2} \text{ [since } \mathcal{N} > \lambda, k > \lambda] \end{aligned}$$

We note that $Z \cdot \mathcal{T}^s$ is an upper bound on \mathcal{Y} . Thus, if \mathcal{Y} is large, then either Z must be large, or \mathcal{T}^s must be large. More precisely, using the above bound on Z , and Lemma 4.4.1, we get:

$$\begin{aligned} \Pr\left[\mathcal{Y} > \frac{216N\mathcal{N}k}{\lambda^2} \ln\left(\frac{2}{\delta}\right) \ln 2N\right] &\leq \Pr\left[\left(Z > \frac{18\mathcal{N}k}{\lambda^2} \ln\left(\frac{2}{\delta}\right)\right) \cup (\mathcal{T}^s > 12N \ln 2N)\right] \\ &\leq \Pr\left[Z > \frac{18\mathcal{N}k}{\lambda^2} \ln\left(\frac{2}{\delta}\right)\right] + \Pr[\mathcal{T}^s > 12N \ln 2N] \\ &< \frac{\delta}{2} + \frac{1}{2N} < \delta \end{aligned}$$

■

4.5 Simulation Results

We used simulation to understand the following aspects of the algorithms that we developed. First, we wanted to know how easy it was to implement these algorithms. Next, since theoretical analysis is a pessimistic worst case analysis, we can expect the performance observed during simulation to be better than the theoretical predictions. We set out to measure by how much is the measured performance better than the theoretical predictions. Specifically, we measured the convergence time, and the error rate of the algorithm (these terms are described precisely below).

4.5.1 Input Data and Metrics Used.

We used two types of datasets for the simulations.

Data Set I: Pareto-like Distribution The first one was generated by a Pareto-like distribution. Given the domain of items $[m] = \{1, 2, \dots, m\}$, to make the frequent algorithms applicable, we needed a data distribution that can be made arbitrarily skewed, i.e., we wanted n ($n \ll m$) of the data items from $[m]$ to occur very frequently in the data set. During generating the data, we kept a parameter to specify the number of frequent items. We made all the frequent items equally probable, and the total probability of the frequent items was specified by yet another parameter. For example, if we want the data to have 10 frequent items, and the sum of the probabilities of these 10 items is 0.5, then the probability of generating each of these 10 frequent items would be 0.05. Hence, if we have 5,000 nodes, and 100 elements per node, then the expected number of occurrences of each item that we desire to be frequent is $5000 \times 100 \times 0.05 = 25,000$. We tried three different dataset sizes: 500000, 1000000 and 1500000. For each dataset size, we generated 10 different datasets, and for each of these 10 different datasets (of the same size), we formed the sketch and gossiped it 10 times, to average out the errors due to randomization. So each reading using this distribution (in Figures 4.2 and 4.3) is an average of 100 readings.

Dataset II: Zipfian+Uniform Distributions The second dataset was generated from a mixture of Zipfian and uniform distributions. Once again, we fixed a small number (n) of items from $[m]$ that would be generated with high frequency. According to the Zipfian distribution, the probability of the r^{th} most frequent item ($r \in \{1, 2, \dots, n\}$) was assigned by the following probability mass function:

$$f(r) = \frac{\frac{1}{r}}{\sum_{i=1}^n \frac{1}{i}}$$

The sum of the probabilities of these n frequent items was set to $\theta < 1$. The remaining $m - n$ items from the domain $[m]$ were all assigned equal probability (this is where the uniform distribution came in), which was $\frac{1-\theta}{m-n}$. For each of the relative and absolute error algorithm,

we worked with datasets of three different sizes: 500000, 1500000 and 4000000. To average out the errors due to randomization, we created 100 different datasets of each size, and for each dataset, we repeated the experiment 50 times, and averaged the error rate over these 5,000 runs; so each point in the plots in Figures 4.4 and 4.5 resulted from 5000 repetitions.

We analyze the *convergence time* and the *error rates* of the algorithms as functions of the network size and message cost of the gossip, respectively.

4.5.2 Convergence Time

Informally, a system is defined to have converged if it is in a configuration where every pair of nodes have “seen” each other’s sketch, either through direct or indirect communication. More precisely, we define that for two nodes i and j , node i has communicated directly with node j if i sent a message to j . We (recursively) define that node i has communicated indirectly with node j if there exists a node k such that i has communicated directly/indirectly with k and then k has communicated directly/indirectly with j .

Note that according to the above definition, system convergence is sufficient to ensure that further communication will not lead to any changes in the state of the sketch at any node. In Figure 4.1, we plot the number of rounds of gossip required for convergence as a function of the network size N . Because the convergence time for uniform gossip depends only on the network size, the same results apply for both absolute and relative thresholds. We note that all sampled elements at a node are disseminated together. Hence, the convergence time is a function only of the network size N , and does not depend on the size of the dataset, or on the size of the samples.

Error Rate: Since the algorithm is a randomized approximation algorithm, there is a small, but non-zero probability that the algorithm will fail, i.e. it would report infrequent items as frequent (false positive) and/or would fail to identify frequent items (false negative). The user specifies the degree of accuracy desired through the approximation error ψ (for relative threshold) or λ (for absolute threshold) and the error probability δ .

