2017

Distributed Data Streaming Algorithms for Network Anomaly Detection

Wenji Chen
Iowa State University

Distributed data streaming algorithms for network anomaly detection

by

Wenji Chen

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:

Yong Guan, Major Professor
Jennifer Lee Newman
Srikanta Tirthapura
Daji Qiao
Doug Jacobson

Iowa State University

Ames, Iowa

2017

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank my advisor Yong Guan for his support and guidance through the program of PhD study. I would also like to thank my committee members to give me valuable suggestions to complete my thesis. I'm also glad to have many valuable friends around me during my stay at Iowa State University which have been inspiring my life since I came. At the last but not the least, I would like to thank my parents, Qinzhen Xie and Qingan Chen, for their understanding and supporting for all these years.

ABSTRACT

Network attacks and anomalies such as DDoS attacks, service outages, email spamming are happening everyday, causing various problems for users such as financial loss, inconvenience due to service unavailability, personal information leakage and so on. Different methods have been studied and developed to tackle these network attacks, and among them data streaming algorithms are quite powerful, useful and flexible schemes that have many applications in network attack detection and identification. Data streaming algorithms usually use limited space to store aggregated information and report certain properties of the traffic in short and constant time.

There are several challenges for designing data streaming algorithms. Firstly, network traffic is usually distributed and monitored at different locations, and it is often desirable to aggregate the distributed monitoring information together to detect attacks which might be low-profile at a single location; thus data streaming algorithms have to support data merging without loss of information. Secondly, network traffic is usually in high-speed and large-volume; data streaming algorithms have to process data fast and smart to save space and time. Thirdly, sometimes only detection is not useful enough and identification of targets make more sense, in which case data streaming algorithms have to be concise and reversible.

In this dissertation, we study three different types of data streaming algorithms: hot item identification, distinct element counting and superspreader identification. We propose new algorithms to solve these problems and evaluate them with both theoretical analysis and experiments to show their effectiveness and improvements upon previous methods.

CHAPTER 1. OVERVIEW

1.1 Motivations

1.1.1 Network Attacks and Anomalies

Computer network has become an important part of people's life in the near decades. There are countless examples that people make use of the computer network to support and enjoy their everyday life: companies run their business over the network; customers use it to purchase services and merchandise without walking out of their doors; people share their experience and ideas on social network websites. However, together with the flourishing of the computer network, related security issues have also mushroomed: (D)DoS attacks, worm/virus spreading, email spamming, service outages, and botnets, to name a few of them.

- **Distributed Denial of Service** (DDoS) attacks are used by malicious parties to flood victim's network service to make it unavailable to intended users, and now are also used as smokescreen to steal customer data or intellectual properties 201 (b). On average 28 DDoS attacks happen every hour in 2013 according to a report 201 (c) and cost companies considerable amounts of money each year.

- **Service outage** is sometimes caused by outside attacks such as DDoS and sometimes the inside software or hardware problems. It would render network services unavailable to the intended users and cause financial loss. An example is the service outages of Amazon's Cloud Service AWS which bring down a portion of clients'

business services and make them suffer from losing revenue and customer faith everytime it happens.

- **Worm Spreading** usually starts from one or several source(s) and expands to a large scale of hosts in the network very quickly. Each infected host will try to connect to other hosts to spread the worm as quickly as possible to consume network bandwidth or corrupt files on computers.

1.1.2 Countermeasures

Various kinds of methods have been developed to detect and defend against different network attacks and anomalies. The basic idea is that network traffic could be monitored to capture multiple features or patterns, and then models, signatures or basic profiles of the traffic could be created based on these observations, which would be used as baselines to detect attacks and anomalies that may behave differently from the baselines.

Traditional Firewalls are used by network administrators to filter network traffic coming into and going out of their network by comparing signatures of the packets, to detect worms, malwares, port-based DoS attacks, illegal connections and other anti-rule packets. However, firewalls are not very useful for detecting DDoS attacks, service outages and so on, because they are not designed for such attacks and anomalies involving a large number of parties 201 (a).

Network monitoring tools such as NetFlow are used on routers to collect IP network traffic flowing through them. Such tools usually aggregate IP packets into flows at each router and then export them to a central collector or monitoring center for processing, analyzing, and profiling. Such tools are the first defensive line to detect attacks such as DDoS and worm spreading, since they have access to the traffic that is just sent out by the malicious parties. However, there are two major problems with such monitoring tools. One problem is that they keep logs of traffic data at monitoring center which often cost huge communication overhead to transmit, large amount of space to store and

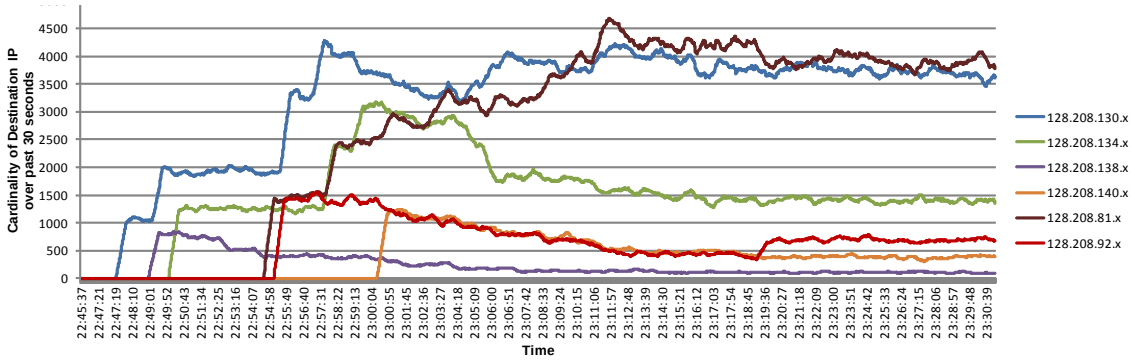


Figure 1.1 The time series of number of distinct destination IP addresses of the packets in each sub-net during the breakout of the witty worm.

a relatively long time to analyze. Another problem is that they usually use sampling techniques to reduce the load of data to be collected and analyze the sampled data regardless of their importance, which may result in biased and not accurate results.

In the near decades, data streaming algorithms have been proposed to help detect network attacks and anomalies. The idea is that network traffic could be modelled as data streams, and different features of the traffic could be captured and maintained in low cost of space as well as time, while estimated with high accuracy by data streaming algorithms. For example, in figure 1.1 each line represents the time series of cardinalities of the destination IP addresses in the packets sent from the hosts in the corresponding source IP group during the breakout of the Witty worm. We can see that each line shows a sudden increase from zero to several hundred or even thousand which is caused by the large number of random connections from the infected hosts in those small groups of source IPs. In this example, the network packets going through network monitors could be modelled as data streams; the feature we want to capture here is the cardinality of destination IP addresses for each source IP address. Since there are very possibly more than one network monitors and thus distributed data streams, we want to calculate the features locally at each monitor and then gather the aggregated compact information at a central location to get the final results.

Data streaming algorithms overcome the shortcomings of some network monitoring tools. They consume less space and time which is a key feature that makes them suitable for high-speed network traffic monitoring. They analyze the whole traffic and target the most important information related to specific problems, and produce reliable and accurate estimation results, while sampling techniques usually work on a small portion of the traffic which may lead to biased results.

There are different data streaming problems that have been studied by researchers, such as hot items/heavy hitters, superspreaders, distinct elements counting, dynamic membership querying, trend detection, and so on, which are often used for different network traffic monitoring purposes. For example, elephant flows that take up a large portion of network bandwidth are monitored to detect DDoS attacks and control quality of services. Elephant flows can be modelled as hot items in data streams that have very high frequencies and identified by hot item identification algorithms. Another example is the service outage detection which could be modelled as distinct element counting problem that used to detect DDoS attacks and monitor quality of service by network providers.

In this dissertation, we study three of the data streaming problems: hot item identification, distinct element counting and superspreader detection. We propose several algorithms and frameworks to resolve these problems in distributed and dynamic data streams, using combined techniques and small-size data structures which allow fast process. We use theoretical analysis and experimental evaluations to show that our algorithms introduce less overhead and improve upon previous data streaming algorithms in terms of space and time cost, as well as accuracy.

1.2 Challenges

There are several challenges for designing data streaming algorithms to defend against network attacks and anomalies.

- Firstly, the huge amount and high speed of Internet traffic put constraints on the space and time costs of the detection methods. Data streaming algorithms have to process data fast (in wire speed) and smart to save space and time. Saving data in disk is not tolerable and algorithms that could fit in memory is the minimum requirement. With constraints on space and time, estimation instead of exact computation is the most efficient technique and the accuracy of estimations is another challenge.

- Secondly, network traffic is usually distributed and monitored at different locations, and it is often desirable to aggregate the distributed monitoring information together to detect attacks which might be low-profile at a single location; thus data streaming algorithms may have to support information merging at low communication and processing costs without loss of accuracy.

- Thirdly, sometimes only detection is not useful enough and identification of targets make more sense, in which case data streaming algorithms have to be concise and reversible. Data streaming algorithm usually uses compact data structures to represent aggregated information of data which obscures the original identities and it is hard to maintain or recover these original identities of the targets.

CHAPTER 2. PRELIMINARIES

2.1 Data Streaming Models

A general data stream S is a series of data items in the following format:

$$(x_1, v_1), (x_2, v_2), \ldots, (x_i, v_i), \ldots$$

where x_i is the identifier (ID) of the i_{th} item and v_i is an update of the item's value. All the IDs of the items belong to a universe U with size $|U| = N$, and without loss of generality we suppose any ID $x_i \in [0, N-1]$. Value v_i is usually integer values in a certain field. An example is that when monitoring the flow sizes sent from different hosts, the IDs would be the source IP addresses in the IP packet and the values would be the packet sizes which are always positive.

According to the ranges of the items' values, data stream models can be categorized into two different types:

- **Insertion-only Model**. In this model, the values of items can only be positive. A special case is that values are always 1. This model is sometimes referred to cash register model Gilbert et al. (2001).

- **Dynamic Model**. In this model, the values of items can be either positive or negative. And this model is sometimes referred to turnstile model.

Whether it is to count the number of distinct elements or to identify hot items in the data stream, the algorithm would be very simple if we have enough space and time to

store all the information of the data stream. However, this is often neither practical nor efficient. So we are interested in developing data streaming algorithms which are often probabilistic and use small space as well as running time to approximate the values we want with guaranteed low error and high success probability. The goals of data stream algorithms are formally defined as follows:

Definition 1. *Let $f(S)$ be the value we want to calculate on data stream S, given expected error bound $0 < \epsilon < 1$ and failure probability $0 < \delta < 1$, a data streaming algorithm would give an approximate result $\tilde{f}(S)$ satisfying*

$$||f(S) - \tilde{f}(S)|| \leq \epsilon ||f(S)|| \tag{2.1}$$

with probability at least $1 - \delta$, using space $O(poly(\frac{1}{\epsilon}, \log \frac{1}{\delta}, \log N, \log w))$ and expected updating time $O(1)$.

2.2 Basic Techniques

We review some basic techniques that are often used in data streaming algorithms to reduce data size and extract important information from data streams that usually have very large size and often require fast processing. Many of the techniques below are also used by our approaches in the following chapters to solve different types of problems.

2.2.1 Normalization and Randomization

The distribution of the real world data is not predictable and may change over time, so it is not realistic to cast the data into a fixed model. When monitoring and analyzing the traffic, one technique to simplify the data is normalization. By normalization, here we mean applying some random mapping on the data such that the resulted data would have a uniform distribution over the mapped field and it would become easier to apply statistical analysis on the mapped data.

Universal hash functions Carter and N.Wegman (1979) are good tools for data normalization because they are easy to implement and have good performance on randomization and anti-collisions. Pairwise independent hash functions are required by some algorithms to guarantee that the randomization is uniform, which means that all hash values are equally likely.

2.2.2 Sampling

The incredible size of network traffic also makes analysis more difficult in terms of time, space and complexity. To mitigate this, sampling is a quite powerful technique which could reduce the burden of heavy computation and at the same time remove unrelated or less-important information from the data. The basic sampling scheme is fixed rate sampling: for each data item, toss a biased coin to determine if it would be sampled or not. For example, for hot item identification, we could use fixed sampling rate α to sample and keep the items and those elements whose frequency is larger than α would very likely be sampled and thus identified.

Sometimes, distinct sampling would be more efficient when we care more about the number of distinct elements rather than their frequencies, such as distinct element counting and superspreader detection. Distinct sampling means we sample the distinct elements using a fixed rate, regardless of how frequently it appears. This could be modeled as sampling over a set of distinct elements. The classic method for distinct sampling is firstly proposed in Flajolet and Martin (1985) where each element is randomly mapped to a binary string and sampled according to its binary representation: if the sampling rate is $\frac{1}{2^r}$ then an element would be sampled if its least significant bit is the r-bit from the left. This method is very powerful because its sampling rate is closely related to the representation of the item and we could adapt/reduce the sampling rate if the size of the distinct element set increases.

2.2.3 Sketching

Sketching refers to data structures that usually uses hash functions to randomize data and stores aggregated information of the data in compact space. Because sketches are usually space-efficient and fast to update, they are suitable for applications where distributed systems need to share information in an time-and-space efficient way. The famous sketches would be the different variations of Bloom filters and the count-min sketch.

A Bloom filter is a compact bit array which represents a set of distinct elements and supports membership queries: to answer the question that if an element is in the set or not with low false positive error rate and no false negative errors. A good survey of Bloom filters is available in Broder et al. (2002). A generalization of basic Bloom filter is the Counting Bloom filter which is proposed in Fan et al. (2000) and extends the bit array to an array of counters which supports both insertions and deletions of elements. There are some other variations of Bloom filters such as Dynamic Bloom filters in Guo et al. (2010) and Incremental Bloom filters in Hao et al. (2008) which works for dynamic data streams whose size is not known in advance.

Another famous sketch is the count-min sketch which is proposed in Cormode and Muthukrishnan (2005a). This is yet an extension of the Counting Bloom filter and use 2-dimensional array of counters to represent the frequencies of elements. We apply similar ideas in our approaches and use multi-dimensional counters to fit into different properties of the problems.

2.2.4 Miscellaneous

2.2.4.1 Balls-and-bins Model

Balls-and-bins model is a probabilistic model that enables estimating number of distinct elements in an efficient way. This model says that when randomly throwing A balls

into K bins where $A < K$, the probability of collisions of balls in a bin is very low, the variance of number of non-empty bins is bounded and small, and we could use the expected value of non-empty bins to estimate the number of distinct elements. This model is elegant when combined with the adaptive distinct sampling method to estimate the number of distinct elements in a large data stream, because adaptive distinct sampling could give us a small set of sampled distinct balls whose size is always bounded and we do not have to change the bins when the size of the real data set grows. The random throwing of balls could be modeled by pairwise independent hash functions.

2.2.4.2 Group Testing

Group testing is a procedure to identify a small part of a set which have certain properties that are different from other elements of the set, by testing on subsets instead of testing each individual element. Group testing fits into problems such as hot item identification and superspreader identification where the top-K elements are our targets and have different properties from the remaining elements: the hot items have higher frequencies than other non-hot items and superspreaders have larger size of cardinalities than others.

It is usually easy to identify if there is only one "particular" item within the set that has different property from others, and it becomes more difficult if there are more than one. In the later case, we could use divide-and-conquer principle to break down the problem further: randomly divide the whole set into several subsets and hope each subset will contain at most one "particular" item such that we can do group testing on each subset to find out the "particular" item if there is one.

CHAPTER 3. HOT ITEM IDENTIFICATION

3.1 Introduction

In this chapter, we will study the problem of detecting and identifying hot items in data streams. An item is defined as hot if its aggregated value takes larger than a threshold portion of the total value of all the items in the traffic. For example, in an IP packet stream going through a router, the items could be defined as the source IPs and the values of items could be the packet sizes; a hot item would be a source IP whose total packet size is larger than $\frac{1}{k+1}$ of the total packet size passing through the router. Hot items are also referred to as heavy hitters or icebergs in some papers and they have been found useful for detecting DDoS attacks Ayres et al. (2006), discovering worms Cheetancheri et al. (2007), finding frequently accessed content in Content Delivery Networks and P2P systemsLi and Lee (2008).

There are two different cases to be considered when solving the problem of identifying hot items in network traffic. One case is that we have a single monitor node and we only have to monitor a single data stream with is going through this monitor. The other case is that we have multiple distributed monitor nodes and each of them monitors a data stream. In the later case, usually the data in the multiple data streams have to be combined together to get the final answers.

Space and time constraints are two challenges for us to design solutions to the problem of identifying hot items in network traffic streams. The network traffic goes through network monitoring nodes at a very fast speed, for example, a router's throughput can

be several Gbps. It is not realistic to store all the data in to disks and analyze them later. Usually sampling or sketching, which uses much smaller space and processing time, is used to keep a summary of the original data and estimate some statistics of the original data. However, using summaries of data to estimate the original data means that we may generate estimation errors. So how to reduce the errors is another challenge for sampling and sketching based methods. In some monitoring applications, we need methods which can process data (nearly) on-line and report network anomalies in a timely manner. In such cases, data structures that can fit into memory and algorithms that can process data on-line are preferred.

The literature on the problem of identifying hot items is rich. In the single data stream case, there have been a bunch of work done to identify hot items and they can be roughly separated into two categories: counter-based algorithms such as FREQUENT algorithm in MISRA and GRIES (1982), LossyCounting algorithm in Manku and Motwani (2002), SpaceSaving algorithm in Metwally et al. (2005); sketch-based algorithms such as Count sketch in Charikar et al. (2002), Count-Min sketch in Cormode and Muthukrishnan (2005a), Group Testing based sketch in Cormode and Muthukrishnan (2005b). In multiple data streams case, there are also some work in recent years trying to solve the problem by either using gossip-based algorithms Jelasity et al. (2005)Sacha and Montresor (2013) or combining sampling and sketch together Zhao et al. (2006)Zhao et al. (2010)Huang et al. (2011).

Most of the above mentioned methods have their own short-comings. For example, counter-based methods can only work for data streams whose items' values are positive only and they will generate both false positives and false negatives; most of the sketch-based algorithms need a time-consuming recovery process to recover the identities of the hot items because the sketches only maintain the information of aggregated values but not the item IDs; gossip-based algorithms need large space and time overhead at each monitor node to maintain the information of local items; the space and time costs of

sampling-based algorithms heavily depend on the distributions and sizes of data streams and are not stable.

We propose a method to detect and identify hot items in either a single data stream or multiple distributed data streams with the following guarantees:

1. go over the whole data streams only once and identify all the hot items with high probability and low false positive rate;

2. use small constant space and updating time at each data node to maintain the information of the local data stream;

3. need small communication overhead to merge the information from all the distributed data nodes;

4. identify hot items in distributed data streams using small space and time;

5. run faster and generate less false positives and false negatives than previous methods.

3.2 Related Work

3.2.1 Hot Items in Single Data Stream

There has been a lot of work done on finding hot items in a single data stream and many of them can be found in the survey Cormode and Hadjieleftheriou (2010). These methods could be categorised into two types: counter-based methods and sketch-based methods. Counter-based methods such as MISRA and GRIES (1982)Demaine et al. (2002)Manku and Motwani (2002) Metwally et al. (2005) use a fixed amount of counters, as a summary of the data stream, to maintain the hot items deterministically and can report the hot items with bounded error. Although counter-based methods use small space, usually within 1MB, they can only handle incremental data streams,

which contain only positive update values. If the data stream is dynamic, which means that it contains both positive and negative update values, then counter-based methods will become short-sighted and mis-classify some non-hot items, which are hot at the beginning of the data stream and not hot at the end of it, as hot.

Sketch-based methods such as Count-Min Cormode and Muthukrishnan (2005a), Count-SketchCharikar et al. (2002), Group-Testing-based-Sketch Cormode and Muthukrishnan (2005b) use random projections based on hash functions to maintain aggregated information of all the items' update values. For recovering the identities of the hot items, some of them such as Count-Min and Count-Sketch have to use an additional heap-like data structure to maintain the current hot items in order to save recovery time, otherwise they have to go through each possible item ID and query the data structure to estimate its frequency. However, in such cases, these methods cannot handle dynamic data streams. To the best of our knowledge, the Group-Testing-based-Sketch in Cormode and Muthukrishnan (2005b) gives the best sketch-based method for identifying hot items in a single dynamic data stream in terms of space, update time and recovery time.

3.2.2 Hot Items in Distributed Data Streams

When the update values of items are distributed in multiple separated data streams, the local maintenance cost at each data node and the communication cost for merging the information maintained at each data node to identify the hot items have to be considered.

Some of the methods for single data stream can be extended to solve the problem. It has been shown in Berinde et al. (2009) that for counter-based methods, counter-summaries of distributed data streams can be merged together at a center node to generate a counter-summary of the union of the data streams, providing that the counter-summary for each single stream uses a large enough number of counters to guarantee the error bounds. Both the local calculation cost and the data communication cost are low for such extended methods. However they cannot handle dynamic data streams due to

the nature of counter-based methods: the counters are monotonous and are not able to track the dynamic changes in the data streams.

For sketch-based methods, the nature of the sketches allows them to be combined together to create a new sketch for the union of multiple data streams. For example, the Group-Testing-based-Sketch in Cormode and Muthukrishnan (2005b) uses a two dimensional array of counters and each counter calculates the summation of values of those items who are mapped to this counter. So when there are multiple data streams and corresponding sketches, if they use the same projections, i.e. hash functions, then we could summing up the counters from the multiple sketches to create a new sketch for the union of the data streams.

