Accurately Estimating User Cardinalities and Detecting Super Spreaders over Time

Peng Jia, Pinghui Wang, Yuchao Zhang, Xiangliang Zhang, Jing Tao,
Jianwei Ding, Xiaohong Guan, Don Towsley

Abstract—Online monitoring user cardinalities in graph streams is fundamental for many applications such as anomaly detection. These graph streams may contain edge duplicates and have a large number of user-item pairs, which makes it infeasible to exactly compute user cardinalities due to limited computational and memory resources. Existing methods are designed to approximately estimate user cardinalities, but their accuracy highly depends on complex parameters and they cannot provide anytime-available estimation. To address these problems, we develop novel bit/register sharing algorithms, which use a bit/register array to build a compact sketch of all users' connected items. Our algorithms exploit the dynamic properties of the bit/register arrays (e.g., the fraction of zero bits in the bit array) to significantly improve the estimation accuracy, and have low time complexity $O(1)$ to update the estimations for a new user-item pair. In addition, our algorithms are simple and easy to use, without requirements to tune any parameter. Furthermore, we extend our methods to detect super spreaders with large cardinalities in real-time. We evaluate the performance of our methods on real-world datasets. The experimental results demonstrate that our methods are several times more accurate and faster than state-of-the-art methods using the same amount of memory.

Index Terms—sketch, cardinality estimation, super spreader detection.

1 INTRODUCTION

Many real-world networks are given in the form of graph streams. Calling network is such an example with nodes representing users and an edge representing a call from one user to another. When web surfing activities are modeled as a bipartite graph stream where users are from one node partition and websites are from the other node partition, a user has edges to the websites that he/she browsed. Monitoring the cardinalities (or degrees) of users in these networks is fundamental for many applications such as network anomaly detection [15], [54], where a user's cardinality is defined to be the number of *distinct* users/items that the user connects to in the unipartite/bipartite graph stream of interest. Due to the large-size and high-speed nature of these graph streams, it is infeasible to collect the entire graph especially when the computational and memory resources are limited [44], [37]. For example, network routers have fast but very small

memories, which leads their traffic monitoring modules incapable to exactly compute the cardinalities of network users. Therefore, it is important to develop fast and memory-efficient algorithms to approximately compute user cardinalities over time.

However, existing methods have the following challenges:

Challenge 1: One can implement Bloom Filters [5] to handle edge duplicates and exploit frequency estimation methods such as Count-Min sketch [9] to approximate user cardinalities, but it requires large memory usage and results in large estimation errors. Compared with the methods such as Count-Min sketch [9] that utilize a variety of counters to record each user's frequency (i.e., the number of times the user occurs) over time, it is essential to build a hash table of distinct occurred edges such as a Bloom Filter [5] to handle edge duplicates in graph streams when computing user cardinalities. Specifically, a Bloom Filter is a one-dimensional array consisting of m bits and z independent hash functions. For each incoming edge, the Bloom Filter randomly selects z bits using z hash functions and sets them to one. Afterwards, to check whether an edge is presented previously, the Bloom Filter uses z hash functions to get z corresponding bits of the edge. An edge is treated as a duplicated edge if all these z associated bits are all set to one and a new edge otherwise. The key characteristic of the Bloom Filter is that it has a false positive but no false negative. The false positive is calculated as $p = (1 - (1 - 1/m)^{zn^{(t)}})^z$, where $n^{(t)}$ is the sum of all user cardinalities at time t . The more edges occur in graph streams, the larger false positive is caused. Therefore, it is necessary to assign a large memory space to the Bloom Filter to achieve the desired false positive. Also, the false positives of wrongly identified edges result in complex estimation errors in Count-Min sketch because the false positives are unknown in advance. Moreover, Count-Min usually provides over-estimations for user cardinalities, and it is easy to mis-classify users with small cardinalities as users with large cardinalities and generate large estimation bias.

- Peng Jia, Pinghui Wang, Yuchao Zhang, Jing Tao, and Xiaohong Guan are with MOE Key Laboratory for Intelligent Networks and Network Security, Xi'an Jiaotong University, Xi'an, Shaanxi, 710049, China. Tel.: +086-29-82664603; Fax: +086-29-82664603; E-mail: pengjia@sei.xjtu.edu.cn, {phwang, jtao, xhguan}@mail.xjtu.edu.cn, zhangyuchao01@stu.xjtu.edu.cn. Pinghui Wang, Jing Tao and Xiaohong Guan are also with Shenzhen Research Institute of Xi'an Jiaotong University, Shenzhen, China. Jing Tao is also with Zhejiang Research Institute of Xi'an Jiaotong University, Hangzhou, China. Xiaohong Guan is also with the Center for Intelligent and Networked Systems, Tsinghua National Lab for Information Science and Technology, Tsinghua University, Beijing, China. Xiangliang Zhang is with the King Abdullah University of Science and Technology, Thuwal 23955, Saudi Arabia. Email: xiangliang.zhang@kaust.edu.sa. Jianwei Ding is with the Science and Technology on Communication Security Laboratory, 30th Research Institute of China Electronics Technology Group Corporation, Chengdu, China. Email: mathe_007@163.com. Don Towsley is with the Department of Computer Science, University of Massachusetts, Amherst, MA 01003. Email: towsley@cs.umass.edu. Pinghui Wang is the corresponding author.