We now describe the *error rate* metric that we used for measuring the observed error in our

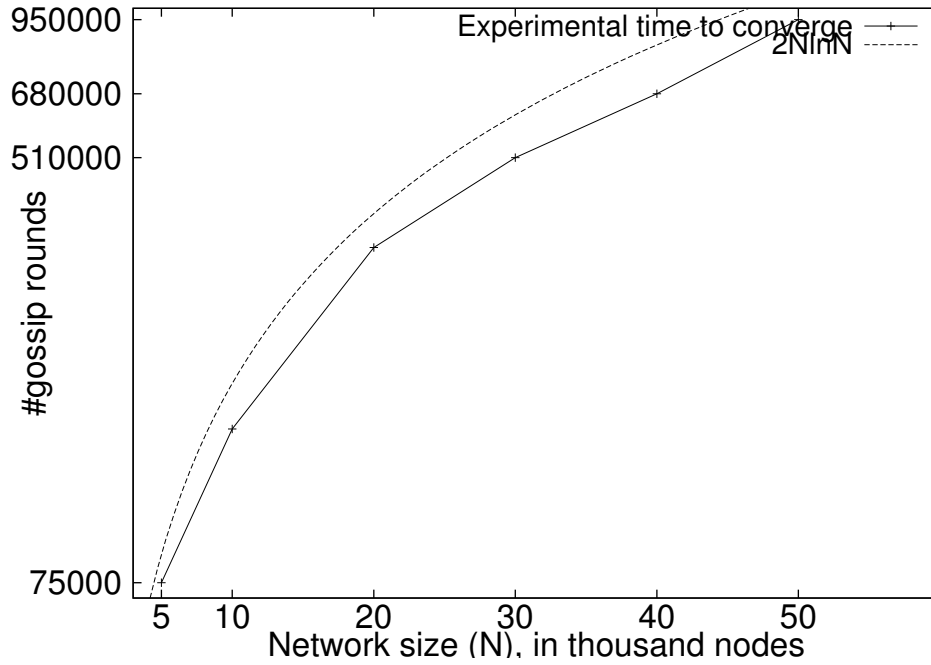


Figure 4.1: The number of rounds till convergence versus network size N .

experiments. Note that while the algorithm guarantees a low probability of error, we measure the actual fraction of time that an error occurred during the simulations. The *False Negative Rate* is defined as the ratio of the number of false negatives reported by a node to the number of data items that are frequent, i.e., what fraction of the frequent items were not identified as frequent by the node. The *False Positive Rate* is defined as the ratio of the number of false positives reported by a node to the number of data items that are not frequent, but have occurred at least once in the input. The *Error Rate* is defined as the maximum of the false negative and the false positive rates. Since all nodes attain the same state once convergence occurs, the error rate can be recorded from an arbitrarily selected node (we recorded it from node 0). To see whether we really needed a sketch size as large as predicted by theory, we measured the observed error rate at various sketch sizes.

For relative threshold, the theoretical sketch size was $t = \frac{c_r}{\psi^2} \ln \frac{3}{\delta}$ (with $c_r = 128$ as revealed by the analysis), so we tried sketch sizes for various values of the constant c_r . Figures 4.2 and 4.4 show the error rates as a function of the sketch size, for the Pareto-like distribution and

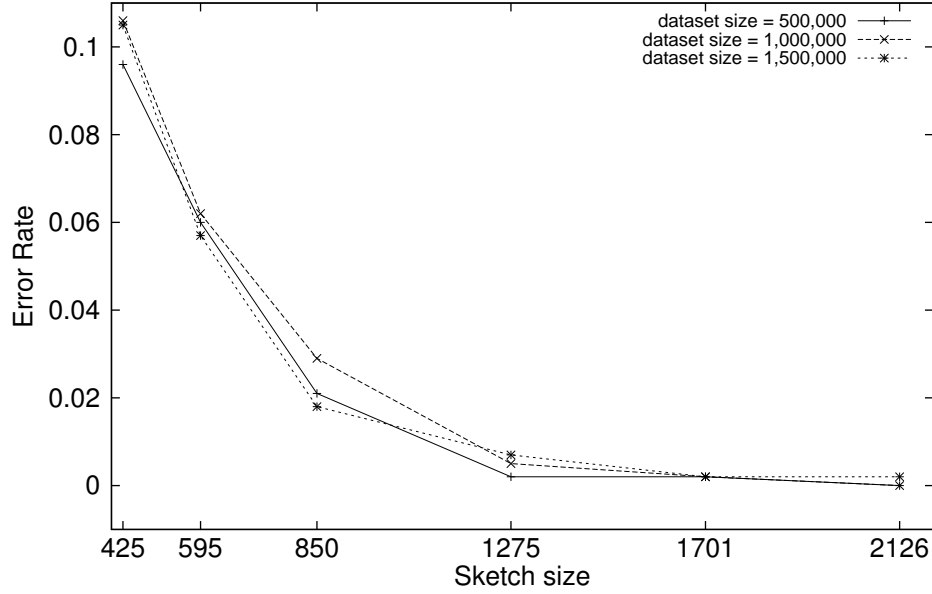


Figure 4.2: The error rate as a function of the sketch size for the relative error algorithm, with the dataset generated by the Pareto-like distribution. $\phi = 0.081$, $\psi = 0.02$ and $\delta = 0.1$.

the mixed distribution respectively.