Recently several methods have been proposed to identify hot items in multiple distributed data streams. Some of them Jelasity et al. (2005)Sacha and Montresor (2013) are based on the idea of gossip algorithm. In gossip algorithm, each data node uses counter-based method to locally maintain a list of items with high local frequencies, and gossip with their neighbours to exchange information of their top-listed items; after multiple rounds of gossiping, the information will converge to generate a same list of global hot items at each node. Such gossip algorithms do not require a center node, however they need multiple rounds of gossiping which involves a large communication overhead. What's more, the local usage of counter-based method limits such gossip algorithms to be used only for incremental data streams.

Some other methods Zhao et al. (2006)Zhao et al. (2010)Huang et al. (2011) combines sampling and sketches locally which samples the items with a constant probability and maintains a sketch to estimate the aggregate values for the sampled items, and the sampled items as well as the sketch are sent to a central node to be combined together for global hot item identification or detection. These methods have some drawbacks: firstly they only work for incremental data streams; secondly the space usages are not constant and heavily depend on the data stream. These drawbacks make them not

desirable comparing to previously discussed extended counter-based and sketch-based methods.

There have been some work done Li and Lee (2008)Gangam et al. (2013) to identify hot items in distributed data streams exactly, without any false positives or false negatives. The netFilter proposed in Li and Lee (2008) organizes the nodes in a hierarchical way; at each node it separates the local items into disjoint groups using hash functions and aggregates the values for each group; the groups' aggregated values are sent from the leaf nodes to the root node in the hierarchy to filter out candidate groups which contain hot items; then the root node will send back the candidate group information to all the other nodes; in a second round, each node will find out local items that are in the candidate groups and aggregate them at the root node to finally identify true hot items. The Pegasus system proposed in Gangam et al. (2013) uses the similar idea as Li and Lee (2008). The problem with these methods is that they need each node to maintain all the local items and their exact values and will go through these data multiple times, which involves large space and time overhead locally.

3.2.3 Group Testing

Group testing techniques have been used to identify hot items in data streams. The sketch-based method used in Cormode and Muthukrishnan (2005b) is based on the idea of group testing. It separates all the items into $2k$ groups and in each group it further divides the group into multiple subgroups and does tests on these subgroups to see if it contains a hot item; if there is one and only one hot item in a group, then the test results can be used to recover the ID of this hot item. By separating the item set several times with different separating ways, all the hot items can be identified with very high probability. In this method, when it separates the item set into $2k$ groups, there will be at least k groups that contain none of the hot items and these groups will not be useful for identifying hot items, which is a waste of space. In this chapter, we propose a new

separating scheme to reduce such waste of space, and show how to choose parameters to improve the accuracy and balance space and time according to application requirements.

There are existing non-adaptive group testing algorithms that can be used to identify hot items from a set of items, however the decoding time of these algorithms does not allow them to be practically used. In Indyk et al. (2010) a new decodable non-adaptive group testing algorithm is proposed which significantly decrease the decoding time. However, it still cannot beat the method used in Cormode and Muthukrishnan (2005b) in terms of hot item recovery time.

3.3 Problem Definition

In this section, we define the problem of identifying hot items in multiple distributed data streams. There are N dynamic data streams $S_1, S_2, \ldots, S_N$, and each of them is a sequence of pairs in the following format:

$$(x_1, v_1), (x_2, v_2), \ldots, (x_i, v_i), \ldots$$

where x_i is the identifier (ID) of an item and v_i is an update of the item's value. All the IDs of the items belong to a universe U with size $|U| = m$, and without loss of generality we suppose for any ID $x_i \in [0, m-1]$. Value v_i can be positive or negative integers.

In a single data stream S, the net value $n(x)$ of an item x is defined as the summation of all the updates of values for this item:

$$n(x) = \sum_{(x_i, v_i) \in S \wedge x_i = x} v_i. \tag{3.1}$$

If an item y never appears in the data stream S, then its net value $n(y) = 0$. We assume that for each item which appears in the data stream, its net value is non-negative. The frequency of an item x in data stream S is defined as

$$f(x) = \frac{n(x)}{\sum_{i=0}^{m-1} n(i)}. \tag{3.2}$$

From the definition, we can see that the summation of frequencies of all the items is 1: $\sum_{i=0}^{m-1} f(i) = 1$.

Given a parameter k, an item x in a single data stream S is a hot item if its frequency is larger than the threshold $\frac{1}{k+1}$, that is, $f(x) > \frac{1}{k+1}$. We can see that there are at most k hot items in S. If there are multiple data streams and the updates of values for a certain item could be distributed in these multiple data streams, then we can combine the multiple data streams together to get a single data stream, and the definition of hot items in these multiple data streams will be similar to that in a single data stream.

We want to give a solution to the problem of identifying the IDs of hot items in a single or multiple dynamic data streams with success probability $1 - \delta$, where δ is a pre-selected parameter, with small space, update time and ID recovery time.

Table 3.1 Notations for Hot Item Identification

Notation	Meaning
U	universe of the item's identifier
m	size of the universe U
N	number of data streams
k	parameter of a threshold to determine if an item is hot or not
δ	upper bound of the fail probability of recovering all the hot items , between 0 and 1
L	number of layers(subgroups) used in the algorithm
T	number of hash functions used in the algorithm
d	tunable positive parameter to determine the value of L and T

3.4 Our Approach

We will first present our approach to identify hot items in a single data stream, and then extend it to identify hot items in multiple data streams.

3.4.1 Identify One Hot Item

From paper Cormode and Muthukrishnan (2005b) , we know that if we have a set of items where the frequency of one item is larger than the total frequency of other items,

then it will be very easy to identify this hot item using $O(\log m)$ counters with one-pass algorithm.

The idea of this method is as following. We use a counter C_0 to calculate the total values of all the items in the set. We divide the item set into $\log m$ subsets, and for each subset we use a counter C_i to maintain the summation of the values of the items in this subset. For each item i, it is assigned to the j_{th} subset if its j_{th} bit in its binary representation is 1. In this way, for each subset we can determine if it contains a hot item or not by comparing C_i with C_0: if $C_i \geq \frac{C_0}{2}$ then it contains a hot item, since the total frequency of all the items except the hot item is smaller than the frequency of the hot item; otherwise not. By going through all the $O(\log m)$ counters, we can recover each bit of the hot item and get the ID of the hot item.

3.4.2 Identify Multiple Hot Items

3.4.2.1 Basic Idea

The basic idea to identify multiple hot items is to divide the whole item set into L subgroups and at the same time we have to make sure the following event ξ_1 happen

Event 1. *for each hot item hot(i), there exists at least one subgroup g satisfying the condition that the total frequency of the items except hot(i) in g is smaller than the frequency threshold $\frac{1}{k+1}$.*

The condition in event ξ_1 described above indicates that only one of the hot items, that is $hot(i)$, is in subgroup g. If we can realize the above dividing procedure, then for any hot item $hot(i)$ we can always find a subgroup that contains only one hot item $hot(i)$, and we can identify it using the method given in 3.4.1.

An intuitive idea to realize the dividing procedure is to use a hash function that randomly and uniformly assigns an item to one of the L subgroups. We hope that all the hot items can be mapped to different subgroups, and there is no collision of hot items in

each subgroup, that is, no two hot items are assigned to the same subgroup. If we know as a prior the IDs of the hot items, then we can design a hash function that will generate no collision; however we do not know. So there is possibility that two hot items will be mapped to the same subgroup. From the balls-and-bins model, we know that the larger L is, the smaller the possibility of collision is. Of course, L should be no smaller than k according to the Pigeonhole principle, where k is the upper bound of the number of hot items.

We also notice that when $L > k$, if we only use one hash function, there would be subgroups that do not contain any of the hot items, and these subgroups will be wasted. In order to utilize these wasted subgroups, we can use more hash functions to divide the item set repeatedly. We can use $T(T > 1)$ different hash functions and assign each item to T subgroups. In this case, we hope that two hot items which are mapped to a same subgroup under a hash function will be mapped to other two different subgroups under another hash function. On the other hand, when using T hash functions, there is also possibility that two hot items would be mapped to a same subgroup by two different hash functions.

Based on the above observations, we have to consider how to select the parameters L and T, to make the event ξ_1 happen with a high probability $1 - \delta$ and at the same time keep L and T small. We will use theoretical analysis to show how to choose L and T in next section.

3.4.3 Detailed Algorithm

The pseudo code of our algorithm is given in figure 3.1,3.2,3.3,3.4. The detailed description of the algorithm is given below.

Initialize Data Structure. We use L subgroups and each subgroup is represented by a one-dimensional array of $\log m + 1$ counters. These counters are all initialized to 0. We independently pick T hash functions from the pair-wise independent hash function

1: INITIALIZATION():
2: Set $L \leftarrow \frac{2^d}{d} k \log \frac{k}{\delta}$, $T \leftarrow \frac{1}{d} \log \frac{k}{\delta}$.
3: Set counters $C[0, \ldots, L-1][0, \ldots, \log m] \leftarrow 0$.
4: Randomly select T hash functions $h[1, ..., T]$ from pair-wise independent hash function family $H_2(P, L)$.
5: Set $sum \leftarrow 0$.
6: Set $\mathcal{X} \leftarrow \emptyset$.

Figure 3.1 Pseudo code for initializing the algorithm. d, k, δ are all pre-selected parameters.

1: UPDATEITEM(x, v):
2: Set $sum \leftarrow sum + v$.
3: **for** $i = 1$ to T **do**
4: Set $g \leftarrow h[i](x)$.
5: $C[g][0] \leftarrow C[g][0] + v$.
6: **for** $j = 1$ to $\log m$ **do**
7: **if** j_{th} bit of x is 1 **then**
8: Set $C[g][j] \leftarrow C[g][j] + v$.
9: **end if**
10: **end for**
11: **end for**

Figure 3.2 Pseudo code for updating an item.

```
1: RECOVER():
2: Set θ = sum/(k+1).
3: for i = 1 to L do
4:    if C[i][0] > θ then
5:       Set x ← 0, r ← 1.
6:       for j = 1 to log m do
7:          if C[i][j] > θ then
8:             if C[i][0] − C[i][j] > θ then
9:                Continue to next i
10:            end if
11:            x ← x + r
12:         end if
13:         r ← r × 2.
14:      end for
15:      Add x to X.
16:   end if
17: end for
```

Figure 3.3 Pseudo code for recovering hot items.

```
1: REMOVEFP():
2: for all x ∈ X do
3:    for i = 1 to T do
4:       if C[h[i](x)][0] ≤ θ then
5:          Delete x from X.
6:       end if
7:    end for
8: end for
```

Figure 3.4 Pseudo code for removing false positives.

family $H_2(P, L)$ Carter and Wegman (1979). Each hash function in $H_2(P, L)$ takes the form, $h(x) = (ax + b) \bmod P \bmod L$, where P is a large prime number larger than L, and a, b are randomly picked positive numbers smaller than P which guarantee the randomness of the hash function. We use a counter sum to maintain the total values of all the items' updates, which is initialized to 0. The hot item candidate set $\mathcal{X}$ is initialized to empty.

Update An Item. When a new update (x, v) arrives, we first add the value v to the total count sum. Then we will use the T hash functions to find T subgroups for x which corresponds to T one-dim arrays of counters. In each one-dim array, the first counter calculates the summations of values of the items in this subgroup; the remaining $\log m$ counters calculates the summations of values of items in this subgroup whose corresponding bit is 1.

Recover Hot Items. In the hot item recovery procedure, we first calculate the threshold value $\theta = \frac{sum}{k+1}$, which means that any item whose total updates of values is larger than θ is a hot item. To recover every possible hot item, we go through the subgroups one by one. For each subgroup, if its first counter is larger than θ then it may contain one hot item or more; otherwise we will skip this subgroup. When a subgroup may contain one or more hot items, we will check further if it contains only one hot item or more than one, and at the same time recover the hot item ID during the check process using method in subsection 3.4.1. The check process for a certain subgroup g is as follows: if there exists a counter $C[g][i] > \theta$ and $C[g][0] - C[g][i] > \theta$, then there may be two or more hot items in this subgroup, and we will not use this subgroup to recover hot items.

Remove False Positives. There is possibility that in a subgroup which contains only one of the hot items, the total frequency of the items except the hot item in this subgroup is larger than $\frac{1}{k+1}$. If this case happens, then a counter in the subgroup may be falsely set to a value larger than the threshold value θ and the recovered hot item ID

would be wrong. In order to remove such false positives, for each recovered hot item ID, we would find the T subgroups which it has been assigned to and check the first counter in each subgroup. If all the T first counters are larger than θ, then we will report it as a hot item; otherwise, we will not report it. Note that, after this false positive removing procedure, there may still be some false positives in the reported hot items.

3.4.4 Distributed Algorithm

The algorithm described above is for identifying hot items in a single data stream. When the value updates of an item is distributed in multiple data streams, the naive method to identify hot items in these data streams is to combine the data stream together into a single data stream and then use the algorithm above to go through this newly generated single data stream just once and identify hot items in it. This method involves large communication overhead if the multiple data streams are physically separated. Observing that each instance of our data structure is a sketch of the data stream, merging two instances of our data structure to create a new data structure will have the same effect as merging two data streams into a single stream and create an instance of our data structure on the single stream. The merging is correct given that

- the sizes of the data structures are the same and the hash functions used for each data structure are the same;
- the counters at the same positions of the data structures are added together to generate the counter in the new data structure at the same position.

3.5 Theoretical Analysis

3.5.1 How to choose T and L

Suppose a hot item is assigned to a subgroup, we are interested in the distribution of the summation of frequencies of other items that are also assigned to this subgroup by

all of the T hash functions, which could be modelled by the following random variables. We define a random variable $X^g_{i,h}$ as following:

$$X^g_{i,h} = \begin{cases} f(i) & , \quad \text{if } i \text{ is assigned to } g \text{ by } h \\ 0 & , \quad \text{otherwise} \end{cases} \tag{3.3}$$

where $i \in [0, m-1]$ is an item, $h \in [1, T]$ is the h_{th} hash function and $g \in [0, L-1]$ is the g_{th} subgroup.

We assume each hash function will assign each item to one of the L subgroups with uniform probability $\frac{1}{L}$, so we can know $Pr(X^g_{i,h} = f(i)) = \frac{1}{L}$ and $Pr(X^g_{i,h} = 0) = 1 - \frac{1}{L}$. The expectation of $X^g_{i,h}$ would be

$$E[X^g_{i,h}] = \frac{f(i)}{L} \tag{3.4}$$

Now consider a hot item $hot(j)$ is assigned to the g_{th} subgroup by one of the T hash functions, we define

$$F^g_{i \neq hot(j)} = \sum_{h=1}^{T} (\sum_{i \neq hot(j)} X^g_{i,h}) \tag{3.5}$$

then $F^g_{i \neq hot(j)}$ would be the total frequency of all the items in g_{th} subgroup except $hot(j)$. Let c be a constant value and $c > 1$. According to Markov Inequality, we would have

$$Pr(F^g_{i \neq hot(j)} > cE[F^g_{i \neq hot(j)}]) < \frac{1}{c}. \tag{3.6}$$

If we have

$$E[F^g_{i \neq hot(j)}] < \frac{1}{c(k+1)} \tag{3.7}$$

then we will have

$$Pr(F^g_{i \neq hot(j)} > \frac{1}{k+1}) < \frac{1}{c} \tag{3.8}$$

We can calculate the expectation of $F^g_{i \neq hot(j)}$ as following

$$E[F^g_{i \neq hot(j)}] = E[\sum_{h=1}^{T} (\sum_{i \neq hot(j)} X^g_{i,h})] = \sum_{h=1}^{T} \sum_{i \neq hot(j)} E[X^g_{i,h}] \tag{3.9}$$

$$= \sum_{h=1}^{T} \sum_{i \neq hot(j)} \frac{f(i)}{L} = \frac{T}{L} \sum_{i \neq hot(j)} f(i) < \frac{T}{L}(1 - \frac{1}{k+1}).$$

Let $\frac{T}{L}(1-\frac{1}{k+1}) \leq \frac{1}{c(k+1)}$, which means

$$\frac{T}{L} \leq \frac{1}{ck} \tag{3.10}$$

then we will have inequality 3.8. Notice that inequality 3.8 says that for a hot item that is assigned to a certain subgroup by one of the hash functions, the probability of that the summation of frequencies of all the other items in this subgroup is larger than the threshold $\frac{1}{k+1}$ is smaller than $\frac{1}{c}$, which means that the probability of fail on identifying a hot item under one of the hash functions is smaller than $\frac{1}{c}$.

Since we independently choose the T hash functions, the probability of fail on identifying a hot item under T hash functions is smaller than $(\frac{1}{c})^T$. By union bound, the probability of fail on identifying any of the k hot items is smaller than $k(\frac{1}{c})^T$. So the probability of successfully identifying all the hot items is larger than $1-k(\frac{1}{c})^T$.

We are interested in how to set the value of L and T such that the fail bound δ can be achieved and at the same time the space and time used are small. So let $k(\frac{1}{c})^T = \delta$, then $\frac{k}{\delta} = c^T$. Let $c = 2^d$ and $d > 0$, we can get $T = \frac{1}{d}\log\frac{k}{\delta}$.

From inequality 3.10, we know that $L \geq ckT$, so we can get $L \geq \frac{2^d}{d}k\log\frac{k}{\delta}$. Since L is the number of subgroups we have to maintain and the smaller L is the smaller space we have to use, we will let $L = \frac{2^d}{d}k\log\frac{k}{\delta}$.

Lemma 1. *The probability that our algorithm can identify all the hot items (up to k) by using $T = \frac{1}{d}\log\frac{k}{\delta}$ hash functions and $L = \frac{2^d}{d}k\log\frac{k}{\delta}$ subgroups, where $d > 0$, is at least $1-\delta$.*

Proof. When there are exactly k hot items, then the correctness of this lemma can be easily deduced from the analysis in this section. When there are less than k hot items, the union bound will give us a even smaller probability of fail on identifying any of the hot items, which means that the success probability of identifying all the hot items is even higher than $1-\delta$. □

Now we have $T = \frac{1}{d}\log\frac{k}{\delta}$ and $L = \frac{2^d}{d}k\log\frac{k}{\delta}$, given that k and δ are pre-selected parameters, we can tune the value of d to choose different pairs of T and L. We want both T and L as small as possible since T decides how many hash functions to use which determines the time cost and L decides how many subgroups to maintain which determines the space cost. Observing that T decreases as d increases, while L first decreases and then increases as d increases, there is a trade-off between the time cost and space cost.

The method in paper Cormode and Muthukrishnan (2005b) uses $\log\frac{k}{\delta}$ hash functions and $2k\log\frac{k}{\delta}$ subgroups, which corresponds to $d = 1$ in our method. However, if we set $d = 2$, we can use $T = \frac{1}{2}\log\frac{k}{\delta}$ hash functions and still $L = 2k\log\frac{k}{\delta}$ subgroups to achieve the same success bound, which is an improvement on the time cost.

3.5.2 False Positives

Lemma 2. *The expected number of false hot items that are generated by our algorithm is smaller than* $\frac{k}{d}\log\frac{k}{\delta}$.

Proof. Inequality 3.8 says that the probability of generating a false positive in a subgroup which contains at least one hot item is less than $\frac{1}{c}$, where $c > 1$. Define a random variable Y_i, $i \in [1, L]$ as following:

$$Y_i = \begin{cases} 1 & , \quad \text{if subgroup } i \text{ generates a false hot item} \\ 0 & , \quad \text{otherwise} \end{cases} \tag{3.11}$$

We separate the subgroups into two sets: one set denoted as G_{hot} which contains all the subgroups that contain at least one hot item in it; the other set is denoted as G_{nhot} which contains all the subgroups that do not contain any hot items. For a subgroup $i \in G_{hot}$, we have $Pr(Y_i = 1) < \frac{1}{c}$, $Pr(Y_i = 0) > 1 - \frac{1}{c}$. For a subgroup $j \in G_{nhot}$, denote $F^j_{x\neq hot}$ be the total frequency of all the items in the j subgroup, then similar as equation 3.5,

we have

$$F^j_{x\neq hot} = \frac{T}{L}\sum_{x\neq hot} f(x) \tag{3.12}$$

Suppose we have at least one hot item in the data set, then we will have the expectation of $F^j_{x\neq hot}$ the same as inequality 3.9, which will give us the same result that $Pr(Y_j = 1) < \frac{1}{c}$, $Pr(Y_j = 0) > 1 - \frac{1}{c}$.

Let random variable $Y = \sum_{i\in[1,L]} Y_i$, then Y is the number of false positives that are generated by our algorithm. We have

$$E[Y] = E[\sum_{i\in[1,L]} Y_i] = \sum_{i\in[1,L]} E[Y_i] < \frac{L}{c} = \frac{k}{d}\log\frac{k}{\delta} \tag{3.13}$$

So the expected number of false positives is fewer than $\frac{k}{d}\log\frac{k}{\delta}$. □

The method in Cormode and Muthukrishnan (2005b) is an variation of our algorithm under the case $d = 1$. We can see that under such condition, if we let $d > 1$ in our algorithm, then we will generate fewer false positives. An intuitive explanation is as following. Suppose there are totally M different item IDs, in Cormode and Muthukrishnan (2005b) they assign all the items to $2k$ subgroups and in each subgroup there are expected $\frac{M}{2k}$ distinct items. While in our algorithm, we assign M distinct items into $L = \frac{2^d}{d}k\log\frac{k}{\delta}$ subgroups. Although we will have T different assignments of M distinct items, the expected items in each subgroup is $\frac{MT}{L} = \frac{M}{2^d k}$. So if $d > 1$, we will have smaller number of items in a subgroup. If the number of items in a subgroup is small, then the probability of that the total frequency of the items in the subgroup is larger than the threshold is smaller, so the probability of generating a false hot item is smaller.