Challenge 2: *It is difficult to set an optimal value for the size of existing sketch methods.* A variety of cardinality estimation methods such as Linear-Time Probabilistic Counting (LPC) [48] and HyperLogLog (HLL) [17] are developed to approximately compute cardinalities. An LPC/HLL sketch consists of m bits/registers, where m is a parameter affecting the estimation accuracy. Since user cardinalities are not known in advance and change over time, one needs to set m large (e.g., thousand) to achieve reasonable accuracy for each user, whose cardinality may vary over a large range. However, this method wastes considerable memory because a large value of m is not necessary for most users, which have small cardinalities. To solve this problem, [54], [53], [46], [49] develop different virtual sketch methods to compress each user’s LPC/HLL sketch into a large bit/register array shared by all users. These virtual sketch methods build each user’s virtual LPC/HLL sketch using m bits/registers randomly selected from the large bit/register array. This significantly reduces memory usage because each bit/register may be used by more than one user. However, bits/registers in a user’s virtual LPC/HLL sketch may be contaminated by other users. We refer to these bits/registers as “noisy” bits/registers. In practice, most users have small cardinalities and their virtual LPC/HLL sketches tend to contain many “noisy” bits/registers, which results in large estimation errors.

Challenge 3: *Existing methods are customized to estimate user cardinalities after all the data has been observed and unable to report user cardinalities on the fly.* For real-time applications like online anomaly detection, it is important to track user cardinalities in real-time. For example, network monitoring systems are required to detect abnormal IP addresses such as super spreaders (i.e., IP addresses with cardinalities larger than a specified threshold) as soon as possible. Moreover, online monitoring of IP address cardinalities over time also facilitates online detection of stealthy attacks launched from a subclass of IP addresses.

To address the above challenges, we develop two novel streaming algorithms FreeBS and FreeRS to accurately estimate user cardinalities over time. In addition, in many cases such as anomaly detection, we mainly focus on abnormal users with extremely large cardinalities, and there is no need to cost a lot of memory resources to set a counter for each user to keep track of its cardinality. Therefore, we extend both our methods in combination with the *Unbiased Space Saving* method [43] to detect super spreaders in real-time. We summarize our main contributions as:

- Compared with previous bit and register sharing methods, our algorithms FreeBS and FreeRS exploit the dynamic properties of the bit/register array over time to significantly improve the estimation accuracy. The dynamic properties of the bit/register array can be summarized as: 1) The number of bits/registers used by a user dynamically increases as its cardinality increases over time and thus users with larger cardinalities tend to use more bits/registers; 2) The fraction of zeros bits in the bit array and the register values in the register array dynamically change over time. FreeBS and FreeRS incrementally estimate user cardinalities based on the probability of each new arriving user-item pair changing bits/registers in the bit/register array, which decreases over time.
- Our algorithms report user cardinality estimations on the fly and allow to track user cardinalities in real-time. The time complexity is reduced from $O(m)$ in state-of-the-art methods CSE [53] and vHLL [49] to $O(1)$ for updating user cardinality estimations each time they observe a new user-item pair.

- We extend our algorithms to detect super spreaders whose cardinalities rank top- K among all users in real-time, which is more accurate and faster than existing state-of-the-art methods.
- We evaluate the performance of our methods on real-world datasets. Experimental results demonstrate that our methods are orders of magnitude faster and up to 10,000 times more accurate than state-of-the-art methods using the same amount of memory.

The rest of this paper is organized as follows. Section 2 summarizes related work. The problem formulation is presented in Section 3. Section 4 introduces preliminaries. Section 5 presents our algorithms FreeBS and FreeRS. The extension of our methods to detect super spreaders is shown in Section 6. The performance evaluation and testing results are presented in Section 7. Conclusions and future works then follow.

2 RELATED WORK

Estimating data streams’ cardinalities. In addition to LPC [48] and HLL [17], there are also many methods developed to estimate cardinality for each user. In detail, [15], [7] combine LPC and different sampling methods to enlarge the estimation range of LPC. Flajolet and Martin [18] develop a sketch method FM, which uses a register to estimate the data stream’s cardinality bounded by 2^w , where w is the number of bits in the register. To further improve the accuracy of [18], sketch methods MinCount [3], LogLog [12], HyperLogLog++ [24], RoughEstimator [28], and HLL-TailCut+ [50] are developed to use a list of m registers and compress the size of each register from 32 bits to 5 or 4 bits. [20], [34] develop several cardinality estimators based on order statistics of observed samples. Ting [42] introduces the concept of an area cutting process to generally model the above sketch methods and provides a martingale based estimator to further improve the accuracy of these methods. Chen et.al [8] extend HLL to estimate the cardinality over sliding windows. Besides these sketch methods, Wegman’s adaptive sampling [16] and distinct sampling [19] are also developed for estimating a large stream’s cardinality. Furthermore, Harmouch et al. [23] summarize existing cardinality estimation methods and present experiments to evaluate their performance.