For absolute threshold, the theoretical sampling probability was $\frac{c_a k}{\lambda^2} \ln \frac{2}{\delta}$ (with $c_a = 12$ as revealed by the analysis), so we tried sketch sizes for various values of the constant c_a . Figures 4.3 and 4.5 show the error rates as a function of c_a , for different dataset sizes and corresponding different values of k and λ , for the Pareto-like distribution and the mixed distribution respectively.

4.5.3 Observations

We make the following observations from our experience with the simulations, and results in Figures 4.1 to 4.5.

- The theoretical analysis predicted that for a system with N nodes, $12N \ln 2N$ rounds of gossip are sufficient for convergence, with high probability. By carrying out simulations with upto 50,000 nodes, we found from simulations that convergence was typically achieved with less than $2N \ln N$ rounds of gossip for all the values of N that we tried (see Figure 4.1).

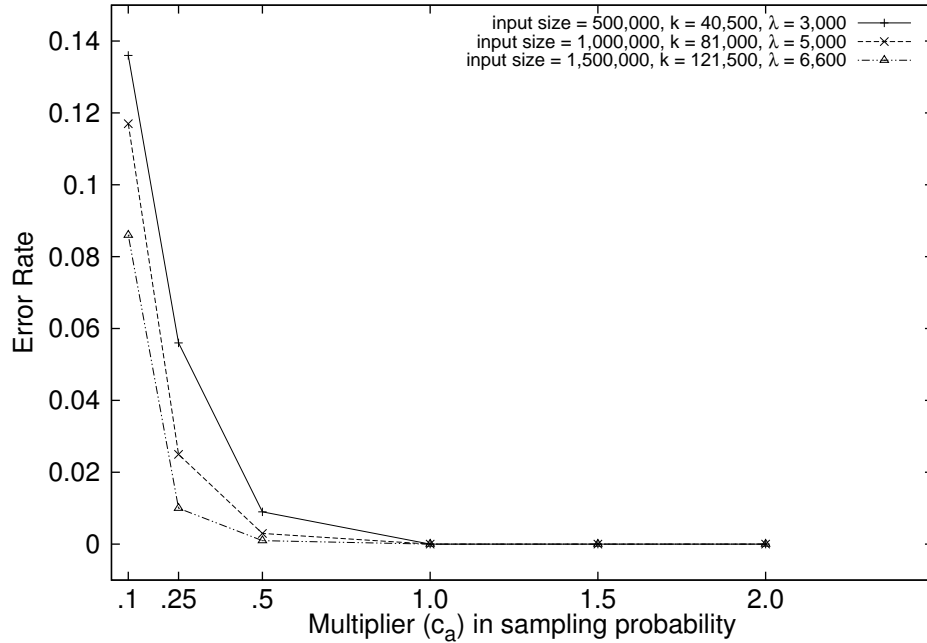


Figure 4.3: The error rate as a function of c_a , a multiplier in the sampling probability, for the absolute error algorithm. The dataset is generated by the Pareto-like distribution. Note that the expected sketch size increases linearly with the sampling probability. $\delta = 0.1$.

- For relative threshold, while the analysis showed that in the expression for the sketch size $\frac{c_r}{\psi^2} \ln \frac{3}{\delta}$, a constant factor $c_r = 128$ is necessary, we found in our experiments that a constant factor of $c_r = 0.25$ was sufficient in all cases to meet the desired error bounds. This indicates that in practice, the required sketch sizes may be much smaller than predicted by theory.
- For absolute threshold, while the analysis showed that in the expression for the sampling probability $\frac{c_a k}{\lambda^2} \ln \frac{2}{\delta}$, a constant factor $c_a = 12$ is necessary, we found in our experiments that a constant factor of $c_a = 2$ was sufficient in all cases to meet the desired error bounds. This indicates that in practice, the required sampling probability may be smaller than predicted by theory.

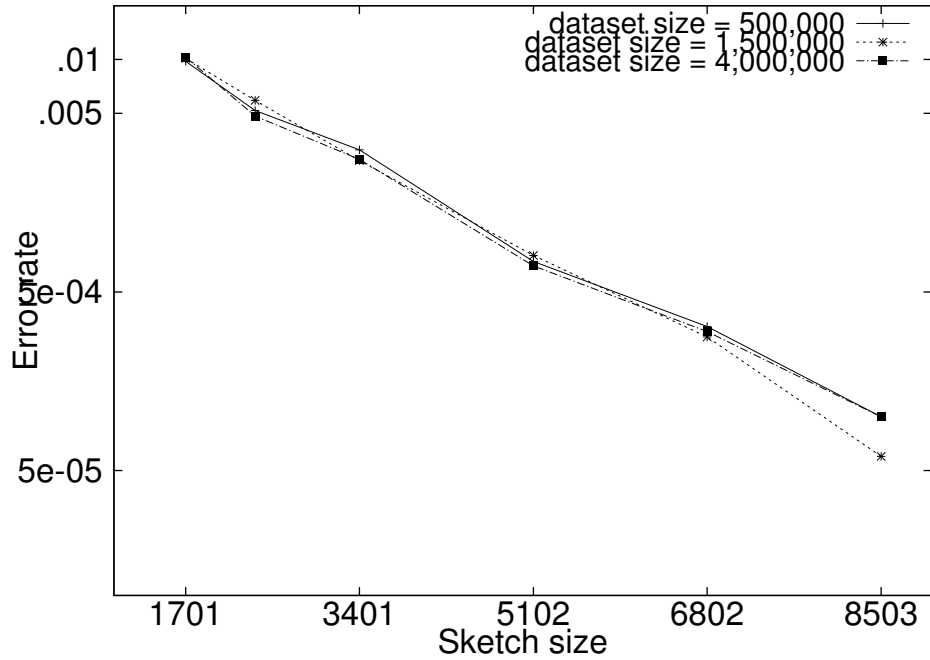


Figure 4.4: The error rate as a function of the sketch size for the relative error algorithm, with the dataset generated by the mixed distribution.