3.5.3 Space and Time

Lemma 3. *The space needed is* $O(\frac{2^d}{d}k\log\frac{k}{\delta}\log m)$ *machine words. The update time is* $O(\frac{1}{d}\log\frac{k}{\delta}\log m)$. *The recovery time including the false positive removing procedure is* $O(\frac{2^d}{d}k\log\frac{k}{\delta}(\log m + \frac{1}{d}\log\frac{k}{\delta}))$.

Proof. Since we maintain $L = \frac{2^d}{d} k \log \frac{k}{\delta}$ arrays and each array is composed of $\frac{1}{d} \log \frac{k}{\delta} + 1$ counters. We assume each counter consumes a machine word, then we will need $\frac{2^d}{d} k \log \frac{k}{\delta} \log m$ machines words for all the counters. We additionally maintains a *sum* counter and $2T$ values for $T = \frac{1}{d} \log \frac{k}{\delta}$ hash functions, so the total space we need is $O(\frac{2^d}{d} k \log \frac{k}{\delta} \log m)$.

Suppose the calculation of a hash value needs $O(1)$ time and a counter update needs $O(1)$ time. When updating an (item,value) pair, we need to calculate T hash values and update upto $T(\log m + 1) + 1$ counters, so the total update time is $O(\frac{1}{d} \log \frac{k}{\delta} \log m)$.

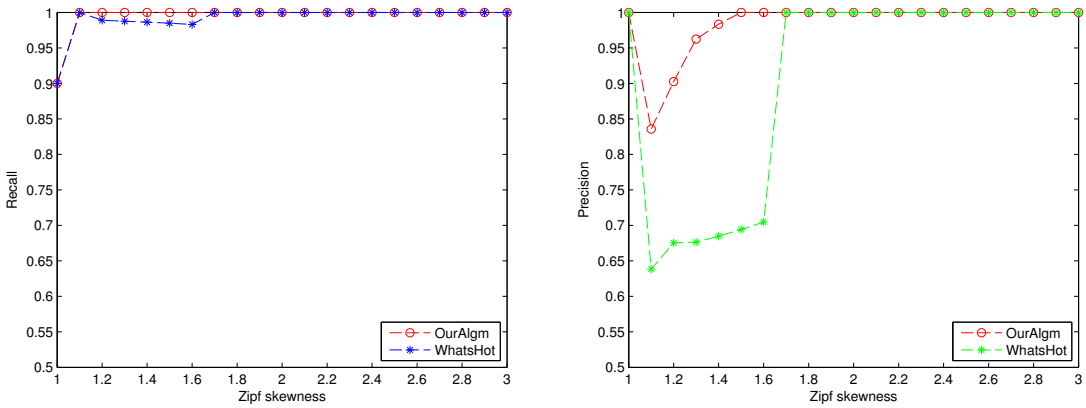
During the hot item recovery procedure, we need to go through all the L subgroups and check each counter in each subgroup, so we need $L \log m$ time. During the false positive removing procedure, we will verify up to L candidate hot items and each verification will calculate T hash values and check T counters, so this step needs $O(2TL)$ time. Adding them together, we need $O(\frac{2^d}{d} k \log \frac{k}{\delta} (\log m + \frac{1}{d} \log \frac{k}{\delta}))$ time for recovering hot items. □

3.6 Performance Evaluation

In order to see the performance of our algorithm and the comparison with the algorithm in Cormode and Muthukrishnan (2005b), we implemented both algorithms in C++. For simplicity, from now on we will refer to our algorithm as OurAlgm and their algorithm as WhatsHot. The evaluations are in terms of recall, precision and average update time. The definitions of the three terms are as following.

- **Recall**: proportion of true hot items that are identified. This is related to false negative. The smaller the number of false negatives is, the higher the recall is and the better the algorithm is. Recall is always between 0 and 1, including both sides.

- **Precision**: proportion of the hot items identified by the algorithm that are true hot items. This is related to false positives. The smaller the number of false positives



(a) Recall on synthetic data

(b) Precision on synthetic data

Figure 3.5 Comparing recall and precision of WhatsHot and OurAlgm using same space but different numbers of hash functions. Synthetic data are used.

is, the higher the precision is and the better the algorithm is. Precision is always between 0 and 1, including both sides.

- **Average Update Time**: the average time used for updating a new item into the data structure of the algorithm. The smaller it is, the better the algorithm is.

Synthetic Data Sets. We generated a group of data sets and each data set contains 100000 items drawn from zipf distribution, using the R package zipfR. We change the skewness of the zipf distributions to generate different data sets. The higher the skewness is, the fewer hot items there are in the data set; and vice versa. The skewness of the zipf distribution is given by a decimal which is no smaller than 1, and when skewness equals 1 the distribution is uniform.

3.6.1 Algorithm Comparison

As the analysis in previous sections indicated, given that k and δ is pre-selected and fixed, OurAlgm will generate less false positives than WhatsHot when we use same number of groups/layers (same space) as but half the number of hash functions of WhatsHot. At the same time, our algorithm can maintain the probability of identifying all

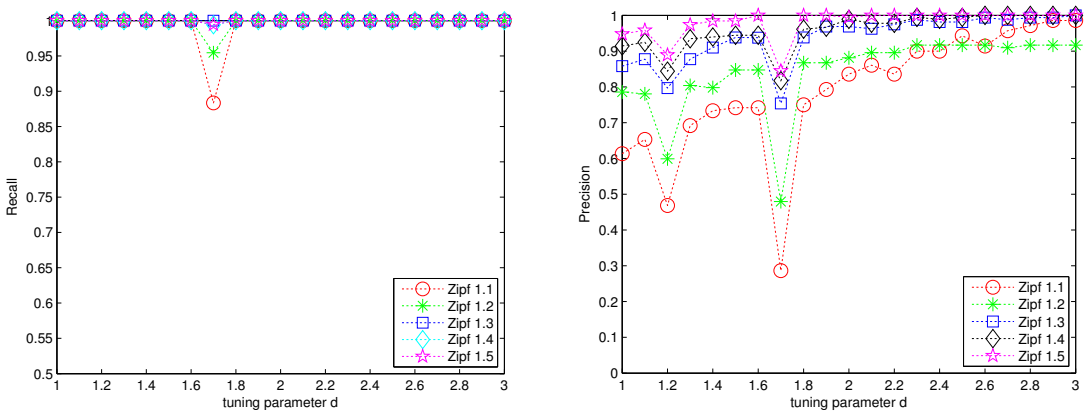
Table 3.2 Time comparison between WhatsHot and OurAlgm. $k = 99$,$\delta = 0.01$,$d = 2$.

Algorithm	AverageUpdateTime(microseconds)
WhatsHot	29
OurAlgm	21

the hot items. And by using less hash functions, our algorithm should perform faster than WhatsHot.

In this part, we fix $k = 99$, $\delta = 0.01$ for both WhatsHot and OurAlgm, and set $d = 2$ for OurAlgm. In this case, the two algorithms will use same space, while OurAlgm will use half the number of hash functions of WhatsHot. We run the two algorithms over the synthetic data sets of different zipf skewness. For each data set, we run each algorithm 10 times and calculate the average recall, precision and update time. The experiment results are shown in figure 3.5 and table 3.2.

We can see from figure 3.5 that given the same space, OurAlgm maintains recall at 1 under different zipf distributions, while WhatsHot has lower recall when the distribution is less skewed. Also, OurAlgm has higher precision than WhatsHot especially when the distribution is less skewed. This makes sense because when the number of groups is fixed, using more hash functions will make the probability of two hot items assigned to a same group higher, and thus the probability to identify all the hot items lower. And when the data is less skewed, the frequencies of the hot items and non-hot items are both close to the threshold $\frac{1}{k+1}$, which makes it harder to identify a true hot item in a group correctly: a small number of non-hot items in the same group will generate a false positive and decrease the precision. By using less hash functions, OurAlgm will assign fewer items into a same group and make the probability of generating false positives smaller and precision higher. Table 3.2 shows that given same space OurAlgm runs faster than WhatsHot when updating the items. Combining with the figure, we could say that OurAlgm performs better than WhatsHot in terms of recall, precision and average update time.



(a) Recall under different d

(b) Precision under different d

Figure 3.6 Recall and precision of OurAlgm with different parameter d which determines the space and time requirements. Synthetic data are used for testing.

3.6.2 Tuning Parameter d

We run OurAlgm over the synthetic data sets under different settings of parameter d ($d > 0$) to see if the recall can always keep high and how the precision will change with d. The results are shown in figure 3.6. Sub-figure 3.6a shows that OurAlgm will always have recall near 1 while d changes. This follows the proof in section 3.5 that as long as L and T satisfies $L > kT$, OurAlgm will have high probability to identify all the hot items. Sub-figure 3.6b shows that the precision of OurAlgm will generally increase as d increases. This makes sense that when d is larger, L is larger and T is smaller which means that the number of items assigned to a same group is smaller. This makes the probability of generating a false positive in a group smaller.

From the definition of $L = \frac{2^d}{d} k \log \frac{k}{\delta}$ and $T = \frac{1}{d} \log \frac{k}{\delta}$, we can see that L takes smallest value when $d = \frac{1}{\ln 2}$ while T decreases as d increases. Since L determines the space to use and T determines the time to use, if our algorithm is used in practice, we would like to choose d neither too large nor too small. And if there is a preference on space or time, then d could be tuned to fit to the preference. We plot the average update time and number of layers/subgroups used for different values of d and show them in figure 3.7.

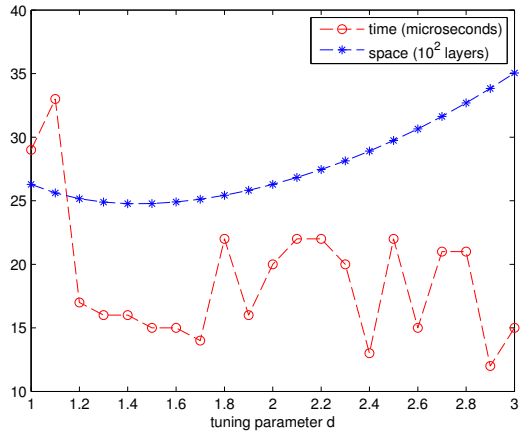


Figure 3.7 Average update time and number of layers(subgroups) used for different d. Synthetic data are used for testing.

Combining figures 3.6 and figure 3.7, we can see that setting d between 2 and 3 would be a good choice for balancing time and space while keep high recall and precision.

3.7 Conclusion

In this chapter, we have studied the problem of how to identify hot items in single or multiple dynamic data streams which is a variant of hot items identification. We've shown that most of the existing methods for hot item identification do not work for the problem formalized in this chapter. Our proposed method improves upon previous method and uses Group Testing based sketching algorithm to identify hot items in single or multiple dynamic data streams. Both theoretical analysis and experimental evaluations show that our algorithm can provide higher recall and precision for identifying hot items than the previous method.

CHAPTER 4. DISTINCT ELEMENT COUNTING

4.1 Introduction

Distinct element counting problem has been found useful in many network monitoring applications to detect different kinds of cyber attacks or anomalies, such as DDoS attacks, worm spreading, spam email delivery, botnet takeover and malicious domains. When researching on the problem of detecting server outages, we came across the distinct element counting problem again. The case is that when a server outage happens, the users' TCP connection setup requests fail at the handshake phase at an abnormally high rate because the server is unable to respond to the users' TCP connection requests. Network monitoring systems can count the number of unsuccessful TCP connection requests to detect a server outage Padmanabhan et al. (2006)Glatz and Dimitropoulos (2012). An unsuccessful TCP connection request is one that the user sends a TCP SYN to the server but does not receive the TCP SYN-ACK from the server, which leads to that the three-way handshake cannot complete.

In the server outage detection case, a user-sent SYN can be modelled as an insertion of a distinct element and a server-sent SYN-ACK can be modelled as a deletion of a distinct element. The number of distinct SYNs with no corresponding SYN-ACKs would be equal to the number of unsuccessful TCP connection requests, which can be modelled as the number of remaining distinct elements. A user can repeatedly send TCP connection requests to the server until s/he gets response from the server or gives up, which correspond to one or more (repeated) insertions of a distinct element. A user's

successful TCP connection with the server may drop because of server outage and the user may try to reconnect to the server which corresponds to a re-insertion of a distinct element. In such a case, the distinct element counting problem solved in this context is different from all the existing versions of distinct element counting problems in the following ways:

1. Insertions and deletions of a same distinct element can be observed in two-way (round-trip) streams at a single monitoring point. They may also appear in different streams at two or more different monitoring points.

2. A deletion of a distinct element cancels out all the previous insertions of this element.

3. A distinct element can be re-inserted after it has been deleted.

And the goal of the distinct element counting problem in this case is to count the number of remaining distinct elements that were inserted but have not been deleted, and further calculate the ratio of such distinct elements over all appeared distinct elements. To our best knowledge, this is the first formulated distinct element counting problem of this type, which is different from all existing types of distinct element counting problems in the literature.

There have been a lot of work done to solve different versions of the distinct element counting problem, but none of these solutions can solve the problem described here. For example, LogLog Durand and Flajolet (2003), HyperLogLog Flajolet et al. (2007), Balls-BinsKane et al. (2010) works on insertion-only data streams, and solutions in Ganguly et al. (2004) Ganguly (2007) Gemulla et al. (2008) Kane et al. (2010) works on the Hamming norm estimation problem which measures the number of insertions and deletions of a distinct element to decide whether to count it. We will discuss and compare these solutions in the related work section.

In this chapter, we formulate a new type of distinct element counting problem that none of existing solutions work. We will present a probabilistic estimation method to solve the distinct element counting problem in not only a single data stream but also distributed dynamic streams. We first give a solution for the problem in a single data stream case, in which we use adaptive sampling technique to keep a fixed-maximum-size sample of the distinct elements in the data stream and use this sample to give a $(1 \pm \epsilon)$ estimation with high success probability. We also design a data structure based on Cuckoo hashing Pagh and Rodler (2004) to maintain our sample which can provide expected constant update time of each element as well as constant estimation time using bounded space. Then we extend this solution to solve the distinct counting problem in distributed dynamic data streams. Our extended method works by maintaining our small-space data structure locally for each data stream, sending these data structures to a central monitor, merging them together into a single final data structure at the central monitor, and then estimate the distinct elements using this final data structure. In this way, we save the space and time overhead in the transmission and merging of the original data streams.

The rest of this chapter is organized as following. We will overview some related work in section 4.2. We formally define our problem in section 4.3. We give our solution for single data stream in section 4.4 and our solution for distributed data streams in section 4.5. The theoretical analysis and experimental evaluations are provided in section 4.6 and 4.7 respectively.

4.2 Related Work

Probabilistic counting of the number of distinct elements in a data stream has a long line of research activities since the famous Flajolet-Martin sketch Flajolet and Martin (1985) was proposed in 1985. These solutions work under different assumptions. Most of

them work in the case where only insertions of elements exist in the data stream Flajolet and Martin (1985) Gibbons and Tirthapura (2001) Bar-Yossef et al. (2002) Durand and Flajolet (2003) Cormode et al. (2005) Flajolet et al. (2007) Kane et al. (2010). When there are also deletions of elements in the data stream, the problem of counting distinct elements becomes more difficult to solve.

If there is no re-insertions in the data stream, then a variation of the optimal algorithm Kane et al. (2010) for insertion only data stream would be an optimal solution: we can consider the insertions and deletions of elements as two different data streams, estimate the distinct elements in the two data streams separately using the optimal algorithm, and then subtract the number of distinct elements in the deletion only data stream from the insertion only data stream to get the number of remaining distinct elements. However, since we care about the order of insertions and deletions of a distinct element, this variation of Kane et al. (2010) cannot work for our problem.

Some works solve the distinct counting problem with both insertions and deletions of elements in a data stream but only works on a set of distinct elements Gemulla et al. (2008), which assumes that the insertion or deletion of a distinct element happens at most once. Some works estimate the Hamming norm of a data stream Gibbons (2001) Cormode et al. (2002) Ganguly et al. (2004) Ganguly (2007) Kane et al. (2010), where the number of insertions and deletions of a distinct element will determine whether or not to count this distinct element: if the number of deletions is not equal to the number of insertions, we will not count the distinct element, otherwise we will. These solutions cannot work for our problem, since we do not care about the number of insertions and deletions of a distinct element and we only care about the order of them.

Recently, there is work done to solve the distinct element counting problem in distributed data streams. Good upper and lower bounds are given for distinct element counting in insertion-only data streams in papers Cormode et al. (2011) Woodruff and Zhang (2012). However, as paper Arackaparambil et al. (2009) has shown, for dynamic

data streams with both insertion and deletion of elements, there is no good upper bound for distinct element counting problem, and it gives a lower bound which depends on the size of the input data streams.

The ideas used in previous solutions for different versions of distinct element counting problem can be utilized in our case to help solving the problem. The basic idea is to map the input distinct elements uniformly onto a certain output field and try to maintain a small part of this output field to estimate the number of input distinct elements. There are several ways to choose a small part to maintain. For example, we can choose to maintain the k smallest output values, or we can maintain the group of output values which is about $\frac{1}{2^r}$ of the total number of distinct elements. Such technique can be considered as distinct sampling where the distinct elements are sampled uniformly at a certain sampling rate and the sample set is used to estimate the original set size. The ideal case is that the mapping is one-to-one and no collision happens. But in practice this is impossible except we know the data stream first and create a mapping table for it, which is of course impossible. However, we can use universal hash functions Carter and Wegman (1979) to bound the probability of mapping collisions and still get good estimations.

When we want to maintain a set of elements, it comes to the problem of how to design a data structure and the algorithm such that insertions and deletions of elements would take as little time as possible. The most efficient data structure to solve this problem would be a dictionary where each element can find its position in $O(1)$ time. However, in practice, the implementation of such a dictionary always has the problem of collisions, where two elements may be mapped to the same position in the dictionary. There are many ways to resolve such collisions, such as linear probing, chained hashing, multiple-choice hashing. Among them, Cuckoo hashing Pagh and Rodler (2004) solves the collision problem with constant update time using universal hash functions.

4.3 Problem Definition

We model a single data stream $\mathcal{S}$ as an ordered sequence of insertions and deletions of elements denoted by

$$(x_1, v_1), (x_2, v_2), \ldots, (x_i, v_i), \ldots$$

where x_i is the identifier of the i_{th} element and v_i indicates whether it is an insertion or a deletion of the element. In our problem, all the elements' identifiers take values in a universe of size N, and we suppose $x_i \in [0, N-1]$. Value v_i can be 1 or -1: $v_i = 1$ indicates an insertion, and $v_i = -1$ indicates a deletion which will cancel out all the previous insertions of the element.

The set of all distinct elements that have been inserted into the data stream when the t_{th} update has occurred (measuring point t) is defined as

$$\mathcal{A}^t = \{x | \exists (x_i, v_i) \in \mathcal{S} \colon x_i = x \wedge v_i = 1 \wedge 1 \leq i \leq t\}$$

The size of set $\mathcal{A}^t$ is denoted as $\alpha^t = |\mathcal{A}^t|$.

The set of remaining distinct elements in the data stream $\mathcal{S}$ when the t_{th} update has occurred (measuring point t) would be

$$\begin{aligned}\mathcal{B}^t = \{x | (\exists (x_i, v_i) \in \mathcal{S} \colon x_i = x \wedge v_i = 1 \wedge 1 \leq i \leq t) \\ \wedge (\neg \exists (x_j, v_j) \in \mathcal{S} \colon x_j = x \wedge v_j = -1 \wedge 1 \leq i < j \leq t)\}\end{aligned}$$

So the set $\mathcal{B}^t$ contains all the remaining distinct elements which have been inserted in the data stream but not yet deleted. The size of set $\mathcal{B}^t$ is denoted as $\beta^t = |\mathcal{B}^t|$. We can see that $\mathcal{B}^t \subseteq \mathcal{A}^t$ and $\beta^t \leq \alpha^t$.

Problem Definition: we want to give an (ϵ, δ) estimation $\tilde{\beta}^t$ of β^t such that $|\tilde{\beta}^t - \beta^t| \leq \epsilon\beta^t$ with probability at least $1 - \delta$, where parameters ϵ and δ are pre-selected. We also want to give an (ϵ, δ) estimation on the ratio of $\mathcal{B}^t$ to $\mathcal{A}^t$: $q^t = \frac{\beta^t}{\alpha^t}$.

When there are multiple distributed data streams $\mathcal{S}_1$, $\mathcal{S}_2$, ..., $\mathcal{S}_m$, the problem becomes to measure β^t and $q^t = \frac{\beta^t}{\alpha^t}$ of the data stream $\mathcal{S}'$, where $\mathcal{S}'$ is a union of $\mathcal{S}_1$, $\mathcal{S}_2$,

$\ldots, \mathcal{S}_m$ and the elements in $\mathcal{S}'$ are ordered by their appearing time. In next section, we will give a solution for the problem in a single data stream, and then extend this solution to solve the problem in multiple data streams in a new section following that.