Estimating all user cardinalities. To achieve a desired accuracy, above methods require a large number of bits/registers for each user because user cardinalities are unknown in advance and vary over a large range. This is not memory-efficient because most users have small cardinalities. To solve this problem, [54], [53], [49], [46] develop different virtual sketch methods to compress the LPC/HLL sketches of all users into a large shared bit/register array shared by all users. To reduce “noisy” bits in virtual LPC sketches, [41] builds a small regular LPC sketch for each user, which also provides information for estimating the user cardinality. All these virtual sketch methods have to set a large value for the virtual sketch size to achieve desired accuracy for user cardinalities over a large range. However, it results in high computation costs and large estimation errors, because most users have small cardinalities and many bits/registers in their virtual sketches tend to be contaminated by “noisy” bits. In addition, the above virtual sketch methods are customized to non-streaming settings, i.e., estimate host cardinalities at the end of an interval, and are computationally expensive to be extended to streaming settings.

Detecting Heavy Hitters or Super Spreads. Heavy hitter and super spreader are two different but important anomaly user behaviors. Usually heavy hitters are defined as users whose frequencies are larger than a given threshold, while super spreaders are

defined as users whose cardinalities exceed a specific threshold. To detect heavy hitters, Graham et al. [10] present both *deterministic sample* and *randomized sketch* algorithms to find heavy hitters in graph streams consisting of item insertions and deletions. [11] calculates heavy hitters in multiple dimensions at a high speed with very small memory space. Cristian et al. [13], [14] present two algorithms *sample and hold* and *multistage filters* to detect heavy hitters without storing entire small network flows. Also, [33], [26], [30], [52] develop general sketch algorithms to detect a variety of network measurements at the same time. Specially, [52] is faster and more memory efficient because it divides users into a heavy part and a light part according to their frequencies, which is consistent with the basic idea in [51]. Moreover, to detect top- K heavy hitters, Metwally et al. [35] present a *Stream-Summary* data structure but it is biased for each arriving element. Ting [43] later proposes a probabilistic model to replace the minimum element in the data structure with a varied probability to reduce the estimation bias. The policy of updating the minimum counter in [43] is consistent with [4], which is developed for solving frequent and top- K heavy hitter estimation problems. In addition, Basat et al. [4] further propose a hardware-friendly method by dividing the logical locations into a variety of sets containing a fixed number of elements. Gong et al. [21] introduce a novel strategy *count-with-exponential-decay* to achieve high precision in finding top- K heavy hitters, but it is not memory-efficient in comparison with [35], [43]. Schweller et al. [38] develop reversible sketch data structure and reverse hashing to monitor users whose frequencies change significantly from one period to another.

When there exist edge duplicates in graph streams, the heavy hitter estimation methods cannot be directly used to detect super spreaders, and one requires to use a hash table to store distinct items. Guan et al. [22] present a connection degree sketch based method and use *Chinese Remainder Theorem* to monitor and reverse super spreaders. This method is also implemented in [46], [47]. Besides, Liu et al. [32] propose a novel mergeable and reversible data structure for distributed network monitoring system to detect super spreaders. [31] further introduces a *Vector Bloom Filter* (VBF), which uses *Bit-Extraction* hash function to improve the cardinality estimation with lower memory usage and less computation time. Although above methods can utilize well-designed hash functions to reverse users without storing them, the computation costs are large because they require to enumerate and combine almost all possible hash values to reconstruct users, which is not practicable to detect super spreaders over time. Also, these methods always return false users not actually occurred in graph streams and introduce large estimation errors. In addition, there are also many sampling based methods to detect super spreaders. Venkataraman et al. [45] first propose flow sampling based streaming algorithms to detect super spreaders and provide proven accuracy and performance bounds. [27] further presents adaptive sampling methods to detect super spreaders in real time. Cao et al. [6] show an online sampling approach to identify super spreaders whose cardinalities exceed a threshold or within specific ranges. Tang et al. [40] propose a reservoir sampling based method but it is biased. Shi et al. [39] combine sampling and streaming algorithms and introduce a “Filter-Tracker-Digester” framework to catch top spreaders and scanners online. All these sampling methods first filter out users with small cardinalities when implemented to detect super spreaders, but they cannot guarantee all super spreaders are sampled and exhibit large errors using their cardinality estimation methods.