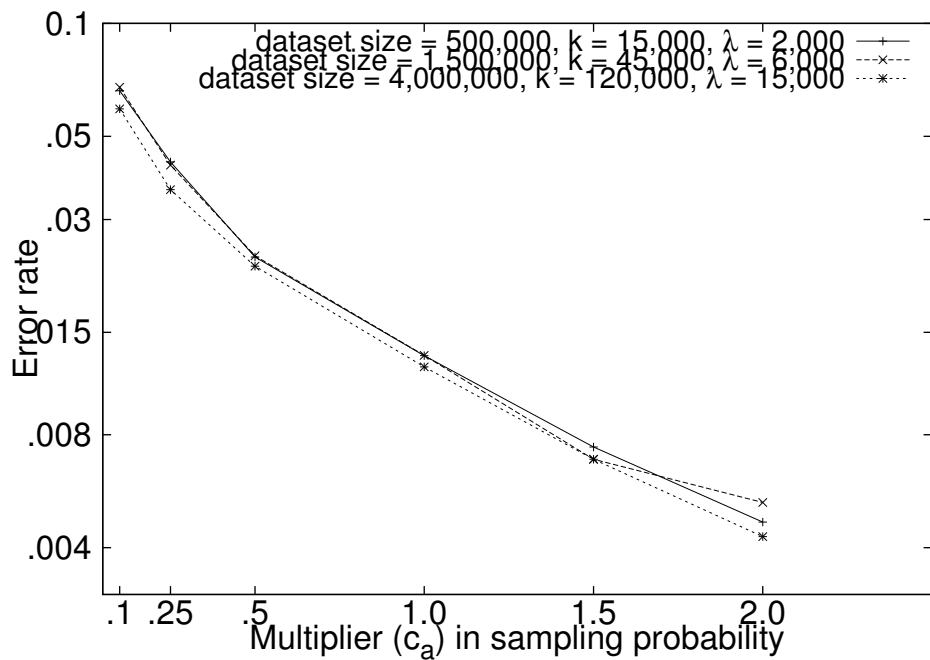


Figure 4.5: The error rate as a function of c_a , a multiplier in the sampling probability, for the absolute error algorithm. The dataset is generated by the mixed distribution.

4.6 Synchronous Model

In the synchronous communication model, all nodes transmit equally often. In each communication round, every node can send a message to one other (randomly chosen) node. We use a result due to Frieze and Grimmett [53], who considered the time to spread a rumor in a network. In their model, there is a rumor message that has to spread to everyone in a population of size N . Initially, a single person has the rumor. In every communication round, each person who already has the rumor conveys it to another randomly chosen person in the population, and we are interested in the number of rounds taken for the rumor to spread to all N nodes. Note the similarity to our model of synchronous gossip.

Theorem 4.6.1 (Frieze and Grimmett 1985) *Let T^N denote the number of rounds required to spread a rumor among a population of size N . Then, (1) $\lim_{N \rightarrow \infty} \frac{T^N}{\log_2 N} = \ln 2$ with high probability and, (2) For $\gamma > 0$, $\Pr[T^N > (1 + (\gamma + 1) \ln 2) \log_2 N] = o(N^{-\gamma})$*

Suppose that instead of a single rumor, there were α different rumors originating at different nodes, and all these rumors were being disseminated simultaneously among the N nodes. Let \mathcal{T}^α be the number of rounds required for all the nodes to receive all α rumors.

Lemma 4.6.1 *With probability $1 - o(\frac{1}{N})$, $\mathcal{T}^\alpha \leq (1 + 2 \ln 2) \log_2 N + \ln \alpha$.*

Proof: For $i = 1 \dots \alpha$, let t_i denote the number of rounds required to disseminate rumor i . Since all the α rumors are being disseminated simultaneously, we have $\mathcal{T}^\alpha = \max_{i=1}^\alpha t_i$. Using the union bound:

$$\Pr[\mathcal{T}^\alpha > x] = \Pr\left[\bigcup_{i=1}^{\alpha} (t_i > x)\right] \leq \sum_{i=1}^{\alpha} \Pr[t_i > x] = \alpha \Pr[T^N > x]$$

Using $\gamma = 1 + \log_N \alpha$ in Theorem 4.6.1, we get $\Pr[\mathcal{T}^\alpha > (1 + 2 \ln 2) \log_2 N + \ln \alpha] = o(\frac{1}{N^\alpha})$, and the result follows. ■

Our algorithms for the synchronous time model for relative and absolute thresholds, are described as Algorithms 13 and 14 respectively. These differ from the algorithms for the asynchronous models (Algorithms 11 and 12) in that in every round of communication, every