Table 4.1 Notations for Distinct Element Counting

Notation	Meaning
α^t	number of distinct elements appeared in the data stream at measuring point t
β^t	number of remaining distinct elements in the data stream at measuring point t
q^t	ratio of remaining distinct elements to all appeared distinct elements at measuring point t
N	the size of the universe
ϵ	error bound, between 0 and 1
δ	fail probability bound, between 0 and 1
ψ	the maximum size of the distinct elements sample we maintain
$lsb(x)$	position of the least significant bit 1 of a binary value, e.g. $lsb(100_2) = 2, lsb(001_2) = 0$

4.4 Distinct Element Counting over Single Dynamic Data Stream

4.4.1 Basic Idea

The basic idea of our algorithm is as following. At each time point t, the set $\mathcal{A}^t$ can be divided into two sets:

- $\mathcal{B}^t$: distinct elements that were inserted and have not been deleted.
- $\mathcal{A}^t \setminus \mathcal{B}^t$: distinct elements that were inserted but have been deleted.

At each time point, we use a same sampling probability to randomly and uniformly sample from $\mathcal{B}^t$ and $\mathcal{A}^t \setminus \mathcal{B}^t$ respectively, and keep the sampled elements from the two sets together as the sample. Actually, this sampling process has the same effect as

sampling directly from all the distinct elements appeared in the data stream $\mathcal{A}^t$. We keep the sample within a fixed maximum size by decreasing the sampling probability when the sample size increases as the size of $\mathcal{A}^t$ increases. As long as the sample size is large enough and the ratio q^t is larger than the pre-selected threshold q_0 that we want to detect, we can always keep some distinct elements from $\mathcal{B}^t$ in our sample and use these elements to estimate the size of $\mathcal{B}^t$.

Since we do not know how many distinct elements there are in the data stream, we do not know neither the best sampling probability we should use to keep a sample of the fixed maximum size. However, we could use a series of known sampling probabilities of gradually decreasing values to maintain the size of the sample set within the maximum value. In our algorithm, we use sampling probabilities $\frac{1}{2^b}$ where b starts from 0 and increases by 1 each time the sample size exceeds the maximum value.

The reason why we sample distinct elements from both $\mathcal{B}^t$ and $\mathcal{A}^t \setminus \mathcal{B}^t$ is that we do not know whether an element will be deleted or not when it is inserted, that is, we do not know whether it belongs to $\mathcal{B}^t$ or $\mathcal{A}^t \setminus \mathcal{B}^t$ at the insertion time, until the deletion really happens. Another merit we can get from this method is that we can also estimate the ratio q^t since we have the information of distinct samples from both $\mathcal{B}^t$ and $\mathcal{A}^t \setminus \mathcal{B}^t$. To simplify the expressions we omit the time notation t from now on.

4.4.2 Issues To Resolve

4.4.2.1 Adaptive Distinct Sampling

We use a pair-wise independent hash functionCarter and Wegman (1979) h to map each element $x \in [0, N-1]$ randomly and uniformly to an integer value $h(x) \in [0, N-1]$ and use the least significant bit of the hash value $i = lsb(h(x))$ where $i \in [0, \log N]$ to determine whether it will be sampled or not. We define $lsb(0) = \log N$. Given the sampling level b, if $i \geq b$, then we would sample this element; otherwise not. Since each

element will be mapped to $i = lsb(h(x))$ with probability $\frac{1}{2^{i+1}}$, given the sampling level b, we are sampling each element with probability $\frac{1}{2^b}$.

For each sample level $b \in [0, \log N]$, the total sample set is denoted as $\mathcal{A}_b$ and the sample set containing only remaining distinct elements is denoted as $\mathcal{B}_b$. As sample size increases, we increase b gradually. Each time b increases by 1, we can remove implicitly about half of the distinct samples from our data structure and make room for new samples.

4.4.2.2 Sample Set Maintenance

In order to provide quick lookup/insertion/deletion of distinct elements, we use a data structure composed of Cuckoo hashing tables and a binary search tree (BST) to maintain sampled distinct elements. Cuckoo hashing resolves hash collision problem by using two hash tables and mapping each distinct element into one position of each table. Each distinct element is stored in one of the two positions if either of them is available. When inserting an element x into the Cuckoo hashing tables, it first checks its position in the first table, if it is available then it is inserted there; otherwise, it will kick out the previous occupant at the position and insert x there anyway. The kicked-out element will be inserted into its position in the other table, and if collision occurs again, then the similar kick-out method is used. This process continues until all the elements are inserted in one of their two positions or the maximum number of iterations is reached.

In the original Cuckoo Hashing version, if the collision cannot be solved at the end of the iterations, then new hash functions are used to rehash the elements such that no collision will happen. Such rehashing technique will make worst-case insertions very costly. It has been shown in Kirsch et al. (2010) that with a linked list of small constant size, we can handle the collisions in the Cuckoo Hashing tables efficiently. In our implementation, we use binary search tree to maintain all the elements with collisions.

4.4.3 Data Structure and Algorithms

Our data structure contains two hash tables TABLE-1, TABLE-2 and a BST (binary search tree). Each hash table is of size $K = C\psi$, where C is a constant factor and ψ is the size of the distinct sample set we want to maintain. In our algorithm, we set $K = \psi$ which makes our data structure half full in most cases. Each of the two hash tables is associated with a hash function, h_1 and h_2 respectively, which is randomly chosen from the universal pair-wise independent hash function family H_2 Carter and Wegman (1979). Each entry of the hash tables and the BST keeps three values $< x, lsb(h_0(x)), d >$: x is the identifier of the element, $lsb(h_0(x))$ is the least significant bit of the hash value of x and d indicates whether x has been deleted or not: $d = 1$ means it is deleted and $d = 0$ means it is not deleted. For the BST, the nodes in the tree are ordered according to the value of x. Each time we increase b by 1, we will deleted the nodes in BST which contain elements that are not sampled to reduce the size of the BST.

We also maintain two groups of values s_i and r_i where $i \in [0, \log N]$, which are updated when insertions and deletions of elements happen. $s_i = |\mathcal{A}_i| - |\mathcal{A}_{i+1}|$, $s_{\log N} = |\mathcal{A}_{\log N}|$, $s = \Sigma_{i=b}^{\log N} s_i$. $r_i = |\mathcal{B}_i| - |\mathcal{B}_{i+1}|$, $r_{\log N} = |\mathcal{B}_{\log N}|$, $r = \Sigma_{i=b}^{\log N} r_i$. Here, s counts the sampled distinct elements that have been inserted in the data stream at the current sampling level b, while r counts the sampled distinct elements which have not been deleted yet. We maintain these two values such that we do not need to scan the data structure to count the remaining sampled distinct elements at estimation to save time. The values s_i and r_i are easy to maintain, since each insertion/deletion of a sampled element would only change at most one of the values in these two groups: $s_{lsb(h_0(x))}$ and $r_{lsb(h_0(x))}$. The operations on the data structure are shown in Fig. 4.1, 4.2, 4.3, 4.4 respectively.

1: Initialize $K = \psi$ entries as empty $\emptyset$ for T_1 and T_2 respectively, where $\psi = \frac{C_1}{\epsilon^2 q_0}$.
2: Initialize the BST.
3: Pick random hash functions $h_0 \in H_2([1, N], [1, N])$, $h_1, h_2 \in H_2([1, N], [1, K])$.
4: Set $MAXLOOP \leftarrow C_2 \log \psi$.
5: Set $b \leftarrow 0$.
6: Set $s \leftarrow 0$, $s_i \leftarrow 0$ for $i = 0, 1, 2, \ldots, \log N$.
7: Set $r \leftarrow 0$, $r_j \leftarrow 0$ for $j = 0, 1, 2, \ldots, \log N$.

Figure 4.1 Algorithm: initialization. ϵ, q_0, C_1, C_2 are all pre-selected parameters.

4.5 Distinct Element Counting over Distributed Dynamic Data Streams

When the data are physically distributed in two or more streams, which means that the insertions and deletions of a distinct element may be distributed in multiple data streams, one way to solve the distinct counting problem is to merge the multiple data streams together, sort the elements in the merged data stream according to their appearing time and then do the counting. However, if we use this method we have to ship all the distributed data streams to a central monitor and do the time-consuming sorting, which is infeasible.

Our algorithm described in section 4.4 can be extended to solve the distinct counting problem in a distributed environment with small space and time overhead. We have to change our data structure to remember the time information for each distinct element stored in it. We will add an additional counter ts in each entry of the hash tables and the BST to store the time-stamp of the latest update, either an insertion or a deletion. So we will have four counters in each entry: $\langle x, lsb(h_0(x)), d, ts \rangle$. The stored time-stamp ts could be the real time or relative time. Accordingly, the insertion and deletion algorithms will change: we have to update the time-stamp of the stored distinct elements to the time-stamp of each valid update (insertion or deletion).

Another modification in the extended algorithm is that we have to maintain a list of deletion-only elements. This is because the insertions and deletions of a distinct element

Set $x' \leftarrow x$, $y \leftarrow lsb(h_0(x))$, $d \leftarrow 0$, $status \leftarrow 0$.
if $y < b$ **then**
  return.
end if
if $T_1[h_1(x)][0] = x$ and $T_1[h_1(x)][2] = 0$ **then**
  return
else if $T_1[h_1(x)][0] = x$ and $T_1[h_1(x)][2] = 1$ **then**
  $T_1[h_1(x)][2] \leftarrow 0$, $status \leftarrow 1$
else if $T_2[h_2(x)][0] = x$ and $T_2[h_2(x)][2] = 0$ **then**
  return
else if $T_2[h_2(x)][0] = x$ and $T_2[h_2(x)][2] = 1$ **then**
  $T_2[h_2(x)][2] = 0$, $status \leftarrow 1$
else if x is in BST and not deleted **then**
  return
else if x is in BST and is deleted **then**
  set x in BST as not deleted, $status \leftarrow 1$
end if
if $status = 0$ **then**
  for $i = 1$ to $MAXLOOP$ **do**
    $< x', y, d > \leftrightarrow T_1[h_1(x')]$
    if $x' = \emptyset$ or $y < b$ **then**
      break.
    end if
    $< x', y, d > \leftrightarrow T_2[h_2(x')]$
    if $x' = \emptyset$ or $y < b$ **then**
      break.
    end if
  end for
  if $i > MAXLOOP$ **then**
    Insert a new node in BST for x with $d = 0$.
  end if
  $s_y \leftarrow s_y + 1$, $s \leftarrow s + 1$.
end if
$r_y \leftarrow r_y + 1$, $r \leftarrow r + 1$.
if $s > \psi$ **then**
  $s \leftarrow s - s_b, r \leftarrow r - r_b, b \leftarrow b + 1$.
end if

Figure 4.2 Algorithm: Insertion of x.

1: Set $y \leftarrow lsb(h_0(x))$, $status \leftarrow 0$.
2: **if** $y < b$ **then**
3: return.
4: **end if**
5: **if** $T_1[h_1(x)][0] = x$ and $T_1[h_1(x)][2] = 0$ **then**
6: Set $T_1[h_1(x)][2] \leftarrow 1$, $status \leftarrow 1$.
7: **else if** $T_2[h_2(x)][0] = x$ and $T_2[h_2(x)][2] = 0$ **then**
8: Set $T_2[h_2(x)][2] \leftarrow 1$, $status \leftarrow 1$.
9: **else if** x is in BST and is not deleted **then**
10: Set the x node in BST as deleted, $status \leftarrow 1$.
11: **end if**
12: **if** $status = 1$ **then**
13: Set $r_y \leftarrow r_y - 1, r \leftarrow r - 1$.
14: **end if**

Figure 4.3 Algorithm: Deletion of x.

1: Output $\tilde{\beta} = 2^b r, \tilde{q} = \frac{r}{s}$.

Figure 4.4 Algorithm: Estimation.

may be distributed in multiple data streams. For example, there is one and only one insertion of x in data stream S_1, while there is one and only one deletion of x in data stream S_2 that has a fresher time-stamp than the insertion in S_1; then we should not count this x as a remaining distinct element. However, if we use the deletion algorithm of section 4.4, we would not maintain the deletion-only x in S_2 and lose this information. As a result, when merging the two data structures for S_1 and S_2, we would consider that x is inserted but not deleted, and count it as a remaining distinct element, which is wrong. In order to avoid such errors, we have to maintain a list of deletion-only distinct elements, whose lsb value is larger than the sampling level, for each data stream. Such a list can be implemented using hash table or BST, with entries as $\langle x, lsb(h_0(x)), ts \rangle$.

We maintain an instance of the above extended algorithm for each single data stream locally. For all the data streams, we use the same hash functions and the same size data structures. At the time of estimation, we send the data structure maintained for each data stream to the central monitor. At the central monitor, we will merge all the data

structures together to get the final estimation results for the union of the distributed data streams. The merging process is similar as the updating process of our single data stream algorithm except that we have to check the time-stamp and the deletion-only list.

Here is an example of merging two data structures, excluding the deletion-only list. Suppose we want to merge two data structures D_1 and D_2 together and the current sampling levels of them are b_1 and b_2 respectively. Without loss of generality, suppose $b_1 \geq b_2$. We will update all the sampled elements stored in D_2 into D_1. We do not update the sampled distinct elements in D_1 into D_2, because after merging, the sampling level would be at least b_1 since the number of inserted distinct elements in the merged data stream is no smaller than either of the two single data streams. For each of the distinct elements stored in D_2, if it is sampled at level b_1, we will update it into D_1: if it is inserted but not deleted, we will insert it into D_1; if it is inserted and then deleted, we will delete it from D_1. Here, for each current update of x, we will check whether the current update time-stamp is fresher than the latest update time-stamp of x stored in D_1: if yes, then we will update x into D_1 and change the time-stamp accordingly; otherwise, we will ignore this update. At last, we will go through the deletion-only elements in both D_1 and D_2 and delete the corresponding elements.

When there are more than two distributed data streams, we can use merge-sort-like method to merge all the corresponding data structures together into a single data structure. However, the deletion-only elements should be processed at the last step, after all the possible insertions have been processed. This is to make sure that we do not lose any information of deletion.

The final single data structure is the same as the one generated from the single data stream which is merged from the multiple data streams by sorting all the elements by time. The reason is that during the merging process, we utilize the time-stamp information to update the data structure: we will always keep the information of the latest insertion or deletion of a sampled distinct element in the merged data stream: if

the latest update is an insertion or deletion with insertion happening before the deletion, then this latest information is stored in the main data structure (Cuckoo-Hash tables and BST); if the latest update is a deletion while there is no insertion happened before the deletion, then this is stored in the deletion-only list. We can see that the above method will save us the huge costs of data transmission and sorting.

4.6 Theoretical Analysis

4.6.1 Error Bounds

Theorem 1. *Each instance of our estimator gives an* $(1\pm\epsilon)$*-estimation of the number of remaining distinct elements in a data stream with probability larger than half. By running* $O(\log\frac{1}{\delta})$ *instances and taking the median we can achieve* $1-\delta$ *success probability.*

Proof. Since the hash function h_0 is universal, we can get for each $i \in [1, N]$, $Pr(lsb(h_0(i)) \geq b) = \frac{1}{2^b}$. The analysis of this mapping process can be found in part 4.4.2.1. Let $X_{b,i}$ be a random variable with $X_{b,i} = 1$ if $lsb(h_0(i)) \geq b$ and $X_{b,i} = 0$ if $lsb(h_0(i)) < b$, for $i \in [1, N]$, $b \in [0, \log N]$. Then $Pr(X_{b,i} = 1) = \frac{1}{2^b}$. The expectation and variance of $X_{b,i}$ are

$$E[X_{b,i}] = 1 \times Pr(X_{b,i} = 1) + 0 \times Pr(X_{b,i} = 0) = \frac{1}{2^b} \tag{4.1}$$

$$Var[X_{b,i}] = E[X_{b,i}^2] - (E[X_{b,i}])^2 = \frac{1}{2^b} - \frac{1}{2^{2b}} \tag{4.2}$$

We have $\mathcal{B}_b = \{i | i \in \mathcal{B} \wedge lsb(h_0(i)) \geq b\}$ where $b \in [0, \log N]$. Let $\beta = |\mathcal{B}|$, $\beta_b = |\mathcal{B}_b|$. β_b can be considered as a random variable which is the sum of the random variables $X_{b,i}$ where $i \in \mathcal{B}$, that is, $\beta_b = \Sigma_{i\in\mathcal{B}} X_{b,i}$. The expectation and variance of β_b are

$$E[\beta_b] = E[\Sigma_{i\in\mathcal{B}} X_{b,i}] = \Sigma_{i\in\mathcal{B}} E[X_{b,i}] = \frac{\beta}{2^b} \tag{4.3}$$

$$Var[\beta_b] = Var[\Sigma_{i\in\mathcal{B}} X_{b,i}] = \Sigma_{i\in\mathcal{B}} Var[X_{b,i}] = \frac{\beta}{2^b} - \frac{\beta}{2^{2b}} \tag{4.4}$$

The above calculation of variance of β_b is based on the pair-wise independence of the hash function h_0 which guarantees that the covariance of $X_{b,i}$ and $X_{b,j}$ is 0 for different $i, j \in [1, N]$.

By Chebyshev's inequality, we have

$$Pr(\Big|\beta_b - E[\beta_b]\Big| \geq \epsilon E[\beta_b]) \leq \frac{Var[\beta_b]}{\epsilon^2 E[\beta_b]^2} \leq \frac{1}{\epsilon^2 E[\beta_b]} \tag{4.5}$$

We have $\mathcal{A}_b = \{i | i \in \mathcal{A} \wedge lsb(h_0(i)) \geq b\}$ where $b \in [0, \log N]$. Let $\alpha = |\mathcal{A}|$, and $\alpha_b = |\mathcal{A}_b|$, then $\alpha_b' s$ are random variables and have similar properties as $\beta_b' s$. We can consider the data set $\mathcal{A}_b$ as a sample of $\mathcal{A}_{b-1}$ with sampling probability $\frac{1}{2}$. Let the event ξ_1 be that $\alpha_{b-1} = L > \psi$. For each distinct element $j \in \mathcal{A}_{b-1}$, we consider the event ξ_2 that $j \in \mathcal{A}_b$. Let the indicator random variable for event ξ_2 be $Y_{b,j}$, such that $Y_{b,j} = 1$ if $j \in \mathcal{A}_b$ and $Y_{b,j} = 0$ if $j \notin \mathcal{A}_b$, given that $j \in \mathcal{A}_{b-1}$. We have the probability distribution for random variable $Y_{b,j}$ as following

$$Pr(Y_{b,j} = 1) = Pr(j \in \mathcal{A}_b | j \in \mathcal{A}_{b-1}) = \frac{1}{2^b} / \frac{1}{2^{b-1}} = \frac{1}{2}$$

$$Pr(Y_{b,j} = 0) = 1 - Pr(Y_{b,j} = 1) = \frac{1}{2} \tag{4.6}$$

And we have the expectation $E[Y_{b,j}] = \frac{1}{2}$.

Conditioned on event ξ_1, we have expectation for α_b

$$E[\alpha_b] = E[\Sigma_{j \in \mathcal{A}_{b-1}} Y_{b,j}] = \Sigma_{j \in \mathcal{A}_{b-1}} E[Y_{b,j}] = \frac{L}{2} > \frac{\psi}{2} \tag{4.7}$$

We increase b by 1 in our algorithm each time the sample set size exceeds the maximum value ψ. So if current sampling level is b, then for the previous sampling level $b-1$, the event $\xi_1 : \alpha_{b-1} = L > \psi$ happens with probability 1. This means that at each estimation time point

$$E[\alpha_b] = \frac{L}{2} > \frac{\psi}{2} \tag{4.8}$$

Since $\beta = q\alpha$, we have

$$E[\beta_b] = \frac{\beta}{2^b} = \frac{q\alpha}{2^b} = qE[\alpha_b] \tag{4.9}$$

From inequality 4.5, we have

$$Pr(\Big|\beta_b - E[\beta_b]\Big| \geq \epsilon E[\beta_b]) \leq \frac{1}{\epsilon^2 E[\beta_b]} = \frac{1}{\epsilon^2 q E[\alpha_b]} < \frac{1}{\epsilon^2 q \frac{\psi}{2}} = \frac{2}{\epsilon^2 q \frac{C_1}{\epsilon^2 q_0}} \leq \frac{2}{C_1} \tag{4.10}$$

So if we set constant $C_1 \geq 4$, we can get $\tilde{\beta} = 2^b \beta_b = 2^b(1 \pm \epsilon)E[\beta_b] = (1 \pm \epsilon)\beta$ with probability at least $\frac{1}{2}$ for each instance of our algorithm. By running $\log \frac{1}{\delta}$ instances and taking the median we can achieve $1 - \delta$ success probability. Also, we assume that $q > q_0$, where q_0 is a pre-selected parameter, which means that we can give an estimation of the remaining distinct elements in the data stream within the given error bound ϵ with probability at least $1 - \delta$ if the ratio of remaining distinct elements to all the distinct elements in the data stream is not smaller than the threshold q_0.