3 PROBLEM FORMULATION

To formally define our problem, we first introduce some notations. Let $\Gamma = e^{(1)}e^{(2)}\dots$ be the graph stream of interest consisting of a sequence of edges. Note that an edge in Γ may appear more than once. In this paper, we focus on bipartite graph streams consisting of edges between users and items. Our methods, however, can be easily extended to unipartite graphs such as computer networks where each edge represents the network communication between a source IP address and a destination IP address, and online social networks where each edge represents the social relation between a user and his/her friend. Let S and D denote the user and item sets respectively. For $t = 1, 2, \dots$, let $e^{(t)} = (s^{(t)}, d^{(t)})$ denote the t^{th} edge of Γ , where $s^{(t)} \in S$ and $d^{(t)} \in D$ are the user and the item of $e^{(t)}$ respectively. Let $N_s^{(t)}$ denote the set of distinct items that user s connects to before and including time t . Define $n_s^{(t)} = |N_s^{(t)}|$ to be the cardinality of user s at time t . Then, $n^{(t)} = \sum_{s \in S} |N_s^{(t)}|$ is the sum of all user cardinalities at time t . In this paper, we develop fast and accurate methods for estimating user cardinalities at times $t = 1, 2, \dots$ using a limited amount of memory. Besides, we aim to monitor top- K super spreaders based on the cardinality estimation results. When no confusion arises, we omit the superscript (t) to ease exposition.

4 PRELIMINARIES

4.1 Estimating a Single User’s Cardinality

4.1.1 Linear-Time Probabilistic Counting

For a user $s \in S$, Linear-Time Probabilistic Counting (LPC) [48] builds a sketch B_s to store the set of items that s connects to, i.e., $N_s^{(t)}$. Formally, B_s is defined as a bit array consisting of m bits, which are initialized to 0. Let $h(d)$ be a uniform hash function with range $\{1, \dots, m\}$. When user-item pair (s, d) arrives, the $h(d)^{\text{th}}$ bit in B_s is set to one, i.e., $B_s[h(d)] = 1$. For any bit $B_s[i]$, $1 \leq i \leq m$, the probability that it remains zero at time t is $P(B_s[i] = 0) = (1 - \frac{1}{m})^{n_s^{(t)}}$. Denote by $U_s^{(t)}$ the number of zero bits in B_s at time t . Then, the expectation of $U_s^{(t)}$ is computed as $\mathbb{E}(U_s^{(t)}) = \sum_{i=1}^m P(B_s[i] = 0) \approx me^{-\frac{n_s^{(t)}}{m}}$. Based on the above equation, when $U_s^{(t)} > 0$, Whang et al. [48] estimate $n_s^{(t)}$ as $\hat{n}_s^{(t, \text{LPC})} = -m \ln \frac{U_s^{(t)}}{m}$. The weakness of LPC is that it has a limited estimation range and can only estimate user cardinalities in range $[0, m \ln m]$.

4.1.2 HyperLogLog

To estimate the cardinality of user s , HyperLogLog (HLL) [17] is developed based on the Flajolet-Martin (FM) sketch [18] R_s consisting of m registers $R_s[1], \dots, R_s[m]$. All m registers are initialized to 0. For $1 \leq i \leq m$, let $R_s^{(t)}[i]$ be the value of $R_s[i]$ at time t . When a user-item pair (s, d) arrives, HLL maps the item into a pair of random variables $h(d)$ and $\rho(d)$, where $h(d)$ is an integer uniformly selected from $\{1, \dots, m\}$ at random and $\rho(d)$ is drawn from a *Geometric*(1/2) distribution, $P(\rho(d) = k) = \frac{1}{2^k}$ for $k = 1, 2, \dots$ ¹. Then, register $R_s[h(d)]$ is updated as $R_s^{(t)}[h(d)] \leftarrow \max\{R_s^{(t-1)}[h(d)], \rho(d)\}$. At time t , Flajolet et al. [17] estimate $n_s^{(t)}$ as $\hat{n}_s^{(t, \text{HLL})} = \frac{\alpha_m m^2}{\sum_{i=1}^m 2^{-R_s^{(t)}[i]}}$, where $\alpha_m =$

1. Functions $h(d)$ and $\rho(d)$ are implemented as: Let $\Phi(d) = \langle x_1 x_2 \dots \rangle$ be the binary format of the output of a hash function $\Phi(d)$, and $b = \lceil \log_2 m \rceil$. Then, $h(d)$ is defined as $h(d) = (x_1 x_2 \dots x_b \bmod m) + 1$ and $\rho(d)$ is defined as the number of leading zeros in $\langle x_{b+1} x_{b+2} \dots \rangle$ plus one.

Figure 1: Overview of bit sharing method CSE and register sharing method vHLL. Virtual CSE/vHLL sketches of users may contain “noisy” bits/registers (e.g., the bit and register in red and bold in the figure).

$\left(m \int_0^\infty \left(\log_2 \frac{2+x}{1+x}\right)^m dx\right)^{-1}$ is a constant to correct bias. The above formula for α_m is complicated. In practice, α_m is computed numerically, e.g., $\alpha_{16} \approx 0.673$, $\alpha_{32} \approx 0.697$, $\alpha_{64} \approx 0.709$, and $\alpha_m \approx 0.7213/(1 + 1.079/m)$ for $m \geq 128$.