Algorithm 13: Synchronous gossip algorithm at node i for finding the frequent items with a relative threshold

Input: Data sets M_i ; error probability δ , relative frequency threshold ϕ , approximation error $\psi < \phi$

```

// Initialization
1  $t \leftarrow \frac{128}{\psi^2} \ln(\frac{3}{\delta})$ 
2  $S_i \leftarrow \Phi$ 
3 foreach  $\ell = 1$  to  $N_i$  do
4   | Choose  $w_i^\ell$  as a uniformly distributed random number in  $(0, 1)$ 
5   | Set  $S_i \leftarrow S_i \cup \{(i, \ell, m_i^\ell, w_i^\ell)\}$ 
6 end
// Gossip
7 foreach round of gossip do
8   | if sketch  $S_j$  is received from node  $j$  then
9     |  $S_i \leftarrow S_i \cup S_j$ 
10    | if  $|S_i| > t$  then
11      | retain  $t$  elements of  $S_i$  with the smallest weights
12    | end
13  | end
14  | select node  $j$  uniformly at random
15  | send  $S_i$  to  $j$ 
16 end
// Query
17 when queried for the frequent items
18 foreach  $v \in \{1, \dots, m\}$  do
19   | if at least  $(\phi - \frac{\psi}{2})t$  (nodeID, elementID, value, weight) tuples exist in  $S_i$  with value  $v$ 
20     | then
21       | report  $v$  as a frequent item
22   | end
23 end

```

Algorithm 14: Synchronous gossip algorithm at node i for frequent items with an absolute threshold k

Input: Data sets M_i ; error probability δ , frequency threshold k , approximation error λ

// Initialization

- 1 $S_i \leftarrow \Phi$
- 2 **foreach** $\ell = 1$ to N_i **do**
- 3 Choose ρ as a uniformly distributed random number in $(0, 1)$
- 4 **if** $\rho < \frac{12k}{\lambda^2} \ln \frac{2}{\delta}$ **then**
- 5 $S_i \leftarrow S_i \cup \{(i, \ell, m_i^\ell)\}$
- 6 **end**
- 7 **end**
- // Gossip
- 8 **foreach** round of gossip **do**
- 9 **if** sketch S_j is received from node j **then**
- 10 $S_i \leftarrow S_i \cup S_j$
- 11 **end**
- 12 select node j uniformly at random from $\{1, \dots, N\}$
- 13 send S_i to j
- 14 **end**
- // Query
- 15 **when** queried for the frequent items
- 16 **foreach** $v \in \{1, \dots, m\}$ **do**
- 17 **if** v occurs more than $r = \frac{12k^2}{\lambda^2} (1 - \frac{\lambda}{2k}) \ln \frac{2}{\delta}$ times in S_i **then**
- 18 report v as a frequent item
- 19 **end**
- 20 **end**

node sends a message. Note that the sampling probability, the sketch size and the thresholds for identification of frequent items in the algorithms for the asynchronous model also suffice for the synchronous model, so the analysis of the random sampling is the same as in the asynchronous model. The only change is in the gossip mechanism. We arrive at the following result:

Theorem 4.6.2 (Synchronous gossip) *If the synchronous algorithms 13 and 14 are run for $4\log_2 N$ rounds, then all frequent items (with relative and absolute thresholds, respectively) will be identified with probability at least $1 - \delta$, and no infrequent item will be identified, with probability at least $1 - \delta$.*

Proof: Similar to the asynchronous time model, in the synchronous model too, the analysis of gossip does not depend on how many elements each node begins with, or how many elements from each node find a place in the sketch; since all the elements from the (local) sketch of a single node get disseminated together. The number of such local sketches is trivially no more than N . If all these items are disseminated to all nodes, then the guarantees will be met. Substituting $\alpha \leq N$ in Lemma 4.6.1 yields the desired result. We can get a slightly tighter result (but asymptotically still the same) by using a better bound on the number of sampled items. ■

Note that for both absolute and relative thresholds, the number of rounds required in the synchronous model is less than that required by the asynchronous model by a factor of $\Theta(N)$ – this is to be expected, since in each round in the asynchronous model, a single message is exchanged while in each round in the asynchronous model, N messages are exchanged.

Bibliography

- [1] <http://www.tcpdump.org/>.
- [2] <http://www.wireshark.org/>.
- [3] <http://aircert.sourceforge.net/yaf/>.
- [4] <http://qosient.com/argus/>.
- [5] <http://oss.oetiker.ch/mrtg/>.
- [6] http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html.
- [7] <http://www.cisco.com/en/US/products/sw/netmgts/ps1964/index.html>.
- [8] <http://www.lancope.com/products/>.
- [9] <http://www.cisco.com/en/US/products/hw/routers/ps167/index.html>.
- [10] <http://www.sgi.com/tech/stl/>.
- [11] <http://www.winpcap.org>.
- [12] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. Tracking join and self-join sizes in limited storage. *Journal of Computer and System Sciences*, 64(3):719–747, 2002.
- [13] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.
- [14] Sergio A. Alvarez. An exact analytical relation among recall, precision, and classification accuracy in information retrieval.
- [15] Rohit Ananthakrishna, Abhinandan Das, Johannes Gehrke, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Efficient approximation of correlated sums on data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):569–572, 2003.