Next, we will show the error bounds of our estimation $\tilde{q}$ of the remaining distinct element ratio $q = \frac{\beta}{\alpha}$. From inequality 4.10, we know that we will have event-1, $(1-\epsilon)\frac{\beta}{2^b} < \beta_b < (1+\epsilon)\frac{\beta}{2^b}$, happen with probability at least $1 - \frac{2}{C_1}$. We can use the similar technique to show that we will have event-2, $(1-\epsilon\sqrt{q_0})\frac{\alpha}{2^b} < \alpha_b < (1+\epsilon\sqrt{q_0})\frac{\alpha}{2^b}$, happen with probability at least $1 - \frac{2}{C_1}$. We can choose a big enough C_1 value to let event-1 and event-2 happen at the same time with probability at least $\frac{1}{2}$. Since $\tilde{q} = \frac{\beta_b}{\alpha_b} = \frac{(1\pm\epsilon)\beta}{(1\pm\epsilon\sqrt{q_0})\alpha} = \frac{1\pm\epsilon}{1\pm\epsilon\sqrt{q_0}}q$, we will have $\frac{1-\epsilon}{1+\epsilon\sqrt{q_0}}q < \tilde{q} < \frac{1+\epsilon}{1-\epsilon\sqrt{q_0}}q$. Since $0 \leq q_0 \leq 1$ and $0 < \epsilon < 1$, we can choose small values for ϵ such that

$$\begin{aligned} \frac{1+\epsilon}{1-\epsilon\sqrt{q_0}} &= 1 + \frac{1+\sqrt{q_0}}{1-\epsilon\sqrt{q_0}}\epsilon = 1 + O(\epsilon) \\ \frac{1-\epsilon}{1+\epsilon\sqrt{q_0}} &= 1 - \frac{1+\sqrt{q_0}}{1+\epsilon\sqrt{q_0}}\epsilon = 1 - O(\epsilon) \end{aligned} \tag{4.11}$$

Above all, we will have $\tilde{q} = (1 \pm O(\epsilon))q$. □

4.6.2 Space and Time

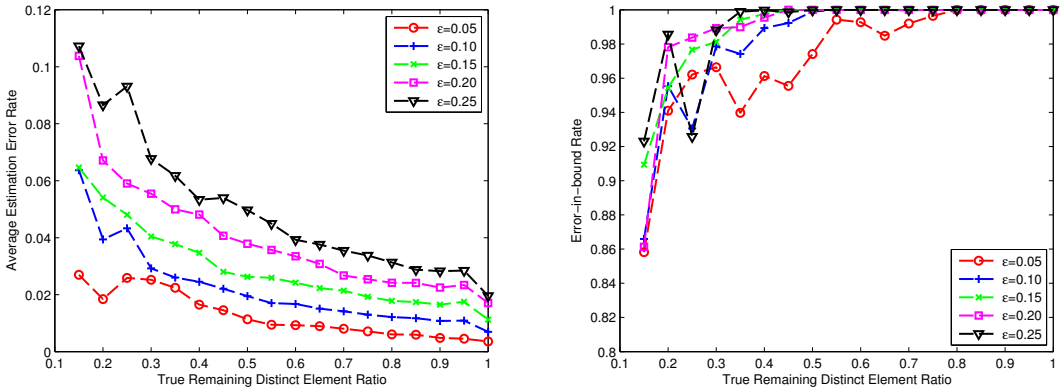
Theorem 2. *Each instance of our algorithm needs* $O(\frac{\log N}{\epsilon^2 q_0})$ *bits space and* $O(1)$ *expected update and estimation time.*

Proof. **Space**. In each instance of our data structure, the Cuckoo hashing tables use $2K = 2(1+\gamma)\frac{C_1}{\epsilon^2 q_0}$ which is $O(\frac{1}{\epsilon^2 q_0})$ entries. From Aumuller et al. (2012), we know that if we use pair-wise independent hash functions for the two Cuckoo hash tables and use an additional linked list of maximum size u, we can preserve the rehashing probability to $\Theta(\frac{1}{\psi^{u+1}})$ for each insertion of a new distinct element. For those elements that cause rehashing, we will maintain them in the BST. Since we always maintain a sample set with size smaller than ψ, we can see that the expected number of distinct elements that can cause rehashing is $\Theta(\frac{1}{\psi^u})$ which is ignorable. So the size of the BST is $O(1)$ if we set u small enough such as 4 which is suggested in Kirsch et al. (2010). For each entry we maintain x, lsb and a deletion indicator bit, so each entry needs $O(\log N)$ bits. Totally, we need $O(\frac{1}{\epsilon^2 q_0}\log N)$ bits for each instance of our algorithm.

Time. Since we are using pair-wise independent hash functions, each calculation of a hash value will take $O(1)$ time Carter and Wegman (1979). For each insertion of a sampled element, we have to check the two positions in the two hash tables by calculating two hash values and search in the BST if needed, so the expected time is $O(1)$. When inserting the sampled element into the data structure, the expected number of iterations is $O(1)$ Pagh and Rodler (2004) Kirsch et al. (2010) Aumuller et al. (2012). Since the expected length of the BST is $O(1)$, the expected time for insertion in BST is also $O(1)$. Totally the expected time for inserting a sampled element into our data structure is $O(1)$. For each deletion of a distinct element, we need the checking process similar as in the insertion algorithm, so the expected time is $O(1)$ too. The estimation time of each instance is $O(1)$ since we only need to do several multiplications and divisions. □

4.7 Performance Evaluation

In this section, we evaluate our algorithm with both synthetic data and real traces. We implement our algorithm using C++ in a single thread. We calculate the estimation



(a) Average estimation error rate

(b) Error-in-bound rate

Figure 4.5 Average error rate and error-in-bound rate of estimations of the remaining distinct element number for different choices of ϵ. Evaluated with synthetic data.

error rate for both remaining distinct element number β and its ratio $q = \frac{\beta}{\alpha}$. Suppose the true value is β and our estimation value is $\tilde{\beta}$, the estimation error rate is defined as $\frac{|\tilde{\beta}-\beta|}{\beta}$. Based on the error rate, we also calculate the error-in-bound rate which is the frequency of successfully getting an error rate within the given error bound ϵ.

4.7.1 Synthetic Data

We generate 50 sets of different synthetic data with Normal distributions or Zipf distributions for this evaluation. Each data set contains $2,000,000$ data items. The IDs of the data items are all integer numbers in $[1, 2^{32}]$, and the values can be 1 or -1 which indicates insertion and deletion respectively. The true α, β, q values for the data sets are all different, while α is always larger than the size ψ of the sample we maintain in our algorithm, and q is always larger than the pre-selected q_0.

4.7.1.1 Different ϵ

In this evaluation, we want to test the accuracy of our algorithm in different settings of ϵ. We fix other parameters as $q_0 = 0.1$, $C_1 = 4$, $C_2 = 3$. We change the value of the

error-bound ϵ to see whether the estimation error rate is within this bound. We run our algorithm on each of the data sets 10 times with different hash functions for each ϵ, and estimate the number of remaining distinct elements at different time points. We measure the average error rate of estimations as well as the rate of successfully bounding the estimation error within the pre-selected parameter ϵ. We grouped the results according to the true value of the remaining distinct elements ratio at the measuring points.

The experiment results are shown in Fig. 4.5. We can see that for each pre-selected error bound ϵ, the estimation errors are within the bound with very high probability (at least 0.85). Even when the remaining distinct elements only takes up 15% of the total (which corresponds to the left-most points), the average estimation errors are very small and the error-in-bound rate, which is larger than 0.85, is much higher than the expected value 0.5 given in theoretical analysis.

4.7.1.2 Size of the BST

We also measure the size of the BST which maintains the elements with collisions to see if it will grow too large to slow down the update time. The result is given in Fig. 4.6. We can see from the figure that the size of the BST increases as ϵ decreases, which makes sense since when ϵ decreases, the hash table size as well as the size of the sample we maintain increases, and we will have more chances of getting collisions. However, the size of the BST is quite small comparing to the size of the hash tables: for different values of ϵ, the BST size is about 2% (or less) of the total size of the two hash tables.

4.7.2 Real Trace

We use a DDoS attack trace from CAIDA DDo () to evaluate our algorithm. The attack makes the server unable to respond to part of the users that are sending TCP requests to it, which leaves a number of unsuccessful TCP connection requests in the traffic. We want to estimate the number of users with unsuccessful connection requests

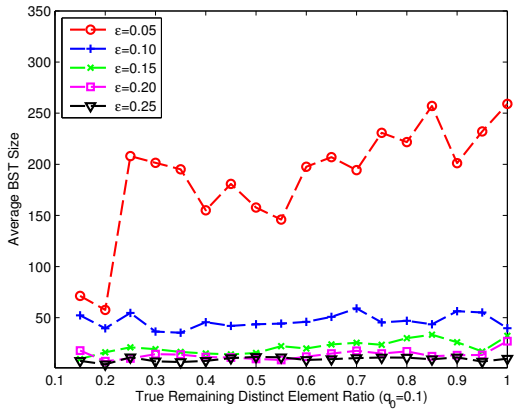
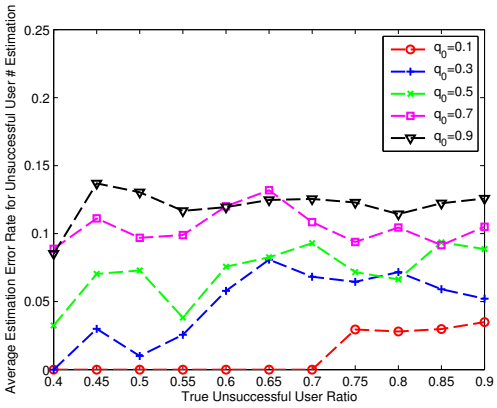


Figure 4.6 Average size of the BST (binary search tree) in our data structure. Evaluated using synthetic data.

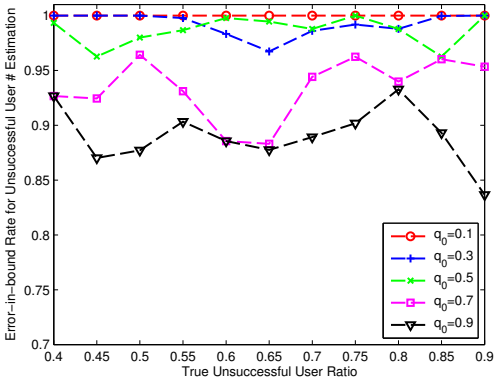
sent to the server (unsuccessful users), and its ratio to the total number of users who send TCP connection requests to the victim server. This data set contains one-hour two-way trace of the DDoS attack. The one-hour trace is split into 5-minutes pcap files and each packet contains the IP headers and transport layer headers.

We construct a data stream for the one-hour trace in the following way: for each packet containing the initial TCP SYN sent to the victim server, we use the source IP address as the ID of the element and 1 as the value; for each packet containing TCP SYN-ACK sent from the victim server, we use destination IP address as the ID of the element and -1 as the value. At each measuring point, the number of remaining distinct element is the number of users with unsuccessful TCP connections, and the total number of distinct elements is the total number of users who send TCP connection requests to the victim server.

We've implemented a distributed version of our algorithm which is described in section 4.5. We evaluate this distributed version using the DDoS2007 trace since it contains timestamp of the packets. We randomly separate the trace into two streams, maintain our extended algorithm on each of the two streams, and finally merge the two data structures to get final estimations. We show here the average error rate and error-in-bound rate



(a) Average error rate for estimating unsuccessful user number (distributed version)



(b) Error-in-bound rate for estimating unsuccessful user number (distributed version)

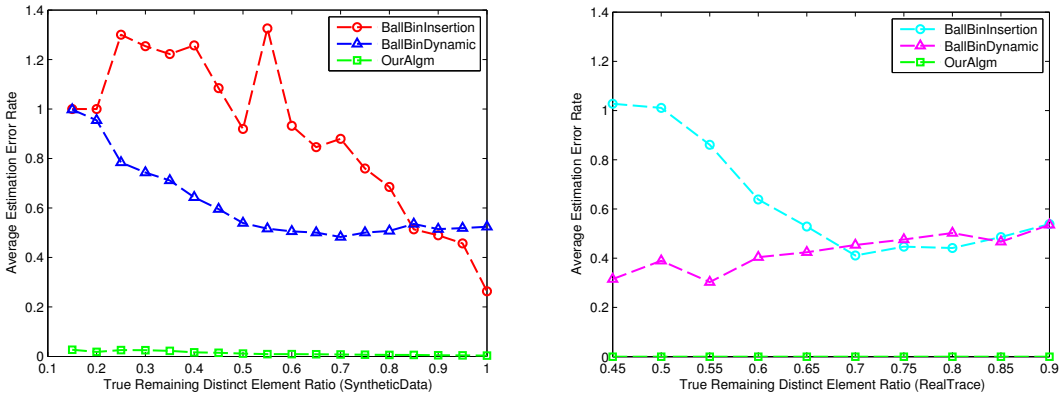
Figure 4.7 Precision of the distributed version of our algorithm for different values of q_0, with $\epsilon = 0.25$. Evaluated with DDoS2007 trace.

for the unsuccessful user number estimations over the two data streams, and present some of the results in Fig. 4.7. We can see that the estimation errors are very small and the error bound can be achieved with very high probabilities. This conforms to our theoretical analysis and verifies that the distributed version of our algorithm also works for our problem.

4.7.3 Comparison of Algorithms

We compare the precision of our algorithm with the insertion-only algorithm in Kane et al. (2010) (referred as BallBinInsertion) and a modified version of Kane et al. (2010) (referred as BallBinDynamic) which can handle deletions of elements in the data stream. We refer to our algorithm as OurAlgm. For BallBinInsertion algorithm, we use two instances of their algorithm to maintain the insertions and deletions of distinct elements respectively, and calculate the remaining distinct elements by subtracting the number of deleted elements from the number of inserted elements.

For BallBinDynamic algorithm, we modify the original algorithm of Kane et al. (2010) by maintaining $\log N$ sampling levels and in each level using $\frac{1}{\epsilon^2}$ counters to count the



(a) Synthetic Data (b) Real Trace

Figure 4.8 Comparing our algorithm with BallBinInsertion and BallBinDynamic algorithms with $\epsilon = 0.05$. Evaluated with both synthetic data and real trace.

number of elements that are sampled in the level and thrown into each counter. When updating the arriving elements, if it is an insertion of element, then we increase the corresponding counter; if it is a deletion of element, we decrease the counter. In this way, we can handle the deletions of elements.

We use both synthetic data and real trace to compare the three algorithms. The evaluation results are shown in Fig. 4.8. It is very clear that both BallBinInsertion and BallBinDynamic algorithms are performing much worse than our algorithm especially when the true remaining ratio is low. This makes sense, since for BallBinInsertion algorithm, the numbers of inserted and deleted elements are both estimated with certain errors, and after subtraction, the estimated number of remaining elements would have larger error. Another reason is that neither of the two algorithms matches the deletions with the corresponding insertions of distinct elements. Both algorithms would underestimate the number of remaining distinct elements when a significant part of the updates in the data stream are deletions.

4.8 Conclusion

In this chapter, we formulate a new type of distinct element counting problem in single and distributed dynamic data streams. We are the first to give a space-and-time-efficient algorithm to solve this new problem which is useful for network monitoring applications. We have given theoretical analysis of our estimation algorithm to show that it can give $(1 \pm \epsilon)$ estimations with high probability, and the space and time required for each instance of our algorithm are $O(\frac{1}{\epsilon^2 q_0} \log N)$ bits and $O(1)$ expected update time per element. We have evaluated our algorithm with synthetic and real data sets to show that our algorithm can give bounded-error estimations with high success probability which conforms to the theoretical analysis.

CHAPTER 5. SUPERSPREADER IDENTIFICATION

5.1 Introduction

Internet Service Providers (ISPs) collect traffic measurements for various purposes like customer accounting and traffic engineering Estan and Varghese (2002), which are also used for traffic anomaly detection Manikopoulos and Papavassiliou (2002), cyber-attack attribution Brenner (2009), network forensic analysis Xie et al. (2006), etc. An important traffic feature of interest is the host cardinality Chen et al. (2009); Cao et al. (2009); Guan et al. (2009), defined as the number of distinct peers that a host communicates with. High-cardinality hosts, known as super-spreaders Venkataraman et al. (2005), are often the signs of many security problems, e.g., (distributed) denial-of-service attacks, spam emails, worm spreading, botnet takeover, etc. For example, a compromised host doing fast scanning for worm propagation often makes an unusually high number of connections in a short time. When a host is infected, it randomly generates destination IP addresses and tries to infect the hosts at these addresses. Thus, high cardinalities are important signs of malware propagation in the network. Many previous works have verified the effectiveness of the cardinality as a primary feature for network security Cao et al. (2009); Guan et al. (2009); Venkataraman et al. (2005); Zhao et al. (2005); Locher (2011).

We study the problem of detecting high-cardinality hosts with the following goals:

- Firstly, the detection of high-cardinality hosts requires a network-wide traffic view. The attack traffic may enter the network from multiple routers. If we only monitor

the cardinality at every single router, the attackers would be missed at any of them. Therefore, we have to merge the traffic measurements from multiple routers, and get a network-wide view of the host cardinalities.

- Secondly, the packets from the same connection must be removed for the cardinality computation. When we merge the traffic measurements from multiple routers, we cannot simply add the cardinality of a host at each router together to calculate its total cardinality. Because each connection may travel multiple routers from the source to the destination, we need to design a method to only count the distinct connections.

- Thirdly, due to the large size of the traffic measurements, ISPs are only able to collect some summaries of the traffic measurements from local routers. We have to design a mergable data structure, referred as a sketch, to reduce the communication costs.

- Lastly, we cannot compute the cardinality for every single host in order to identify the high-cardinality ones. Due to the large size of the IP addresses, we want to only estimate the cardinality for hosts with high-cardinality using limited space and running time.

The above challenges have only been partially addressed in previous work, and there have been no algorithm that can solve all these challenges, to the best of our knowledge. In this chapter, we propose a new data streaming algorithm to compute a mergable Agarwal et al. (2012) and reversible Schweller et al. (2007) sketch, which can be used to identify high-cardinality hosts from network-wide traffic measurements. Our data structure summarizing traffic measurement is designed based on noise group testing Chan et al. (2011), which can identify high-cardinality hosts efficiently in a distributed network monitoring system. Our main idea is that we consider the identification of high-cardinality hosts as a channel-coding problem, which also provides a new theoretical

analysis method for this problem. Our work aims to provide a new scheme for distributed network monitoring, which is much more efficient than the state-of-art solution.

5.2 Related Work

A super-spreader detection algorithm contains two steps: cardinality estimation and host identification. For cardinality estimation sampling or sketching techniques can be used. Sampling technique provides a simple way to keep a subset of the original data to estimate cardinality and at the same time keep information of the identities of the super-spreaders. For example, Venkataraman *et al.* Venkataraman et al. (2005) proposed the first efficient algorithm to identify super-spreaders, which sampled packets from a set of distinct source-destination pairs. Similar sampling method was also used by Cao *et al.* Cao et al. (2009) to identify hosts with moderately large number of connections in the network. However, sampling based algorithm often have problems of space and bias: the space used to keep the sampled data may depend on the original data and not be always bounded; sampled data may also be biased and generate large errors when used to estimate cardinality.

Sketching technique provides a space-efficient way to estimate cardinality. However, many sketching algorithms lack the ability to recover the identities of super-spreaders. For example, Zhao *et al.* Zhao et al. (2005) propose an algorithm combining sketching and sampling techniques for super-spreader identification. Their algorithm first samples the packets and then uses a Bloom Filter, which consists of 2-dimensional bit arrays, to estimate the cardinality. In their method, each packet is hashed by several independent hash functions into different positions in the bit arrays, and the bit on each corresponding position is set to 1. The number of connections at each host can be estimated based on the number of 1s in the bit arrays. Their algorithm is improved by Guan *et al.* Guan et al. (2009) who use Chinese Remainder Theorem to speed up the identification.

A reversible sketch is a summary of the data stream using small space and has the ability to recover the identities of interested items. It may be firstly proposed for frequent item detection Cormode and Muthukrishnan (2005b) which applies *combinatorial group testing* (CGT) to efficiently identify top-k frequent items. The idea is that there are at most k items with a frequency larger than the threshold $\frac{S}{k+1}$ in the data stream, where S is the total frequency of the data stream. If a high-frequency item is put with several items having small frequencies in a same group, then this high-frequency item can be identifies as a majority efficiently by group testing technique. The CGT sketch Cormode and Muthukrishnan (2005b) divides all items into $2k$ groups almost evenly and randomly, and picks out the majority from each group if there exists one. By independently repeating this procedure $\log(k/\delta)$ times, they can guarantee that all top-k frequent items can be picked up as a mority with probability at least $1-\delta$. Schweller *et al.* propose another *Reversible* sketch Schweller et al. (2007) for change detection in network traffic. Both sketches utilize the position information to identify interested hosts in the network. A similar recovery method has also been proposed for traffic anomaly detection with the Principle Component Analysis (PCA) Li et al. (2006) and the aggregate queries in high-speed network Liu et al. (2012).