Since $\hat{n}_s^{(t, \text{HLL})}$ is severely biased for small cardinalities, HLL treats R_s as an LPC sketch of m bits when $\frac{\alpha_m m^2}{\sum_{i=1}^m 2^{-R_s^{(t)}[i]}} < 2.5m$. In this case, $n_s^{(t)}$ is estimated as $-m \ln \frac{\hat{U}_s^{(t)}}{m}$, where $\hat{U}_s^{(t)}$ is the number of zero registers among $R_s[1], \dots, R_s[m]$ at time t . Therefore, we easily find that LPC outperforms HLL for small cardinalities under the same memory usage.

4.1.3 Discussions

To compute all user cardinalities, one can use an LPC or HLL sketch to estimate each user cardinality. Clearly, using a small m , LPC and HLL will exhibit large errors for users with large cardinalities. Most user cardinalities are small and assigning an LPC/HLL sketch with large m to each user in order to accurately estimate large user cardinalities is wasteful as LPC and HLL do not require to set a large m to achieve reasonable estimation accuracy for users with small cardinalities. In practice, the user cardinalities are not known in advance and vary over time. Therefore, it is difficult to set an optimal value of m when using LPC and HLL to estimate all user cardinalities. In the next subsection, we introduce state-of-the-art methods to address this problem, and also discuss their shortcomings.

4.2 Estimating All User Cardinalities

4.2.1 CSE: Compressing LPC Sketches of All Users into a Shared Bit Array

As shown in Figure 1 (a), CSE [53] consists of a large bit array A and m independent hash functions $f_1(s), \dots, f_m(s)$, each mapping users to $\{1, \dots, M\}$, where M is the length of the one dimensional bit array A . Similar to LPC, CSE builds a virtual LPC sketch for each user and embeds LPC sketches of all users in A . For user s , its virtual LPC sketch $\hat{B}_s$ consists of m bits selected randomly from A by the group of hash functions $f_1(s), \dots, f_m(s)$, i.e., $\hat{B}_s = (A[f_1(s)], \dots, A[f_m(s)])$. Each bit in A is initially set to zero. When a user-item pair (s, d) arrives, CSE sets the $h(d)$ th bit in $\hat{B}_s$ to one. Similar to LPC, $h(d)$ is a uniform hash function with range $\{1, \dots, m\}$. Since the $h(d)$ th element in $\hat{B}_s$ is bit $A[f_{h(d)}(s)]$, CSE only needs to set bit $A[f_{h(d)}(s)]$, i.e.,

$A[f_{h(d)}(s)] \leftarrow 1$. Let $\hat{U}_s^{(t)}$ and $U^{(t)}$ be the number of zero bits in $\hat{B}_s$ and A at time t respectively. A user’s virtual LPC sketch can be viewed as a regular LPC sketch containing “noisy” bits (e.g., the bit in red and bold in Figure 1 (a)), that are wrongly set from zero to one by items of other users. To remove the estimation error introduced by “noisy” bits, Yoon et al. [53] estimate $n_s^{(t)}$ as $\hat{n}_s^{(t, \text{CSE})} = -m \ln \frac{\hat{U}_s^{(t)}}{m} + m \ln \frac{U^{(t)}}{M}$. On the right-hand side of the above equation, the first term is the same as the regular LPC, and the second term corrects the error introduced by “noisy” bits.

4.2.2 vHLL: Compressing HLL Sketches of All Users into a Shared Bit Array

Xiao et al. [49] develop a register sharing method, vHLL, which extends the HLL method to estimate all user cardinalities. vHLL consists of a list of M registers $R[1], \dots, R[M]$, which are initialized to zero. To maintain the virtual HLL sketch $\hat{R}_s$ of a user s , vHLL uses m independent hash functions $f_1(s), \dots, f_m(s)$ to randomly select m registers from all registers $R[1], \dots, R[M]$, where each function $f_1(s), \dots, f_m(s)$ maps users to $\{1, \dots, M\}$. Formally, $\hat{R}_s$ is defined as $\hat{R}_s = (R[f_1(s)], \dots, R[f_m(s)])$.

For $1 \leq i \leq M$, let $R^{(t)}[i]$ be the value of $R[i]$ at time t . When a user-item pair (s, d) arrives, it maps the item to a pair of two random variables $h(d)$ and $\rho(d)$, where $h(d)$ is an integer uniformly selected from $\{1, \dots, m\}$ at random, and $\rho(d)$ is a random integer drawn from a *Geometric*(1/2) distribution, which is similar to HLL. We can easily find that the $h(d)$ th element in the virtual HLL sketch of user s is $R[f_{h(d)}(s)]$, therefore, vHLL only needs to update register $R[f_{h(d)}(s)]$ as $R^{(t)}[f_{h(d)}(s)] \leftarrow \max\{R^{(t-1)}[f_{h(d)}(s)], \rho(d)\}$. A user’s virtual HLL sketch can be viewed as a regular HLL containing “noisy” registers (e.g., the register in red and bold in Figure 1 (b)), which are wrongly set by items of other users. To remove the estimation error introduced by “noisy” registers, Xiao et al. [49] estimate $n_s^{(t)}$ as $\hat{n}_s^{(t, \text{vHLL})} = \frac{M}{M-m} \left(\frac{\alpha_m m^2}{\sum_{i=1}^m 2^{-R^{(t)}[f_i(s)]}} - \frac{m \alpha_m M}{\sum_{i=1}^M 2^{-R^{(t)}[i]}} \right)$, where α_m is the same as that of HLL. For the two terms between the parentheses on the right-hand side of the above equation, the first term is the same as the regular HLL, and the second term corrects the error introduced by “noisy” registers. Similar to the regular HLL, the first term is replaced by $-m \ln \frac{\hat{U}_s^{(t)}}{m}$ when $\frac{\alpha_m m^2}{\sum_{i=1}^m 2^{-R^{(t)}[f_i(s)]}} < 2.5m$ where $\hat{U}_s^{(t)}$ is the number of registers among $R[f_1(s)], \dots, R[f_m(s)]$ that equal 0 at time t .