- [16] Austin Appleby. Murmurhash 2.0. <http://sites.google.com/site/murmurhash/>.
- [17] A. Arasu and G. Manku. Approximate counts and quantiles over sliding windows. In *Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 286–296, 2004.
- [18] Stefan Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1–7, 1999.
- [19] Brian Babcock, Mayur Datar, Rajeev Motwani, and Liadan O’Callaghan. Maintaining variance and k-medians over data stream windows. In *Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 234–243, 2003.
- [20] Nagender Bandi, Divyakant Agrawal, and Amr El Abbadi. Fast algorithms for heavy distinct hitters using associative memories. In *Proceedings of the 27th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 6–14, 2007.
- [21] Stephen P. Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Gossip algorithms: design, analysis and applications. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, pages 1653–1664, 2005.
- [22] Stephen P. Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Randomized gossip algorithms. *IEEE Transactions on Information Theory*, 52(6):2508–2530, 2006.
- [23] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing (STOC)*, pages 327–336, 1998.
- [24] CAIDA. OC48 traces dataset. <https://data.caida.org/datasets/oc48/oc48-original/20020814/5min/>.
- [25] CAIDA. OC48 traces dataset. <https://data.caida.org/datasets/oc48/oc48-original/20020814/5min/>.
- [26] Pei Cao and Zhe Wang. Efficient top-k query calculation in distributed networks. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 206–215, 2004.

- [27] Amit Chakrabarti, Khanh Do Ba, and S. Muthukrishnan. Estimating entropy and entropy norm on data streams. In *Proceedings of the 23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 196–205, 2006.
- [28] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proceedings of the 29th International Colloquium Automata, Languages and Programming (ICALP)*, pages 693–703, 2002.
- [29] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.
- [30] Aiyou Chen, Yu Jin, and Jin Cao. Tracking long duration flows in network traffic. In *Proceedings of the 29th IEEE International Conference on Computer Communications (INFOCOM)*, pages 206–210, 2010.
- [31] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. *Proceedings of the VLDB Endowment (PVLDB)*, 1(2):1530–1541, 2008.
- [32] Graham Cormode and Marios Hadjieleftheriou. Finding the frequent items in streams of data. *Commun. ACM*, 52(10):97–105, 2009.
- [33] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Diamond in the rough: Finding hierarchical heavy hitters in multi-dimensional data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 155–166, 2004.
- [34] Graham Cormode and S. Muthukrishnan. What’s hot and what’s not: tracking most frequent items dynamically. In *Proceedings of the 22nd ACM SIGMOD International Conference on Management of Data / Principles of Database Systems (PODS)*, pages 296–306, 2003.
- [35] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
- [36] Graham Cormode and S. Muthukrishnan. Space efficient mining of multigraph streams. In *Proceedings of the 24th ACM SIGMOD International Conference on Management of Data / Principles of Database Systems (PODS)*, pages 271–282, 2005.
- [37] Graham Cormode, Srikanta Tirthapura, and Bojian Xu. Time-decaying sketches for robust aggregation of sensor data. *SIAM Journal on Computing*, 39(4):1309–1339, 2009.

- [38] CS-MARS. http://www.cisco.com/en/US/products/ps6241/products_configuration_example09186a0080b19507.shtml.
- [39] Richard E. Cullingford. Correlation and collaboration in anomaly detection. In *Cybersecurity Applications & Technology Conference For Homeland Security (CATCH)*, pages 251–254, 2009.
- [40] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal of Computing*, 31(6):1794–1813, 2002.
- [41] Supratim Deb, Muriel Médard, and Clifford Choute. Algebraic gossip: a network coding approach to optimal multiple rumor mongering. *IEEE Transactions on Information Theory*, 52(6):2486–2507, 2006.
- [42] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon’s highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.
- [43] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Frequency estimation of internet packet streams with limited space. In *Proceedings of the 10th Annual European Symposium (ESA)*, pages 348–360, 2002.
- [44] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Principles of Distributed Computing (PODC)*, pages 1–12, 1987.
- [45] Alexandros G. Dimakis, Anand D. Sarwate, and Martin J. Wainwright. Geographic gossip: efficient aggregation for sensor networks. In *Proceedings of the Fifth International Conference on Information Processing in Sensor Networks (IPSN)*, pages 69–76, 2006.
- [46] Holger Dreger, Anja Feldmann, Michael Mai, Vern Paxson, and Robin Sommer. Dynamic application-layer protocol analysis for network intrusion detection. In *USENIX*, 2006.
- [47] Richard Durstenfeld. Algorithm 235: Random permutation. *Communications of the ACM*, 7(7):420, 1964.
- [48] Cristian Estan, Stefan Savage, and George Varghese. Automatically inferring patterns of resource consumption in network traffic. In *Proceedings of the ACM SIGCOMM 2003 Conference on Appli-*

cations, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), pages 137–148, 2003.