For cardinality estimation, there has been a lot of work done in the condition of centralized data processing Bar-Yossef et al. (2002) Durand and Flajolet (2003) Flajolet et al. (2007) Kane et al. (2010). Kane, Nelson and Woodruff Kane et al. (2010) have developed an optimal algorithm which can guarantee a half decent $(1\pm\epsilon)$-approximation of cardinality at all time points, using $O(\frac{1}{\epsilon^2}+\log m)$ space, where m is the size of the universe, and $O(1)$ updating and reporting time. In order to provide a small failure probability δ, this algorithm has to be repeated $\log(1/\delta)$ times independently. If this algorithm is used in reversible sketches Schweller et al. (2007); Cormode and Muthukrishnan (2005b); Liu et al. (2012), the space and the running time would be very high for the detection of high-cardinality hosts. Therefore, we propose a more efficient data

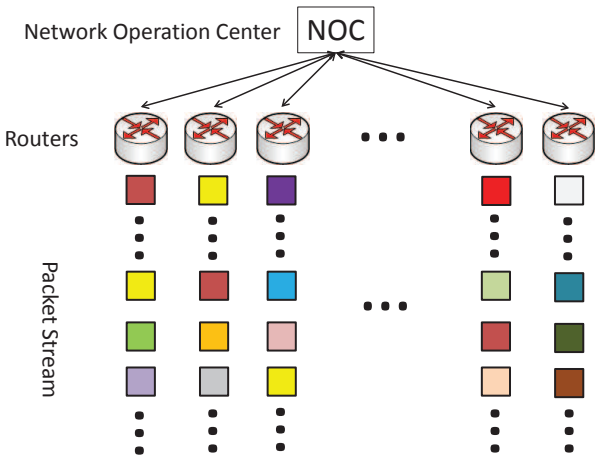


Figure 5.1 Network-wide Traffic Monitoring

structure than trivial solutions, which considers the errors in cardinality estimation.

5.3 Problem Definition

In this chapter, we focus on the identification of high-cardinality hosts from network-wide traffic measurements. Assume there are k routers in the network, each of which monitors a stream of packets as shown in Fig.5.1. At the i-th router for $i = 1, \ldots, k$, there is a packet stream, denoted by

$$(s_{i1}, d_{i1}), (s_{i2}, d_{i2}), \ldots, (s_{it}, d_{it}), \ldots \tag{5.1}$$

where t denotes the current time, and $s_{it}, d_{it} \in \mathcal{U}$ denote the source and the destination IP addresses in a packet, respectively. Let $\mathcal{U}$ denote the set of all the possible IP addresses in the network. And let m denote the size of the universe $\mathcal{U}$: $m = |\mathcal{U}|$. If IPv4 is used, then the size of $\mathcal{U}$ would be 2^{32}.

Let $\mathcal{A}_{it}$ denote the set of packets observed at the i-th router in the current measurement window of length τ,

$$\mathcal{A}_{it} = \{(s_{ij}, d_{ij}) \,|\, j \in [t - \tau, t]\}. \tag{5.2}$$

Table 5.1 Notations for superspreader identification

Notation	Meaning
$\mathcal{U}$	the universe, e.g. set of all possible IP addresses
m	the size of the universe
ϵ	estimation error, $0 < \epsilon < 1$
δ	failure probability for a randomized algorithm, $0 < \delta < 1$
OPT	the optimal cardinality estimation algorithm in Kane et al. (2010)
F_t	sum of the destination cardinalities of all the hosts at time t
$\hat{F}_t$	an estimation of F_t
L	number of layers in our data structure
G	number of groups in each layer of our data structure
κ	number of hash functions used for randomly separating the hosts into subsets
h_j	j^{th} hash function used for separating the hosts into subsets, $1 \leq j \leq \kappa$
$q(s)$	quotient of s divided by L
$W(q(s))$	codeword of $q(s)$ encoded by an error correcting code
$lssb(x)$	position of the least significant bit 1 of x, starting from 0, e.g. $lssb(0100_2) = 2, lssb(0101_2) = 0$
$\oplus$	bit-wise XOR operator

The set of the destination IPs from a host x is denoted by

$$\mathcal{D}_{it}^x = \{d \,|\, (x,d) \in \mathcal{A}_{it}\}. \tag{5.3}$$

We define the destination cardinality D_t^x of a host x at time t as the number of distinct hosts in the union of the sets $\mathcal{D}_{it}^x$,

$$D_t^x \;=\; |\cup_{i=1}^k \mathcal{D}_{it}^x|. \tag{5.4}$$

The high-cardinality hosts are the ones with a large destination cardinality that exceeds a given threshold. Here, we only define the high-destination-cardinality hosts; for high-source-cardinality hosts, the definition is similar which we omit here.

Definition 2. Given a threshold θ, a host $x \in \mathcal{U}$ is identified as a high-cardinality host if

$$D_t^x > \theta. \tag{5.5}$$

Due to the resource limitation, we can only detect the high-cardinality hosts with some approximation errors. Let F_t denote the total number of distinct connections,

$$F_t = |\cup_{i=1}^{k} \mathcal{A}_{it}|. \tag{5.6}$$

It is easy to verify that F_t equals to the sum of the destination cardinalities of all the hosts,

$$F_t = \sum_{x \in \mathcal{U}} D_t^x \tag{5.7}$$

An (ϵ, δ)-approximation algorithm can provide the following error guarantee.

Definition 3. An (ϵ, δ)-approximation algorithm can report any host $x \in \mathcal{U}$ such that

$$D_t^x > \theta + \epsilon F_t \tag{5.8}$$

as a high-cardinality host with a probability at least $1 - \delta$.

Given limited memory, engineers can determine the parameters δ and ϵ to balance the detection error and the missing probability. An optimal selection of δ and ϵ is determined by the distribution of host cardinalities in the network, which is unknown during traffic monitoring. Usually, we choose ϵ small enough to guarantee that $\theta > \epsilon F_t$. To simplify the description, we will omit the subscript t in the rest part of this chapter.

5.4 Our Algorithm

In this section, we describe a new data streaming algorithm for identifying hosts with high destination cardinalities, which is well-known as the super-spreader problem. The problem of high-source-cardinality hosts identification can be solved with the same algorithm by exchanging source and destination IP addresses in our solution. We first introduce our data structure, i.e. sketch, used to summarize traffic measurements. And then, we describe our update algorithm at each local router, and merge algorithm at the NOC. At last, we provide our query algorithm for high-cardinality host identification, which is also our main contribution.

5.4.1 Basic Ideas

Noisy Group Testing. If we are given n distinct hosts and only one of them is a super-spreader, then we can assign these hosts into $\log n$ groups, and the assignment rule is that an item is assigned to group i if its i^{th} bit is 1, where $1 \leq i \leq \log n$. For each group, we calculate the sum of cardinalities of the hosts in the group. If the sum is larger than the threshold, then we consider there is a super-spreader in the group. By checking all the groups, we can know which groups contain the super-spreader, and we can recover the identity of this super-spreader according to the group indices. For calculating the cardinality sum of each group, we use OPT to estimate it, which gives an $(1 \pm \epsilon)$-approximation with probability at least $\frac{2}{3}$. From now on, we use OPT to refer to that optimal cardinality estimation algorithm. Using the cardinality estimation algorithm means that we may get an opposite result for each group: while a group in fact contains a super-spreader, the estimation algorithm result says it does not contain one; vice versa. And this is where the "noise" come from.

Error Correcting Code. In order to remove the noise in the noisy group testing procedure, we use error correcting code to encode the identities of the hosts before they are put into the group testing procedure, and decode the identity of the super-spreader returned from the group testing procedure. Encoding the identity of a host adds some additional bits to it, so the number of groups we are going to maintain will increase correspondingly. During the noisy group testing procedure, one bit of the encoded identity of the super-spreader may be incorrectly recovered due to the cardinality estimation error, the decoding procedure will remove this one-bit error and give back the correct identity of the super-spreader.

Identifying Multiple Super-spreaders. The above group testing technique works when the assumption that there is only one super-spreader in the given set of hosts stands. However, in the packet stream, there may be more than one super-spreaders. In order for the group testing technique to work, we have to separate all the hosts into

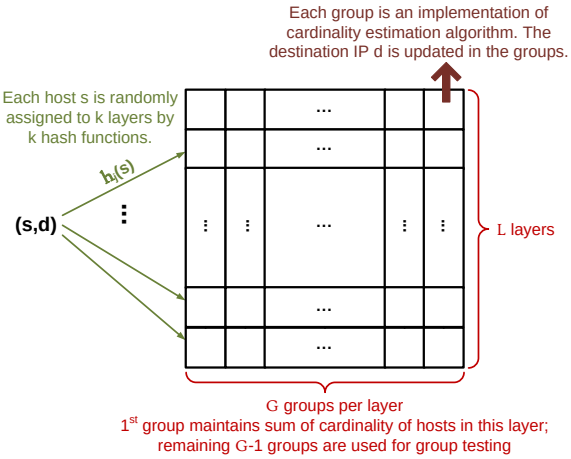


Figure 5.2 An overview of our sketch

different subsets, each of which contains one super-spreader, and use group testing on each subset. Suppose we want to separate all the hosts into L subsets where the number of subsets is larger than the number of super-spreaders, and each subset contains at most one super-spreader. We use κ randomly chosen universal hash functions for the separation procedure: a host is assigned to a subset which is determined by a hash function; each host will be assigned to κ subsets by the κ different hash functions. By using a universal hash function, we can separate the hosts randomly, but we may still end in the condition that two or more super-spreaders are assigned to the same subset; by using multiple different universal hash functions, we can have higher probability that for each super-spreader, there exists one subset that contains this super-spreader and only this one.

5.4.2 Overview of Our Sketch

An overview of our data structure is shown in Fig. 5.2. We use a 2-dimensional $L \times G$ data structure (sketch) to maintain the cardinality information of all the hosts in a packet stream at a router. There are L layers in the data structure, each of which represents a subset of the hosts. Each host is assigned to k layers/subsets by k universal

hash functions $h_j(.)$, $j = 1, 2, \ldots, k$. In each layer, there are G groups which correspond to $G-1$ groups used for group testing and 1 additional group that maintains the sum of cardinalities of all the hosts in the subset. We apply the noisy group testing technique in each layer. For each group, it is an instance of OPT, and can estimate the sum of cardinalities of the hosts assigned to this group. We use $C[a, b]$ to denote a group in our data structure, where $0 \leq a \leq L-1$, $0 \leq b \leq G-1$.

The values of L, G, k are set as below and why they are taking these values would be shown and proved in section 5.5.

$$L = O(\frac{1}{\epsilon} \log \frac{1}{\delta}) \tag{5.9}$$

$$G = O(\log \frac{m}{L}) \tag{5.10}$$

$$k = O(\log \frac{1}{\delta}) \tag{5.11}$$

Besides this 2-dimensional data structure, we also maintain an independent data structure to estimate F_t in the network using OPT.

5.4.3 Update

The brief initialization and update algorithm is shown in Fig. 5.3. When a packet (s, d) arrives at a local router, we will update our sketch maintained at the router by using a similar procedure in Liu et al. (2012). We first use a set of κ hash functions $h_j, 1 \leq j \leq \kappa$ to find a set of κ layers for the host s, and the j^{th} hash function is defined as

$$h_j(x) = (x \bmod L) \oplus h'_j(\lfloor x/L \rfloor) \tag{5.12}$$

for $j \in [1, \kappa]$, where $h'_j(x)$ is a universal pair-wise independent hash function that maps x into $[0, L-1]$. The value of L should be chosen as a power of 2 in order to guarantee that $h_j(x)$ is in the range $[0, L-1]$. The hash function $h_j(x)$ basically XORs the remainder of s divided by L and the randomized quotient $q(s) = \lfloor s/L \rfloor$ as the index of the layer to which s is assigned to.

1: **Initialization():**
2: Initialize L, G, κ.
3: Initialize a 2-dimensional $L \times G$ array $C[L, G]$, each element of which is an instance of OPT.
4: Randomly choose κ pairwise independent hash functions $h'_j : [1, \frac{m}{L}] \rightarrow [0, L-1]$ where $j = 1, \ldots, \kappa$.
5: Initialize $h_j(x) = (x \bmod L) \oplus h'_j(\lfloor x/L \rfloor)$ where $j = 1, \ldots, \kappa$.
6: **Update** (s, d)**:**
7: $q(s) \leftarrow \lfloor \frac{s}{L} \rfloor$
8: $W(q(s)) = \text{Encode}(q(s))$
9: **for** $j = 1$ to κ **do**
10: Insert d into $C[h_j(s), 0]$
11: **for** $i = 1$ to $G - 1$ **do**
12: **if** i^{th} bit of $W(q(s))$ is 1 **then**
13: insert d into $C[h_j(s), i]$
14: **end if**
15: **end for**
16: **end for**

Figure 5.3 Algorithm for initializing the data structure and updating a packet into it.

Suppose the layer selected by hash function h_j is ℓ_j. We will always add the destination d into the first group of ℓ_j. The first group will be helpful to filter false positives in the query procedure. For the remaining groups in ℓ_j, they are used for noisy group testing procedure. As mentioned previously, in order to remove the noise caused by the cardinality estimators, we use an error-correcting code Betten et al. (2006) to encode the quotient $q(s)$ into a codeword $W(q(s))$. As indicated by the group testing idea described in section 5.4.1, the number of groups used here is determined by the length of the codeword $W(q(s))$ in binary representation and further determined by the error-correcting code we use. For example, if the quotient length is 26 in binary representation and Hamming code is used, then the codeword length is 31, which means the number of groups for group testing in each layer would be 31, and $G = 32$. According to the group testing idea, we add the destination d into the groups of layer ℓ_j whose indices correspond to the 1-bits in the binary representation of codeword $W(q(s))$.

There are many different error-correcting codes that can be used in our algorithm. The Hamming code mentioned above is a good choice. Another simple alternative is repetition code, in which way, we simply run the cardinality estimation algorithm r times for each bit in the quotient $q(s)$ and use the majority to decode $q(s)$. In this case, if the quotient length is 26, then the codeword length would be $26r$. A similar idea was used in Guan et al. (2009), where a large Bloom filter is used to bound the error in the cardinality estimation. In our evaluation, we show that we can choose an efficient error-correcting code to reduce the space and the running time, significantly.

Fig. 5.4a shows an example of the update step at a local router. In this figure, each row corresponds to a layer and each square unit in a layer corresponds to a group which is an instance of cardinality estimator. At each local router, we also add (s, d) to the data structure for estimating F_t.

5.4.4 Merge

As indicated in the previous part of this chapter, we use the optimal cardinality estimation algorithm (OPT) in Kane et al. (2010) to estimate the cardinality of destination IPs of the hosts assigned to each group $C[i, j], 0 \leq i \leq L-1, 1 \leq j \leq G-1$. The full optimal algorithm is quite complicated and involves using highly independent hash functions and variable-bit-length array (VLA) to achieve optimal space and time in their theoretical analysis. In our implementation, we use pair-wise independent hash functions and simplified array of counters instead of complicated VLA.

Since we are going to merge multiple sketches collected from routers at the NOC to generate a new sketch, it is important to show that our sketch is easily mergable. By mergable, we mean that we can combine N same-size sketches at multiple routers together to generate a new sketch of the same size, and use this newly generated sketch to recover the super-spreaders; the recovered result is same as that by updating the packets of the multiple streams into a sketch and recovering from it. It is obvious that

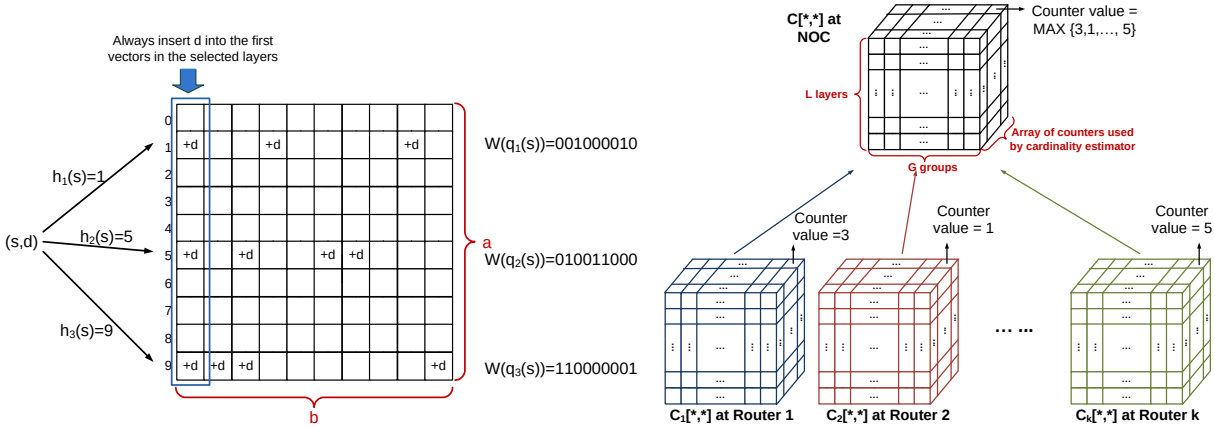
by transmitting the sketches instead of the streams from the routers, we can save a lot of effort.

It is straightforward to see that if we use same-size sketches, same set of κ hash functions $h_j(x), 1 \leq j \leq \kappa$ and same error-correcting code at all the routers, and if the cardinality estimator of each group is mergable, then our sketch is mergable. We briefly describe the OPT algorithm and show that the data structure used in this algorithm is mergable.

The first technique used in OPT is the balls-and-bins model. If we randomly throw α different balls into β bins, then the number of non-empty bins γ is a random variable that has the expectation $E[\gamma] = \beta(1 - (1 - \frac{1}{\beta})^{\alpha})$. When α's value is smaller than β, the variance of γ can be bounded and the observed value of γ is highly concentrated around $E[\gamma]$ with a high probability.

The second technique used in OPT is adaptive sampling. Since it requires that the number of distinct balls thrown into the bins is smaller than the number of bins in order to bound the estimation error and the number of bins do not change, a proper sampling rate needs to be used to always maintain a sampled set of distinct balls which are thrown into the bins. The sampling rate decreases by half everytime the number of distinct balls is larger than the number of bins. The sampling rule is as following: if the current sampling rate is $\frac{1}{2^r}$, then all the distinct balls whose $lssb$ is not smaller than r are sampled. In order to guarantee the randomness of the sampling process, a pair-wise independent hash function is used to randomize the IDs of the distinct balls. The bins are represented by an array of counters, each of which maintains the deepest $lssb$ of the distinct balls that are thrown into this bin/counter. As indicated in Kane et al. (2010), the deepest $lssb$ provides enough information to check if a bin is empty or not: if the $lssb$ maintained by a bin is not smaller than the current r, then this bin is non-empty.

We can see that if we have two instances of OPT with same size as well as hash functions and want to merge them together, then we can check the current sampling

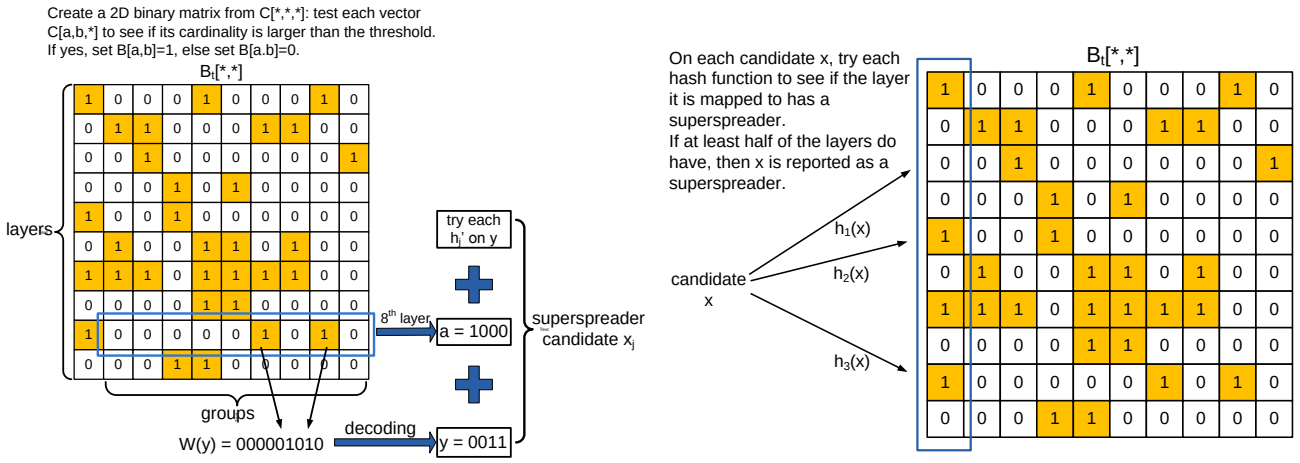


(a) An example of updating (s, d) into our sketch. (b) Merge multiple sketches at NOC.

Figure 5.4 An example of the Update and Merge steps. Left sub-figure shows the update of a source and destination IP address pair into our sketch. Right sub-figure shows the merge of multiple sketches collected from different routers at the NOC to generate a new sketch.

rates of the two OPTs and choose the smaller sampling rate. For the corresponding two arrays of counters, we can use the MAX operation to merge each pair of counters together since the largest $lssb$ is useful for each bin. In this way, we can create a new instance of OPT that maintains the cardinality information of the two streams which is maintained by the two original OPTs respectively, which means that the data structure of OPT is mergable.

In our algorithm, at the end of each measurement interval, the routers will send their sketches $C[*, *]$ to the NOC. And the NOC will merge all sketches together to get a network-wide view of the network traffic. .The data structure for estimating$F$ can be merged in the similar way. An example of the merge step is shown in Fig. 5.4b. In this figure, we use 3-dimensional cubics to represent our sketches, and the additional dimension corresponds to the array of counters in OPT algorithm.



(a) High-cardinality Host Recovery. (b) False Positive Filter-1.

Figure 5.5 An example of the Query step. Left sub-figure shows cardinality test and high-cardinality host recovery. Right sub-figure shows the false positive filter which is based on the first group in each layer.