4.2.3 Discussions

LPC/HLL builds a single bit/register array for each user and each user cardinality is not affected by others. To achieve the desired estimation accuracy, LPC/HLL requires to set a large size m for each user and it results in a waste of memory space especially for users with small cardinalities. On the contrary, CSE/vHLL is more memory-efficient because it allows each bit/register in the shared bit/register array to be used by other users, but it cannot solve the following challenges:

Challenge 1: It is difficult to set parameter m for both CSE and vHLL. The estimation accuracy of CSE and vHLL highly depends on the value of m . Increasing m introduces more “unused” bits in virtual LPC sketches of occurred users, which can become contaminated with noise. Here “unused” bits refer to the bits in a user’s virtual LPC sketch that no user-item pairs of the user are hashed into. However, decreasing m introduces large estimation errors for users with large cardinalities. Similarly, vHLL also confronts the same challenge in determining an optimal m . Later our experimental results will also verify that errors increase with m for users with small cardinalities under CSE and vHLL.

Challenge 2: It is computationally intensive to estimate user cardinalities for all values of t . At each time, both CSE and vHLL require time complexity $O(m)$ to compute the cardinality of a single user. When applied to compute cardinalities for all users in S at all times, CSE and vHLL have to be repeatedly called and will incur a high computational cost, which prohibits their application to high speed streams in an on-line manner.

5 OUR METHODS

In this section, we present our streaming algorithms FreeBS and FreeRS for estimating user cardinalities over time, which are designed based on two novel bit sharing and register sharing techniques respectively. It is not necessary to assign more bits/registers for users with small cardinalities because there will not be a significant improvement for their cardinality estimation accuracy. There should be more bits/registers provided for users with large cardinalities to ensure estimation performance. However, existing sharing methods waste bits/registers for users with small cardinalities while limiting the estimation accuracy for users with large cardinalities. Therefore, unlike vHLL/CSE mapping each user’s items into $m \ll M$ bits/registers, FreeBS/FreeRS randomly maps user-item pairs into a large shared bit/register array. The corresponding bits/registers of a user are sampled according to its cardinality (i.e., the number of connected items). Thus, users with larger cardinalities tend to use more bits/registers. For each user-item pair $e^{(t)} = (s^{(t)}, d^{(t)})$ occurred at time t , we discard it when updating $e^{(t)}$ does not change any bit/register in the bit/register array shared by all users. Otherwise, $e^{(t)}$ is a new user-item pair that does not occur before time t , and we increase the cardinality estimation of user $s^{(t)}$ by $\frac{1}{q^{(t)}}$ [25], where $q^{(t)}$ is defined as the probability that a new user-item pair changes any bit/register in the bit/register array at time t .

5.1 FreeBS: Parameter-Free Bit Sharing

Data Structure. The data structure of FreeBS is shown in Figure 2. FreeBS consists of a one-dimensional bit array B of length M , where each bit $B[j]$, $1 \leq j \leq M$, is initialized to zero. In addition, FreeBS uses a hash function $h^*(e)$ to uniformly and independently map each user-item pair $e = (s, d)$ to an

integer in $\{1, \dots, M\}$ at random, i.e., $P(h^*(e) = i) = \frac{1}{M}$, and $P(h^*(e) = i \wedge h^*(e') = i' | e \neq e') = \frac{1}{M^2}$, $i, i' \in \{1, \dots, M\}$. Note that $h^*(e)$ differs from the hash function $h(s)$ used by CSE that maps user s to an integer in $\{1, \dots, M\}$ at random.

Figure 2: Overview of our method FreeBS.