- [49] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *Proceedings of the ACM SIGCOMM 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 323–336, 2002.
- [50] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems*, 21(3):270–313, 2003.
- [51] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.
- [52] Prahlad Fogla, Monirul Sharif, Roberto Perdisci, Oleg Kolesnikov, and Wenke Lee. Polymorphic blending attacks. In *Proceedings of the 15th USENIX Security Symposium*, 2006.
- [53] A.M. Frieze and G.R. Grimmett. *The Shortest-Path Problem for Graphs with Random Arc-lengths*, volume 10. Elsevier Science Publishers Belsesten Vennootschap, 1985.
- [54] Yan Gao, Yao Zhao, Robert Schweller, Shobha Venkataraman, Yan Chen, Dawn Song, and Ming-Yang Kao. Detecting stealthy attacks using online histograms. In *International Workshop on Quality of Service*, 2007.
- [55] Johannes Gehrke, Flip Korn, and Divesh Srivastava. On computing correlated aggregates over continual data streams. In *Proceedings of the 20th ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 13–24, 2001.
- [56] John Gerth. Incorporating network flows in intrusion incident handling and analysis. In *FLOCON*, 2008.
- [57] Phillip B. Gibbons and Srikanta Tirthapura. Estimating simple functions on the union of data streams. In *Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 281–291, 2001.
- [58] Phillip B. Gibbons and Srikanta Tirthapura. Distributed streams algorithms for sliding windows. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 63–72, 2002.

- [59] F. Giroire, J. Chandrashekar, N. Taft, E. Schooler, and D. Papagiannaki. Exploiting temporal persistence to detect covert botnet channels. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 326–345, 2009.
- [60] Christos Gkantsidis, Milena Mihail, and Amin Saberi. Random walks in peer-to-peer networks. In *Proceedings of the 23rd Conference of the IEEE Communications Society (INFOCOM)*, 2004.
- [61] José M. González and Vern Paxson. Enhancing network intrusion detection with integrated sampling and filtering. In *Proceedings of the 9th International Symposium On Recent Advances In Intrusion Detection (RAID)*, pages 272–289, 2006.
- [62] M. Greenwald and S. Khanna. Space efficient online computation of quantile summaries. In *Proceedings of the 20th ACM International Conference on Management of Data (SIGMOD)*, pages 58–66, 2001.
- [63] Guofei Gu, Prahlad Fogla, David Dagon, Wenke Lee, and Boris Skoric. Towards an information-theoretic framework for analyzing intrusion detection systems. In *Proceedings of the 11th European Symposium on Research in Computer Security (ESORICS)*, pages 527–546, 2006.
- [64] S. Guha, J. Chandrashekar, N. Taft, and K. Papagiannaki. How healthy are today’s enterprise networks? In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement (IMC)*, pages 145–150, 2008.
- [65] Maya Haridasan and Robbert van Renesse. Gossip-based distribution estimation in peer-to-peer networks. In *Proceedings of the 7th International Workshop on Peer-to-Peer Systems (IPTPS)*, 2008.
- [66] Paul Helman, Gunar E. Liepins, and Wynette Richards. Foundations of intrusion detection. In *The 5th IEEE Computer Security Foundations Workshop (CSFW)*, pages 114–120, 1992.
- [67] L. Todd Herberlein, Gihan V. Dias, Karl N. Levitt, Biswanath Mukherjee, Jeff Wood, and David Wolber. A network security monitor. In *IEEE Symposium on Security and Privacy*, pages 296–305, 1990.
- [68] P. Indyk and D. Woodruff. Tight lower bounds for the distinct elements problem. In *Proceedings of the 44th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 283–288, 2003.

- [69] Jaeyeon Jung, Vern Paxson, Arthur W. Berger, and Hari Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy*, pages 211–225, 2004.
- [70] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the 29th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2010.
- [71] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28:51–55, 2003.
- [72] Srinivas R. Kashyap, Supratim Deb, K. V. M. Naidu, Rajeev Rastogi, and Anand Srinivasan. Efficient gossip-based aggregate computation. In *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 308–317, 2006.
- [73] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *Proceedings of the 44th Symposium on Foundations of Computer Science (FOCS)*, pages 482–491, 2003.
- [74] David Kempe and Jon M. Kleinberg. Protocols and impossibility results for gossip-based communication mechanisms. In *Proceedings of the 43rd Symposium on Foundations of Computer Science (FOCS)*, pages 471–480, 2002.
- [75] Ram Keralapura, Graham Cormode, and Jeyashankher Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 289–300, 2006.
- [76] Oleg Kolesnikov and Wenke Lee. Advanced polymorphic worms: Evading IDS by blending in with normal traffic.
- [77] Christopher Krügel, Thomas Toth, and Engin Kirda. Service specific anomaly detection for network intrusion detection. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC)*, pages 201–208, 2002.
- [78] Bibudh Lahiri, Ioannis Akrotirianakis, and Fabian Moerchen. Finding critical thresholds for defining bursts. In *Proceedings of the Data Warehousing and Knowledge Discovery - 13th International Conference (DaWaK)*, pages 484–495, 2011.