5.4.5 Query

Given a threshold θ, our goal is to identify all the hosts x with $D^x > \theta + \epsilon F$. We divide the whole query procedure into four basic steps and the brief steps are shown in Fig. 5.6.

1. *Threshold Computation:* We use the merged OPT data structure at NOC for the F (definition see table 5.1) estimation and denote the estimation result by $\hat{F}$. So the threshold value we will use is $\theta + \epsilon\hat{F}$

2. *Cardinality Test:* We scan each group in the 2-dimensional $L \times G$ array one-by-one, and test whether a high-cardinality host is mapped into the group. We create a $L \times G$ binary matrix $B[*, *]$ to record the scanning results. For each group $C[a, b]$, the number of destinations in this group is estimated by the OPT algorithm. Let $\hat{F}[a, b]$ denote the estimated cardinality from $C[a, b]$. If

$$\hat{F}[a, b] > \theta + \epsilon\hat{F}, \tag{5.13}$$

we mark the bit at $B[a, b]$ as 1, and otherwise 0.

Query():
Get an estimation of total number of distinct destinations $\hat{F}$.
Given θ and ϵ, set $T \leftarrow \theta + \epsilon\hat{F}$.
Initialize a 2-dimensional $L \times G$ binary array $B[L, G]$ as all $0's$.
for $i = 0$ to $L - 1$ **do**
  for $j = 0$ to $G - 1$ **do**
    if $C[i, j] > T$ **then**
      Set $B[i, j] \leftarrow 1$.
    end if
  end for
end for
for $i = 0$ to $L - 1$ **do**
  Set $y \leftarrow$ Decode$(B[i, 1 \ldots G - 1])$.
  for $j = 1$ to κ **do**
    Set $x \leftarrow y \times L + i \oplus h'_j(y)$.
    Insert x into the candidate set $\mathcal{X}$.
  end for
end for

Figure 5.6 Algorithm for querying sketch and getting a candidate list of high-cardinality hosts.

3. *High-cardinality Host Recovery:* Next, we try to recover the identities of high-cardinality hosts from the binary matrix $B[*, *]$. In a^{th} layer, the bits from $B[a, 1]$ to $B[a, G - 1]$ form a binary code that is close to a codeword $W(q(x))$ of a certain unknown high-cardinality host x. By "close", we mean that due to the errors of the cardinality estimation algorithm,the binary code $B[a, 1 \cdots G - 1]$ may have some bits different from the original codeword $W(q(x))$. We apply the decoding algorithm of the error-correcting code, and recover an original message, denoted by y, which is supposed to be $q(x)$. By choosing a large enough codeword, we can guarantee that the message y should be the same as the quotient $q(x)$ with a high probability.

   Given $y = q(x)$ and the layer index a which is a hash value generated by one of the hash functions in $h_j(\cdot), 1 \leq j \leq \kappa$, if we know which hash function is used, then we will be able to recover x. However, we do not know about this. Thus, we

try each hash function one by one, and add all possible high-cardinality candidates to a candidate set $\mathcal{X}$. For each hash function $h_j(\cdot), 1 \leq j \leq \kappa$, we can recover a candidate host x_j from the message y as

$$x_j = y \times L + a \oplus h'_j(y). \tag{5.14}$$

We can see that if we have $y = q(x)$ for the high-cardinality host x, we will have $a = h_j(x)$ and

$$\begin{aligned} & x_j & (5.15) \\ = \; & q_j(x) \cdot L + h_j(x) \oplus h'_j(\lfloor x/L \rfloor) \\ = \; & \lfloor x/L \rfloor \cdot L + (x \bmod L) \oplus h'_j(\lfloor x/L \rfloor) \oplus h'_j(\lfloor x/L \rfloor) \\ = \; & \lfloor x/L \rfloor \cdot L + (x \bmod L) \\ = \; & x. & (5.16) \end{aligned}$$

And the high-cardinality host candidate x_j would be added into the set $\mathcal{X}$.We add all high-cardinality candidates at each layer into the set $\mathcal{X}$. An example is shown in Fig. 5.5a.

4. *False Positive Filters:* Notice that even if we use the error correcting code to remove cardinality estimation errors, depending on the feature of the code we use, it is possible that there may be still some errors that are not corrected and the recovered identities are not true high-cardinality hosts. In order to remove the false positives, we use some techniques to filter out some of them.

   Filter-1. At the last step, we try to remove false positives in the set $\mathcal{X}$ by using the first column in $B[*, *]$. For each candidate $x \in \mathcal{X}$, we use the same set of hash functions $h_j(\cdot)$ to map them into the bits in the first column $B[*, 0]$, as shown in Fig.5.5b. If x is a high-cardinality host, there should be more than half of the bits in its hash positions that are 1s with a high probability. Therefore, if the number

of 1 bits is smaller than half of the number of hash functions, we will remove x from $\mathcal{X}$.

Filter-2. If space allows, we use a Bloom Filter to maintain a sketch of the source IP addresses we have seen in the stream to further reduce the number of false positives. And for each recovered high-cardinality candidate, after it passes filter-1, we will also check whether it is stored in the Bloom Filter: if yes, then we will keep it in the candidate set $\mathcal{X}$; otherwise remove it.

Our algorithm will report the final result $\mathcal{X}$ as the set of the high-cardinality hosts.

Our algorithm follows a general procedure in the group testing problem, which has been widely used in the coding theory. In the following theoretical analysis section, we will analyze our algorithm as a channel-coding problem, which provides the proof for the error bound. In addition, we prove that the running time to identify high-cardinality hosts in our algorithm is sub-linear, which guarantees that our algorithm is practical for the high-speed network monitoring.

5.5 Theoretical Analysis

In the previous section, we provide a description of our data structure and the algorithms to update it and further identify the high-cardinality hosts from it. There are many parameters in our data structure, which have not been determined exactly. In this section, we provide a detailed theoretical analysis to help network engineers to choose proper parameters based on their requirements. The main contribution of our work is that we can identify high-cardinality hosts *efficiently* and accurately in a distributed network monitoring system. Here, by "efficiently" we mean that both the space and the running time can be bounded by a polynomial function of the bit length of the host identification, i.e. $poly(\log m)$; by "accurately" we mean that we can report a list of high-cardinality hosts with low false positive and false negative rates.

5.5.1 Accuracy of Our Algorithm

For our algorithm to be an (ϵ, δ)-approximation algorithm that is defined in 3, our algorithm have to guarantee the following three events to happen with high probability:

- Event 1: for each high-cardinality host x, there exists a layer that contains only one high-cardinality host which is x, such that the group testing technique can work in this layer.
- Event 2: if a layer contains a high-cardinality host x, our query algorithm would recover it correctly.
- Event 3: if a recovered host x is a true high-cardinality host, our false positive filter will not remove it.

Below we will analyze the probabilities that these three events happen respectively, and the corresponding parameters to be used to guarantee these probabilities.

5.5.1.1 Event 1

Lemma 4. Given the number of hash functions $\kappa = \log(\frac{3}{\delta})$ and the number of layers $L = \frac{2}{\epsilon}\log(\frac{3}{\delta})$, for any high-cardinality host x, there exists a layer with $D^x > \theta + \epsilon F$, such that only x is hashed into this layer and no other high-cardinality hosts are hashed into this layer, with probability at least $1 - \frac{\delta}{3}$.

Proof. Each host is hashed by $\kappa = \log\frac{3}{\delta}$ universal hash functions into $L = \frac{2}{\epsilon}\log\frac{3}{\delta}$ layers. For a certain high-cardinality host x and the j^{th} hash function $h_j(\cdot)$, let $a = h_j(x)$ be the layer that x is hashed into by $h_j(\cdot)$. Since the hash function we use is universal which means that each distinct host will be hashed to layer a by $h_j(\cdot)$ with probability $1/L$. Let the sum of the destination cardinalities of hosts other than x that are hashed to layer a by $h_j(\cdot)$ is denoted by F_j^{-x}. Then the expectation of F_j^{-x} would be

$$E[F_j^{-x}] = \frac{F - D^x}{L}. \tag{5.17}$$

Since each host will be hashed by κ universal hash functions, the expectation of the sum of destination cardinalities of hosts other than x that are hashed to layer a by all the hash functions would be

$$\sum_{j\in[1,\kappa]} F_j^{-x} = \frac{\kappa}{L}(F - D^x) = \frac{\epsilon}{2}(F - D^x) < \frac{\epsilon F}{2} \tag{5.18}$$

Based on Markov's inequality, the probability that the sum of the cardinalities of hosts other than x that are hashed to layer a is larger than ϵF_t is smaller than $\frac{1}{2}$. Therefore, layer a contains only one high-cardinality host x with probability at least $\frac{1}{2}$. By using all $\log(\frac{3}{\delta})$ hash functions, for a high-cardinality host x, the probability that there exists a layer such that only one high-cardinality host x is hashed into this layer and no other high-cardinality hosts are hashed into it is at least $1 - \frac{\delta}{3}$. □

5.5.1.2 Event 2

Lemma 5. If there is only one high-cardinality host x hashed into a layer, our query algorithm will recover x and add it into $\mathcal{X}$ with probability at least $1 - \frac{\delta}{3}$, given the length of the codeword $G - 1 > \frac{1}{1-H(2/3)} \log(\frac{m}{L})$, where $H(p)$ is the entropy of a random variable with Bernoulli distribution p.

Proof. For the high-cardinality host x, we suppose it is hashed by hash function $h_j(\cdot)$ into layer a. We encode its quotient $q(x)$ into a codeword $W(q(x))$ with a bit length $G - 1$. At the NOC, we will run the cardinality estimation algorithm and get the bit matrix $B[*, *]$. If the b^{th} bit in $W(q(x))$ is 1, we will get 1 at $B[a, b]$ with probability at least $p = 2/3$ according to the OPT algorithm. Otherwise, we will get 0 with probability at least $p = 2/3$. Therefore, we can treat $B[a, b]$ as the received symbol through a channel as shown in Fig.5.7. Based on the information theory, we can find a way to encode $q(x)$ and decode it from $B[a, *]$ with a high probability. Due to the progress in the list-decoding method Guruswami (2007), we can encode $q(x)$ with a rate close to $1 - H(2/3)$ that is

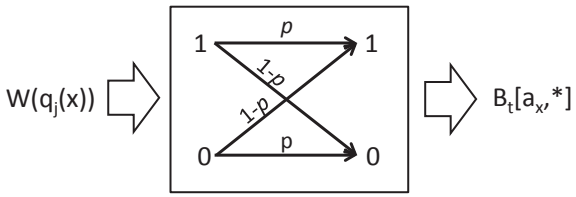


Figure 5.7 A Channel Model for Cardinality Estimation

the capacity of the channel in Fig.5.7, and recover $q(x)$ in $O(poly(\log(\frac{m}{L})))$ running time with a high probability.

Once we find the quotient $q(x)$ of the high-cardinality host x, we can try all hash functions and further recover this host x. Because each host is hashed into $\log\frac{3}{\delta}$ layers, the recovery procedure will fail with probability smaller than $\delta/3$. □

5.5.1.3 Event 3

Lemma 6. By using our false positive filter, our algorithm would report a true high-cardinality host x that is returned by the query process with probability at least $1-\delta/3$.

Proof. With the same analysis in the proof of Lemma 5, for a true high cardinality host x, for $j = 1, \ldots, \kappa$, in the false positive filtering procedure, the bit $B[h_j(x), 0]$ is 1 with probability at least $2/3$. Therefore, the probability that the number of 1s in these bits is larger than $\frac{1}{2}\log(\frac{3}{\delta})$ is at least $1-\delta/3$. Thus, a true high-cardinality host x will pass the filter with probability at least $1-\delta/3$. □

5.5.1.4 Final Results

Based on the three events and the union bound, we have the following result:

Theorem 3. A high-cardinality host x with $D^x > \theta+\epsilon F$ can be reported with probability at least $1-\delta$.

The size of the candidate hosts $|\mathcal{X}|$ can be bounded by the following lemma.

Lemma 7. The number of candidate hosts in our algorithm is bounded by $\frac{2}{\epsilon}\log^2(\frac{3}{\delta})$.

Proof. According to the query algorithm in figure 5.5, we have $L = \frac{2}{\epsilon}\log\frac{3}{\delta}$ layers and we at most recover $\kappa = \log\frac{3}{\delta}$ candidate hosts from each layer. Therefore, the number of candidate hosts can be bounded by $\frac{2}{\epsilon}\log^2(\frac{3}{\delta})$, which is also a loose upper bound of the number of false positives. □

5.5.2 Space and Time

Next, we analyze the space and running time of our algorithm.

Theorem 4. Our data structure requires

$$O(\frac{1}{\epsilon}(\frac{1}{\epsilon^2} + \log m)\log(\frac{1}{\delta})\log(\frac{\epsilon m}{\log(1/\delta)})) \tag{5.19}$$

bits.

Proof. There are $O(\frac{2}{\epsilon}\log\frac{3}{\delta})$ layers in our sketch. In each layer, there are $O(\log\frac{\epsilon m}{\log(1/\delta)})$ groups. And each group requires $O(\frac{1}{\epsilon^2} + \log m)$ bits. By multiplying them tougher, we can get our result. Other parts in our algorithm will require a space much smaller than the sketch $C[*,*]$. □

Because the cardinality estimation can be done in $O(1)$ running time for both update and query, the performance of our algorithm is mainly determined by the identification of high-cardinality hosts at the NOC.

Theorem 5. The update running time at each monitor is

$$O(\log(\frac{1}{\delta})\log(\frac{\epsilon m}{\log(1/\delta)})). \tag{5.20}$$

The merge running time and the query running time at the NOC are

$$O(\frac{1}{\epsilon^3}\log(\frac{1}{\delta})\log(\frac{\epsilon m}{\log(1/\delta)})) \tag{5.21}$$

and

$$O(\frac{1}{\epsilon}\log(\frac{1}{\delta})\log^{O(1)}(\frac{\epsilon m}{\log(1/\delta)}) + \frac{1}{\epsilon}\log^3(\frac{1}{\delta})) \tag{5.22}$$

respectively.

Proof. At each monitor, we use $O(\log(\frac{1}{\delta}))$ hash functions to map each host to layers. For each layer, we need to update at most $O(\log(\frac{\epsilon m}{\log(1/\delta)}))$ groups. But each group only requires $O(1)$ running time. Therefore, the update running time at a local monitor is $O(\log(\frac{1}{\delta})\log(\frac{\epsilon m}{\log(1/\delta)}))$.

In each group, there are $O(\frac{1}{\epsilon^2})$ counters. And there are totally $O(\frac{1}{\epsilon}\log(\frac{1}{\delta})\log(\frac{\epsilon m}{\log(1/\delta)}))$ groups in our sketch. Therefore, the running time for merging sketches in our algorithm is $O(\frac{1}{\epsilon^3}\log(\frac{1}{\delta})\log(\frac{\epsilon m}{\log(1/\delta)}))$.

To compute the binary matrix $B[*,*]$, we only need $O(\frac{1}{\epsilon}\log(\frac{1}{\delta})\log(\frac{\epsilon m}{\log(1/\delta)}))$ running time. To recover a high-cardinality host in each layer, we need $O(\log^{O(1)}(\frac{\epsilon m}{\log(1/\delta)}))$ running time for the list-decoding algorithm Guruswami (2007), and $O(\log(\frac{1}{\delta}))$ running time to map quotient back to the host identification. Thus, we need $O(\frac{1}{\epsilon}\log(\frac{1}{\delta})(\log^{O(1)}(\frac{\epsilon m}{\log(1/\delta)}) + \log(\frac{1}{\delta})))$ running time to get the candidate set $\mathcal{X}$.

Because there are at most $O(\frac{1}{\epsilon}\log^2(\frac{1}{\delta}))$ candidates in $\mathcal{X}$, the algorithm to filter false positives will require $O(\frac{1}{\epsilon}\log^3(\frac{1}{\delta}))$ running time. Therefore, the running time for the query can be bounded by $O(\frac{1}{\epsilon}\log(\frac{1}{\delta})\log^{O(1)}(\frac{\epsilon m}{\log(1/\delta)}) + \frac{1}{\epsilon}\log^3(\frac{1}{\delta}))$. □

Based on the above results, we can identify high-cardinality hosts with a running time bounded by a polynomial function of the bit length of the host identification, i.e. $O(poly(\log m))$. And the space requirement of our algorithm is close to the lower bound of the compress sensing Do Ba et al. (2010).

There is a close relation between our reversible data structure and the compressed sensing problem Candes et al. (2006); Donoho (2006); ric (). For a signal $\mathbf{f}$, its compressed measurement equals to $\mathbf{Mf}$, where $\mathbf{M}$ is a carefully chosen $l \times m$ matrix with $l \ll m$. The group $\mathbf{Mf}$ is called as the measurement group, and can be used recover $\mathbf{f}$ beyond the

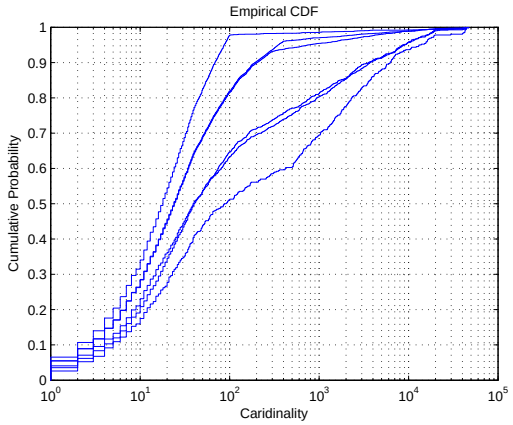


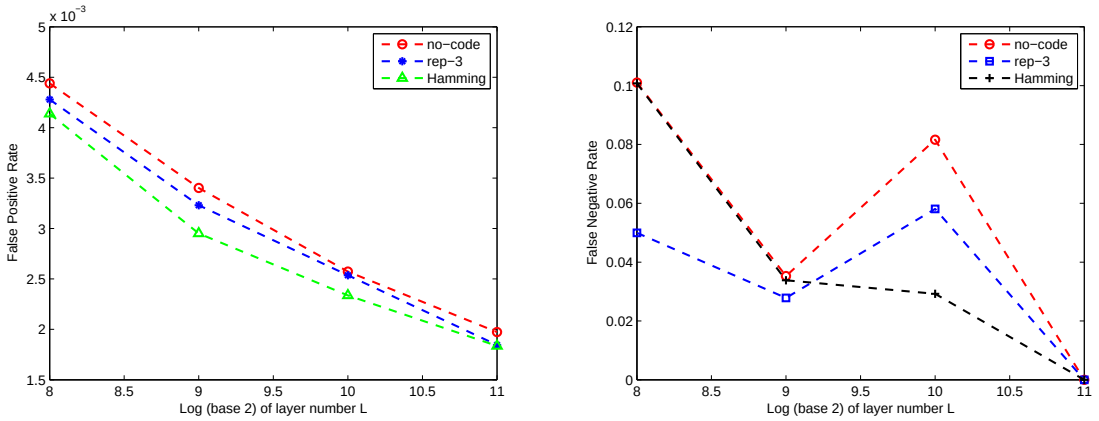
Figure 5.8 Cumulative Distribution of the destination cardinalities of some experimental data sets used in our experiment.

information bound. The high-cardinality host identification introduces a new challenge for the compressed sensing. Besides the noises in the original signal $\mathbf{f}$, there are also noises in the measurement group $\mathbf{Mf}$. In other words, even if there is no noise in $\mathbf{f}$, we still have some errors in the measurement group $\mathbf{Mf}$. This is because there is no efficient data structure that can be used to estimate the cardinality without the approximation or the randomness. We show that the traditional methods using the error-correcting code can be used to recover a sparse signal under the noises in the measurement group $\mathbf{Mf}$.

5.6 Performance Evaluation

5.6.1 Experiment Settings

We use false positive rate and false negative rate to evaluate our algorithm. False positives are source IP addresses which are recovered by our algorithm but not true high-cardinality hosts. False negatives are those source IP addresses which are true high-cardinality hosts but not recovered by our algorithm. False positive rate is calculated as the number of false positives divided by total number of source IP addresses in the data; false negative rate is the number of false negatives divided by total number of true



(a) False positive rate

(b) False negative rate

Figure 5.9 Precision of different versions of our algorithm. $\delta = 0.15$. WittyWorm trace is used.

high-cardinality hosts.

WittyWorm trace Rea () is used to evaluate our algorithm. The WittyWorm data are collected during the breakout of the Witty worm. It contains the packages sent by the infected hosts to a /8 subnet which are all not-in-use IP addresses. Since these traces are collected from all over the world and are very large, we use part of the traces to simulate the break out of the worm in a subnet, and try to detect the high-cardinality hosts in the subnet. We extracted 10 data sets from the WittyWorm trace, and some cumulative distributions of the destination cardinality of the source IP addresses in the data sets are shown in figure 5.8. For each test, we run our algorithm on each data sets at least 50 times and use the average results.

5.6.2 Evaluations

5.6.2.1 Comparisons of Different Coding Methods

In this experiment, we test the precision of three versions of our algorithm which are differentiated by the coding method used for the source IP address's quotient. The three different versions are denoted as no-code, rep-3 and Hamming. As the name indicates,

no-code means we will not encode the quotient, and for each bit of the quotient, it will be updated into one group in each selected layer; rep-3 means that for each bit of quotient, it corresponds to 3 repeated bits in the code word and it will be updated into 3 corresponding groups in each selected layer; Hamming means that we encode the quotient with Hamming code and each bit of the code word will be updated into one group in each selected layer. For this test, we first select the layer number L. When L is fixed we know the length of the source IP's quotient, which determines the number of groups to be used in each layer according to the coding method used.