Algorithm 1: The pseudo-code for FreeBS.

```

 $B[1, \dots, M] \leftarrow [0, \dots, 0];$ 
 $\hat{n}_s^{(\text{FreeBS})} \leftarrow 0, s \in S; \quad m_0 \leftarrow M;$ 
foreach  $e = (s, d)$  in  $\Gamma$  do
    if  $B[h^*(e)] == 0$  then
         $B[h^*(e)] \leftarrow 1;$ 
         $\hat{n}_s^{(\text{FreeBS})} \leftarrow \hat{n}_s^{(\text{FreeBS})} + \frac{M}{m_0};$ 
         $m_0 \leftarrow m_0 - 1;$ 
    end
end

```

Update Procedure. The pseudo-code for FreeBS is shown as Algorithm 1. When a user-item pair $e = (s, d)$ arrives at time t , FreeBS first computes a random variable $h^*(e)$, and then sets $B[h^*(e)]$ to one, i.e., $B[h^*(e)] \leftarrow 1$. Let $B_0^{(t)}$ denote the set of the indices corresponding to zero bits in B at time t . Formally, $B_0^{(t)}$ is defined as $B_0^{(t)} = \{i : B^{(t)}[i] = 0, 1 \leq i \leq M\}$. Let $\hat{n}_s^{(t, \text{FreeBS})}$ denote the cardinality estimate for user s at time t . We initialize $\hat{n}_s^{(0, \text{FreeBS})}$ to 0. Let $m_0^{(t)} = |B_0^{(t)}|$ denote the number of zero bits in B at time t . Let $q_B^{(t)}$ denote the probability of e changing a bit in B from 0 to 1 at time t , and $q_B^{(t)}$ is defined as $q_B^{(t)} = \sum_{i \in B_0^{(t-1)}} P(h^*(e) = i) = \frac{|B_0^{(t-1)}|}{M} = \frac{m_0^{(t-1)}}{M}$. Let $\mathbf{1}(\mathbb{P})$ denote the indicator function that equals 1 when predicate $\mathbb{P}$ is true and 0 otherwise. Besides setting $B[h^*(e)]$ to 1 at time t with the arrival of user-item pair $e = (s, d)$, we also update the cardinality estimate of user s as

$$\hat{n}_s^{(t, \text{FreeBS})} \leftarrow \hat{n}_s^{(t-1, \text{FreeBS})} + \frac{\mathbf{1}(B^{(t-1)}[h^*(e)] = 0)}{q_B^{(t)}}.$$

For any other user $s' \in S \setminus \{s\}$, we keep its cardinality estimate unchanged, i.e., $\hat{n}_{s'}^{(t, \text{FreeBS})} \leftarrow \hat{n}_{s'}^{(t-1, \text{FreeBS})}$. We easily find that $q_B^{(t)}$ can be fast computed incrementally. That is, we initialize $q_B^{(1)}$ to 1, and incrementally compute $q_B^{(t+1)}$ as $q_B^{(t+1)} \leftarrow q_B^{(t)} - \frac{\mathbf{1}(B^{(t-1)}[h^*(e)] = 0)}{M}, t \geq 1$. Hence, the time complexity of FreeBS for processing each user-item pair is $O(1)$.

Error Analysis. Let $T_s^{(t)}$ denote the set of the first occurrence times of user-item pairs associated with user s in stream Γ , i.e., $T_s^{(t)} = \{i : s^{(i)} = s \wedge e^{(j)} \neq e^{(i)}, 0 < j < i \leq t\}$.

Theorem 1. The expectation and variance of $\hat{n}_s^{(t, \text{FreeBS})}$ are

$$\mathbb{E}(\hat{n}_s^{(t, \text{FreeBS})}) = n_s^{(t)},$$

$$\text{Var}(\hat{n}_s^{(t, \text{FreeBS})}) = \sum_{i \in T_s^{(t)}} \mathbb{E}\left(\frac{1}{q_B^{(i)}}\right) - n_s^{(t)},$$

$$\text{where } \mathbb{E}\left(\frac{1}{q_B^{(i)}}\right) \approx e^{\frac{n^{(i)}}{M}} \left(1 + \frac{1}{M} \left(e^{\frac{n^{(i)}}{M}} - \frac{n^{(i)}}{M} - 1\right)\right).$$

Proof. Let δ_e denote an indicator that equals 1 when updating a user-item pair e incurs a value change of $B[h^*(e)]$ (i.e., $B[h^*(e)]$ changes from 0 to 1), and 0 otherwise. Let $\eta^{(i)}$ denote a random variable that represents the update probability of an arriving user-item pair $e = (s, d)$ changing a bit in the bit array from 0 to 1 at time i . Each $\delta_{e^{(i)}}$ is calculated based on the current bit array (i.e., the probability $q_B^{(i)}$). Then we have

$$\mathbb{E}(\delta_{e^{(i)}} | \eta^{(i)} = q_B^{(i)}) = q_B^{(i)},$$

$$\text{Var}(\delta_{e^{(i)}} | \eta^{(i)} = q_B^{(i)}) = q_B^{(i)} - (q_B^{(i)})^2, \quad 1 \leq i \leq t.$$