- [79] Bibudh Lahiri, Jaideep Chandrashekar, and Srikanta Tirthapura. Space-efficient tracking of persistent items in a massive data stream. In *Proceedings of the Fifth ACM International Conference on Distributed Event-Based Systems (DEBS)*, pages 255–266, 2011.
- [80] Bibudh Lahiri and Srikanta Tirthapura. Computing frequent elements using gossip. In *Proceedings of the 15th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 119–130, 2008.
- [81] Bibudh Lahiri and Srikanta Tirthapura. Finding correlated heavy-hitters over data streams. In *Proceedings of the 28th International Performance Computing and Communications Conference (IPCCC)*, pages 307–314, 2009.
- [82] Bibudh Lahiri and Srikanta Tirthapura. Identifying frequent items in a network using gossip. *Journal of Parallel and Distributed Computing*, 70(12):1241–1253, 2010.
- [83] Ashwin Lall, Vyas Sekar, Mitsunori Ogihara, Jun Xu, and Hui Zhang. Data streaming algorithms for estimating entropy of network traffic. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance)*, pages 145–156, 2006.
- [84] Lap-Kei Lee and H. F. Ting. A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In *Proceedings of the 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 290–297, 2006.
- [85] Zhichun Li, Anup Goyal, Yan Chen, and Vern Paxson. Automating analysis of large-scale botnet probing events. In *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 11–22, 2009.
- [86] Amit Manjhi, Vladislav Shkapenyuk, Kedar Dhamdhere, and Christopher Olston. Finding (recently) frequent items in distributed data streams. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 767–778, 2005.
- [87] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, pages 346–357, 2002.

- [88] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th International Conference on Database Theory (ICDT)*, pages 398–412, 2005.
- [89] Jayadev Misra and David Gries. Finding repeated elements. *Science of Computer Programming*, 2(2):143–152, 1982.
- [90] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.
- [91] David Moore, Colleen Shannon, Douglas J. Brown, Geoffrey M. Voelker, and Stefan Savage. Inferring internet denial-of-service activity. *ACM Transactions on Computer Systems*, 24(2):115–139, 2006.
- [92] Damon Mosk-Aoyama and Devavrat Shah. Computing separable functions via gossip. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 113–122, 2006.
- [93] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.
- [94] George Nychis, Vyas Sekar, David G. Andersen, Hyong Kim, and Hui Zhang. An empirical evaluation of entropy-based traffic anomaly detection. In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement (IMC)*, pages 151–156, 2008.
- [95] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.
- [96] David Plonka. Flowscan: A network traffic flow reporting and visualization tool. In *Proceedings of the 13th Systems Administration Conference (LISA)*, pages 305–317, 2000.
- [97] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. An Analysis of the iKeeB (duh) iPhone botnet (worm). <http://mtc.sri.com/iPhone/>.
- [98] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement (IMC)*, pages 41–52, 2006.

- [99] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 13th Systems Administration Conference (LISA)*, pages 229–238, 2000.
- [100] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: new aggregation techniques for sensor networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 239–249, 2004.
- [101] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *IEEE Symposium on Security and Privacy*, pages 305–316, 2010.
- [102] Stuart Staniford, James A. Hoagland, and Joseph M. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, 10(1/2):105–136, 2002.
- [103] Srikanta Tirthapura and David Woodruff. A general method for estimating correlated aggregates over a data stream. In *(to appear) Proc. IEEE International Conference on Data Engineering (ICDE)*, 2012.
- [104] Thomas Toth and Christopher Krügel. Accurate buffer overflow detection via abstract payload execution. In *Recent Advances in Intrusion Detection: 5th International Symposium (RAID)*, pages 274–291, 2002.
- [105] Advanced Persistent Threat. <http://www.usenix.org/event/lisa09/tech/slides/daly.pdf>.
- [106] Botnet Reporting and Termination. http://spamtrackers.eu/wiki/index.php/Botnet_Reporting.
- [107] Google AdWords. <http://www.google.com/ads/adwords2/>.
- [108] CERT advisory CA-1996-21 TCP SYN flooding and IP spoofing attacks. <http://www.cert.org/advisories/CA-1996-21.html>.
- [109] CERT advisory CA-1996-01 UDP port denial-of-service attack. <http://www.cert.org/advisories/CA-1996-01.html>.
- [110] Shobha Venkataraman, Dawn Xiaodong Song, Phillip B. Gibbons, and Avrim Blum. New streaming algorithms for fast detection of superspreaders. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.
- [111] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. In *Recent Advances in Intrusion Detection: 7th International Symposium (RAID)*, pages 203–222, 2004.

- [112] David P. Woodruff. Optimal space lower bounds for all frequency moments. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 167–175, 2004.
- [113] Kuai Xu, Zhi-Li Zhang, and Supratik Bhattacharyya. Profiling internet backbone traffic: behavior models and applications. In *Proceedings of the Annual Conference of the Special Interest Group on Data Communication (SIGCOMM)*, pages 169–180, 2005.
- [114] Kuai Xu, Zhi-Li Zhang, and Supratik Bhattacharyya. Reducing unwanted traffic in a backbone network. Appeared in the Proceedings of the Steps to Reducing Unwanted Traffic on the Internet Workshop(SRUTI), 2005.
- [115] Vinod Yegneswaran, Paul Barford, and Johannes Ullrich. Internet intrusions: global characteristics and prevalence. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 138–147, 2003.
- [116] Linfeng Zhang and Yong Guan. Variance estimation over sliding windows. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 225–232, 2007.
- [117] Linfeng Zhang and Yong Guan. Detecting click fraud in pay-per-click streams of online advertising networks. In *Proceedings of the 28th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 77–84, 2008.
- [118] Yin Zhang, Sumeet Singh, Subhabrata Sen, Nick G. Duffield, and Carsten Lund. Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications. In *Internet Measurement Conference (IMC)*, pages 101–114, 2004.
- [119] Qi Zhao, Mitsunori Ogihara, Haixun Wang, and Jun Xu. Finding global icebergs over distributed data sets. In *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 298–307, 2006.