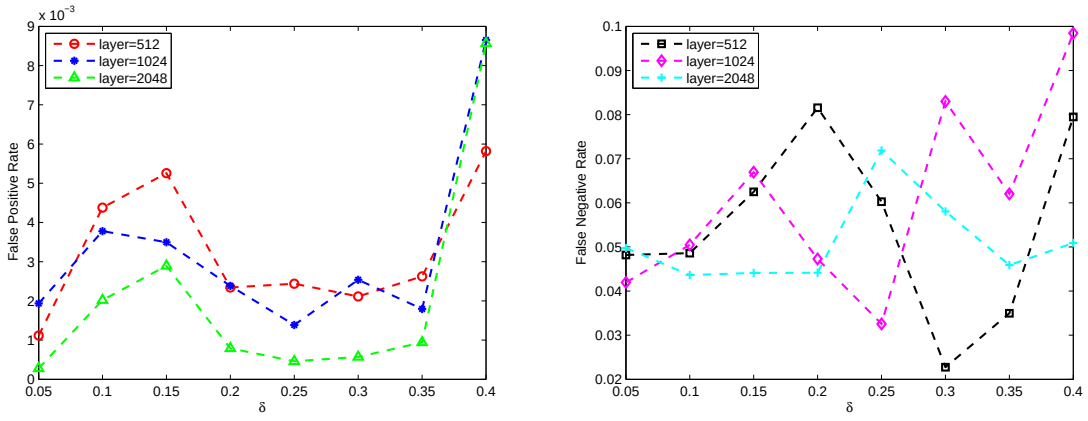
The experiment results are shown in Fig. 5.9. We can see that when the layer number is fixed, the rep-3 and Hamming versions are performing better than the no-code version in the sense that the previous two versions are generating less false positives and false negatives. This is what we expected that the error-correcting code can help us to correct some of the errors caused by cardinality estimation, recover more false high-cardinality hosts, and reduce the number of false positives. However, when using error-correcting codes, we are using more groups in each layer comparing with no-code version, which means that we are using more space to get higher accuracy. On the other hand, when the layer number increases, the false positive rate and false negative rate decrease, which conforms to our algorithm design that when we are using more layers, we will have less chance to get collisions that two or more high-cardinality hosts are stored in the same layer, and will have more chance to filter out false positives using Filter-1.

5.6.2.2 False Positive Filter-2

In this experiment, we compare the condition when only Filter-1 is used to filter false positives and the condition when both Filter-1 and Filter-2 are used. In this test, we used 2^{16} bits for the Bloom Filter. The results are shown in table 5.2. From the results we can see that Filter-2 reduces the false positive rate, which makes sense since we are using extra space for Filter-2 to store the information of source IP addresses which have

Table 5.2 False positive rates for different ϵ under condition when only Filter-1 is used and condition when both Filter-1 and Filter-2 are used. Hamming code version of our algorithm is used. $\delta = 0.15$.

ϵ	Filter-1	Filter-1+Filter-2
0.0045	0.004072	0.002837
0.0085	0.007018	0.002951
0.0170	0.007322	0.004138



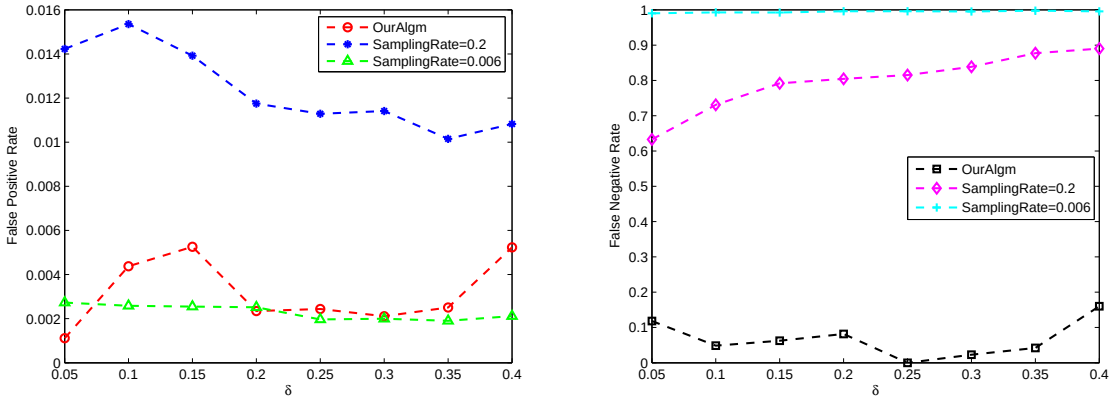
(a) False positive rate (b) False negative rate

Figure 5.10 Precision of our algorithm under different δ and L values. WittyWorm trace is used.

been observed in the data. When our main algorithm recovers high-cardinality hosts using the group testing methods, it may generate some candidates that have never been observed in the data, which could be removed by Filter-2.

5.6.2.3 Error Bound δ

In this experiment, we evaluate the Hamming version of our algorithm under different values of δ to see if the false positive rate and false negative rate can achieve the expected error bound δ. We tested our algorithm for δ values ranging from 0.05 to 0.40 under different values of L for different values of 512, 1024, 2048. When δ and L are fixed, ϵ can be determined accordingly which will also determine the threshold for the high-cardinality hosts.



(a) False positive rate

(b) False negative rate

Figure 5.11 Comparing precision of sampling algorithm and our algorithm under different δ values. WittyWorm trace is used.

The evaluation results are shown in Fig. 5.10. We observe that the false positive rates are much smaller than the error bound δ which means that our algorithm will report a small number of false positives than the expected bound. We also observe that the false negative rate is also bounded by the success bound δ which means that our algorithm can report the expected portion of the true high-cardinality hosts.

5.6.2.4 Compare with Sampling Technique

We implemented the superspreader detection algorithm in paper Venkataraman et al. (2005) which uses one-level sampling to estimate and capture superspreaders in the network packets. In their algorithm, each unique source-destination IP address pair is sampled with a pre-selected sampling rate and stored in a hash table. For each sampled source, its destination cardinality is counted using the sampled packets and finally estimated according to the sampling rate. We implemented the sampling process using a pair-wise independent hash function which maps each 64-bit ID (source and destination IP address pair) universally onto the $[0, 1)$ range and sample the pair if its hash value is smaller than the sampling rate. We compare Hamming code version of our algorithm

with this sampling algorithm to see which one is performing better when the expected detection rate δ is fixed.

We can see from Fig. 5.11a and Fig. 5.11b that the higher the sampling rate is, the higher the false positive rate is and the lower the false negative rate is, which makes sense since the higher the sampling rate is, the more packets will be sampled and more non-superspreaders and superspreaders will be kept. There would be a trade-off between the false positive rate and false negative rate when using sampling technique. Another point is that the higher the sampling rate is, the more space is needed to store the packet information and the source IP addresses' destination cardinality. Also the space increases as the data stream size increases which is not desirable in large amount of distributed data streams. However, our algorithm has both lower false positive rate and false negative rate than the sampling technique, which due to the compact sketch we use to store the information and the false positive filters to remove false positives. Also our algorithm uses fixed space to store the information which does not change when the data stream size changes.

5.7 Conclusion

In sum, we provide a mergable and reversible data structure for the high-cardinality host detection. Our solution is a general solution that can be easily extended to other traffic features or different application domains. We also provide a detailed theoretical analysis of our data structure, which can help engineers to determine the parameters in practice. The high-cardinality host identification also introduces a new challenge for the compressed sensing problem, which will attract more research interests in the future. We hope our work can improve the practice of the traffic monitoring and the anomaly detection in the Internet.

CHAPTER 6. SUMMARY AND DISCUSSION

In this dissertation, we have provided new solutions for hot items identification in chapter 3, distinct element counting in chapter 4 and superspreader identification in chapter 5 in distributed dynamic data streams. Our algorithms could be used for detecting network attacks and anomalies such as DDoS attacks, service outages, worm spreading, etc. In our algorithms, we combine basic techniques such as hash function based randomization, adaptive distinct sampling, multi-dimensional counter-based sketching, group testing and statistical estimations to design highly compact data structures which use small space and constant query/update time and are suitable for network monitoring applications. Our theoretical analysis and data-based experimental evaluations have shown that our solutions work better than previous ones and give promising results.

Our work in this dissertation only resolves a small part of data streaming problems for network monitoring. There are more work to be done in the future to solve other practical problems. For example, we usually assume that the size of the data we are processing is known as a prior such that we could design our algorithms with the best parameters to minimize space and time, but in practice sometimes the data size is unknown or changes often, in which case more flexible data streaming algorithms are preferred to adapt to the unknown data size. Another practical issue is that there are tons of data streaming algorithms that deal with different problems under variant conditions. It would be very useful to create a generalized data streaming framework that combines or supports different types of problems which is also easy to setup and tune parameters, such that it can be used as a straight-forward component in network monitoring tools.

Network monitoring is becoming more and more challenging because of the growth of internet. One example is the concept of IoT (Internet Of Things), which brings physical devices other than computers and mobile phones into the infrastructure of internet. One of the great examples of IoT is smart home where home devices such as fridges, cameras, switchers, temperature controllers are crafted with chips and smart firmwares. These home devices are connected to the internet and can be accessed remotely by the home owner for convenient control. They are smart like computers and mobile phones, but they are not protected by anti-virus softwares like computers and mobile phones which makes them more vulnerable to network attacks. Another property which makes them great target of network attacks is that they stay online forever and are always ready to be compromised by attackers. In the late 2016, a massive DDoS attack against a DNS service provider which brought down piles of internet services was just started from a huge amount of IoT devices controlled by a botnet. This attack opens a new world for DDoS attack that is based on IoT which is easy to break into, stays online all the time and has more capacity than computers and mobile phones (in the future).

To combat with ever-evolving network attacks which utilize new internet concepts, we would have to learn the new features of the evolving internet and keep up with the smart attackers.

BIBLIOGRAPHY

2014 neustar annual ddos attacks and impact report.

The caida dataset on the witty worm - march 19-24, 2004.

The caida ucsd ddos attack 2007 dataset.

Nsfocus ddos threat report 2013.

Rice dsp group. compressed sensing resource: http://dsp.rice.edu/cs.

Agarwal, P. K., Cormode, G., Huang, Z., Phillips, J., Wei, Z., and Yi, K. (2012). Mergeable summaries. In *PODS '12*, pages 23–34, New York, NY, USA. ACM.

Arackaparambil, C., Brody, J., and Chakrabarti, A. (2009). Functional monitoring without monotonicity. *Automata, Languages and Programming, Lecture Notes in Computer Science*, 5555.

Aumuller, M., Dietzfelbinger, M., and Woelfel, P. (2012). Explicit and efficient hash functions suffice for cuckoo hashing with a stash. In *Proceedings of the 20th Annual European Symposium on Algorithms*.

Ayres, P. E., Sun, H., Chao, H. J., and Lau, W. C. (2006). Alpi: A ddos defense system for high-speed networks. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, 24(10):1864–1876.

Bar-Yossef, Z., Jayram, T., Kumar, R., Sivakumar, D., and Trevisan, L. (2002). Counting Distinct Elements in a Data Stream. In *6th International Workshop on Randomization and Approximation Techniques in Computer Science*, Cambridge, Massachusetts.

Berinde, R., Cormode, G., Indyk, P., and Strauss, M. J. (2009). Space-optimal heavy hitters with strong error bounds. In *ACM Principles of Database Systems (PODS'2009)*, Providence, Rhode Island.

Betten, A., Braun, M., Fripertinger, H., Kerber, A., Kohnert, A., and Wassermann, A. (2006). *Error-Correcting Linear Codes: Classification by Isometry and Applications (Algorithms and Computation in Mathematics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Brenner, S. W. (2009). *Cyberthreats: The Emerging Fault Lines of the Nation State*. Oxford University Press, Inc., New York, NY, USA.

Broder, A., Mitzenmacher, M., and Mitzenmacher, A. B. I. M. (2002). Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646.

Candes, E. J., Romberg, J. K., and Tao, T. (2006). Stable signal recovery from incomplete and inaccurate measurements. *Communications on Pure and Applied Mathematics*, 59(8):1207–1223.

Cao, J., Jin, Y., Chen, A., Bu, T., and Zhang, Z.-L. (2009). Identifying high cardinality internet hosts. In *INFOCOM '09, IEEE*.

Carter, J. L. and N.Wegman, M. (1979). Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154.

Carter, J. L. and Wegman, M. N. (1979). Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154.

Chan, C. L., Che, P. H., Jaggi, S., and Saligrama, V. (2011). Non-adaptive probabilistic group testing with noisy measurements: Near-optimal bounds with efficient algorithms. In *Communication, Control, and Computing (Allerton), 2011 49th Annual Allerton Conference on*, pages 1832 –1839.

Charikar, M., Chen, K., and Farach-Colton, M. (2002). Finding frequent items in data streams. *Proceedings of the 29th ICALP International Colloqium on Automata, Language and Programming*, page 693703.

Cheetancheri, S. G., Agosta, J. M., Dash, D. H., Levitt, K. N., Rowe, J., and Schooler, E. M. (2007). A distributed host-based worm detection system. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*.

Chen, A., Li, L. E., and Cao, J. (2009). Tracking cardinality distributions in network traffic. In *INFOCOM 2009, IEEE*, pages 819 –827.

Cormode, G., Datar, M., and Indyk, P. (2002). Comparing data streams using hamming norms (how to zero in). In *Proceedings of the 28th VLDB Conference*.

Cormode, G. and Hadjieleftheriou, M. (2010). Methods for finding frequent items in data streams. *The VLDB Journa*, 19(2):3–20.

Cormode, G. and Muthukrishnan, S. (2005a). An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75.

Cormode, G. and Muthukrishnan, S. (2005b). What's hot and what's not: tracking most frequent items dynamically. *ACM Trans. Database Syst.*, 30:249–278.

Cormode, G., Muthukrishnan, S., and Rozenbaum, I. (2005). Summarizing and mining inverse distributions on data streams via dynamic inverse sampling. In *VLDB '05 Proceedings of the 31st international conference on Very large data bases*.

Cormode, G., Muthukrishnan, S., and Yi, K. (2011). Algorithms for distributed functional monitoring. *ACM Transactions on Algorithms*, 7(2).

Demaine, E. D., Lopez-Ortiz, A., and Munro, J. I. (2002). Frequency estimation of internet packet streams with limited space. In *10th European Symposium on Algorithms (ESA'2002)*, Rome.

Do Ba, K., Indyk, P., Price, E., and Woodruff, D. P. (2010). Lower bounds for sparse recovery. In *SODA*, pages 1190–1197, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Donoho, D. (2006). Compressed sensing. *Information Theory, IEEE Transactions on*, 52(4):1289–1306.

Durand, M. and Flajolet, P. (2003). Loglog Counting of Large Cardinalities. In *European Symposium on Algorithms*, Budapest, Hungary.

Estan, C. and Varghese, G. (2002). New directions in traffic measurement and accounting. In *SIGCOMM '02*, pages 323–336, New York, NY, USA. ACM.

Fan, L., Cao, P., Almeida, J., and Broder, A. (2000). Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3).

Flajolet, P., Fusy, E., Gandouet, O., and Meunier, F. (2007). HyperLogLog: the Analysis of a Near-optimal Cardinality Estimation Algorithm. In *The 2007 Conference on Analysis of Algorithms*, Juan des Pins, France.

Flajolet, P. and Martin, G. (1985). Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2).

Gangam, S., Sharma, P., and Fahmy, S. (2013). Pegasus: Precision hunting for icebergs and anomalies in network flows. In *Proc. of International Conference on Computer Communications (INFOCOM'2013)*, Turin, Italy.

Ganguly, S. (2007). Counting distinct items over update streams. *Theoretical Computer Science*, 378:211–222.

Ganguly, S., Garofalakis, M., and Rastogi, R. (2004). Tracking set-expression cardinalities over continuous update streams. In *Proceedings of the 30th VLDB Conference*.

Gemulla, R., Lehner, W., and Haas, P. J. (2008). Maintaining bounded-size sample synopses of evolving datasets. *The VLDB Journal*, 17:173–201.

Gibbons, P. B. (2001). Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proceedings of the 27th VLDB Conference*.

Gibbons, P. B. and Tirthapura, S. (2001). Estimating simple functions on the union of data streams.

Gilbert, A. C., Kotidis, Y., Muthukrishnan, S., and Strauss, M. J. (2001). Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proceedings of the 27th VLDB Conference*.

Glatz, E. and Dimitropoulos, X. (2012). Classifying internet one-way traffic. In *Internet Measurement Conference 2012*.

Guan, X., Wang, P., and Qin, T. (2009). A new data streaming method for locating hosts with large connection degree. In *GLOBECOM '09*, pages 1 –6.

Guo, D., Wu, J., Chen, H., and Luo, X. (2010). Dynamic bloom filters. *IEEE Transactions on Knowledge and Data Engineering*, 22(1).

Guruswami, V. (2007). Algorithmic results in list decoding. *Found. Trends Theor. Comput. Sci.*, 2(2):107–195.

Hao, F., Kodialam, M., and Lakshman, T. V. (2008). Incremental bloom filters. In *IEEE International Conference on Computer Communications*.

Huang, G., Lall, A., Chuah, C.-N., and Xu, J. (2011). Uncovering global icebergs in distributed streams: Results and implications. *Journal of Network and Systems Management*, 19(1):84–110.

Indyk, P., Ngo, H. Q., and Rudra, A. (2010). Efficiently decodable non-adaptive group testing. In *21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'2010)*, Austin, TX.

Jelasity, M., Montresor, A., and Babaoglu, O. (2005). Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(1):219–252.

Kane, D. M., Nelson, J., and Woodruff, D. P. (2010). An Optimal Algorithm for the Distinct Elements Problem. In *ACM SIGMOD/PODS Conference*, Indianapolis, Indiana, USA.

Kirsch, A., Mitzenmacher, M., and Wieder, U. (2010). More robust hashing: cuckoo hashing with a stash. In *Proceedings of the 16th Annual European Symposium on Algorithms*.

Li, M. and Lee, W.-C. (2008). Identifying frequent items in p2p systems. In *The 28th International Conference on Distributed Computing Systems*, Beijing, China.

Li, X., Bian, F., Crovella, M., Diot, C., Govindan, R., Iannaccone, G., and Lakhina, A. (2006). Detection and identification of network anomalies using sketch subspaces. *IMC '06*, pages 147–152.

Liu, Y., Chen, W., and Guan, Y. (2012). A fast sketch for aggregate queries over high-speed network traffic. In *INFOCOM '12, IEEE*.

Locher, T. (2011). Finding heavy distinct hitters in data streams. In *SPAA '11*, pages 299–308, New York, NY, USA. ACM.

Manikopoulos, C. and Papavassiliou, S. (2002). Network intrusion and fault detection: a statistical anomaly approach. *Communications Magazine, IEEE*, 40(10):76 – 82.

Manku, G. S. and Motwani, R. (2002). Approximate frequency counts over data streams. In *28th International Conference on Very Large Data Bases (VLDB'2002)*, Hong Kong.

Metwally, A., Agrawal, D., and Abbadi, A. E. (2005). Efficient computation of frequent and top-k elements in data streams. In *Proc. of the 10th International Conference on Database Theory (ICDT'2005)*, Edinburgh, Scotland.

MISRA, J. and GRIES, D. (1982). Finding repeated elements. *Science of Computer Programming*, 2(2):143–152.

Padmanabhan, V. N., Ramabhadran, S., Agarwal, S., and Padhye, J. (2006). A study of end-to-end web access failures. In *International Conference on emerging Networking EXperiments and Technologies*.

Pagh, R. and Rodler, F. F. (2004). Cuckoo hashing. *Journal of Algorithms*, 51.

Sacha, J. and Montresor, A. (2013). Identifying frequent items in distributed data sets. *Computing*, 95(4):289307.

Schweller, R., Li, Z., Chen, Y., Gao, Y., Gupta, A., Zhang, Y., Dinda, P. A., Kao, M.-Y., and Memik, G. (2007). Reversible sketches: enabling monitoring and analysis over high-speed data streams. *IEEE/ACM Trans. Netw.*, 15:1059–1072.

Venkataraman, S., Song, D., Gibbons, P. B., and Blum, A. (2005). New streaming algorithms for fast detection of superspreaders. In *NDSS'05*, pages 149–166.

Woodruff, D. and Zhang, Q. (2012). Tight bounds for distributed functional monitoring. In *STOC 12 Proceedings of the forty-fourth annual ACM symposium on Theory of computing*.

Xie, Y., Sekar, V., Reiter, M., and Zhang, H. (2006). Forensic analysis for epidemic attacks in federated networks. *Network Protocols, IEEE International Conference on*, 0:43–53.

Zhao, H. C., Lall, A., Ogihara, M., and Xu, J. J. (2010). Global iceberg detection over distributed data streams. In *Proc. of IEEE International Conference on Data Engineering (ICDE'2010)*, Long Beach, CA.

Zhao, Q., Kumar, A., and Xu, J. (2005). Joint data streaming and sampling techniques for detection of super sources and destinations. In *IMC '05*, pages 7–7, Berkeley, CA, USA. USENIX Association.

Zhao, Q., Ogihara, M., Wang, H., and Xu, J. (2006). Finding global icebergs over distributed data sets. In *Proc. of ACM Syposium on Principles of Database Systems (PODS'2006)*, Chicago, IL.