$\hat{n}_s^{(t, \text{FreeBS})}$ is incrementally calculated based on all random variables $\delta_{e^{(i)}}$ and their corresponding probabilities $q_B^{(i)}$, $i \in T_s^{(t)}$. $\hat{n}_s^{(t, \text{FreeBS})} = \sum_{i \in T_s^{(t)}} \frac{\delta_{e^{(i)}}}{q_B^{(i)}}$. Then, we have

$$\begin{aligned} \mathbb{E}(\hat{n}_s^{(t, \text{FreeBS})} | \eta^{(1)} = q_B^{(1)}, \dots, \eta^{(t)} = q_B^{(t)}) \\ = \mathbb{E}\left(\sum_{i \in T_s^{(t)}} \frac{\delta_{e^{(i)}}}{q_B^{(i)}} \middle| \eta^{(1)} = q_B^{(1)}, \dots, \eta^{(t)} = q_B^{(t)}\right) \\ = \sum_{i \in T_s^{(t)}} \frac{\mathbb{E}(\delta_{e^{(i)}} | \eta^{(1)} = q_B^{(1)}, \dots, \eta^{(t)} = q_B^{(t)})}{q_B^{(i)}} \\ = n_s^{(t)}. \end{aligned}$$

$\mathbb{E}(\delta_{e^{(i)}} | \eta^{(1)} = q_B^{(1)}, \dots, \eta^{(t)} = q_B^{(t)}) = \mathbb{E}(\delta_{e^{(i)}} | \eta^{(i)} = q_B^{(i)}) = q_B^{(i)}$. This is because all random variables $\delta_{e^{(i)}}$, $i \in T_s^{(t)}$, are independent of each other and each of them only has association with the variable $\eta^{(i)}$ at the current time when given a sequence of update probabilities $\eta^{(1)}, \dots, \eta^{(t)}$. Then, we have

$$\begin{aligned} \mathbb{E}(\hat{n}_s^{(t, \text{FreeBS})}) &= \mathbb{E}(\mathbb{E}(\hat{n}_s^{(t, \text{FreeBS})} | \eta^{(1)} = q_B^{(1)}, \dots, \eta^{(t)} = q_B^{(t)})) \\ &= \mathbb{E}(n_s^{(t)}) = n_s^{(t)}. \end{aligned}$$

The variance of $\hat{n}_s^{(t, \text{FreeBS})}$ is computed as

$$\begin{aligned} \text{Var}(\hat{n}_s^{(t, \text{FreeBS})} | \eta^{(1)} = q_B^{(1)}, \dots, \eta^{(t)} = q_B^{(t)}) \\ = \text{Var}\left(\sum_{i \in T_s^{(t)}} \frac{\delta_{e^{(i)}}}{q_B^{(i)}} \middle| \eta^{(1)} = q_B^{(1)}, \dots, \eta^{(t)} = q_B^{(t)}\right) \\ = \sum_{i \in T_s^{(t)}} \frac{\text{Var}(\delta_{e^{(i)}} | \eta^{(1)} = q_B^{(1)}, \dots, \eta^{(t)} = q_B^{(t)})}{(q_B^{(i)})^2} \\ = \sum_{i \in T_s^{(t)}} \frac{q_B^{(i)} - (q_B^{(i)})^2}{(q_B^{(i)})^2} = \sum_{i \in T_s^{(t)}} \frac{1}{q_B^{(i)}} - n_s^{(t)}. \end{aligned}$$

Since $\text{Var}(\mathbb{E}(\hat{n}_s^{(t, \text{FreeBS})} | \eta^{(1)} = q_B^{(1)}, \dots, \eta^{(t)} = q_B^{(t)})) = 0$, using the equation $\text{Var}(X) = \text{Var}(\mathbb{E}(X|Y)) + \mathbb{E}(\text{Var}(X|Y))$, we have

$$\begin{aligned} \text{Var}(\hat{n}_s^{(t, \text{FreeBS})}) &= \mathbb{E}(\text{Var}(\hat{n}_s^{(t, \text{FreeBS})} | \eta^{(1)} = q_B^{(1)}, \dots, \eta^{(t)} = q_B^{(t)})) \\ &= \mathbb{E}\left(\sum_{i \in T_s^{(t)}} \frac{1}{q_B^{(i)}} - n_s^{(t)}\right) \\ &= \sum_{i \in T_s^{(t)}} \mathbb{E}\left(\frac{1}{q_B^{(i)}}\right) - n_s^{(t)}. \end{aligned}$$

In what follows we derive the formula for $\mathbb{E}(\frac{1}{q_B^{(i)}})$. For j specific distinct bits in B , there exist $j! \tau(n^{(i)}, j)$ ways to map $n^{(i)}$ distinct user-item pairs occurred in stream Γ before and including time i into these bits given that each bit has at least one user-item pair, where $\tau(n^{(i)}, j)$, the Stirling number of the second kind [2], is computed as $\tau(n^{(i)}, j) = \sum_{k=0}^{j-1} (-1)^k \binom{j}{k} (j-k)^{n^{(i)}}$, $0 < j \leq n^{(i)}$. In addition, there exist $\binom{M}{j}$ ways to select j distinct bits from B , therefore we have $P(m_0^{(i)} = M - j | n^{(i)}) = \frac{\binom{M}{j} j! \tau(n^{(i)}, j)}{M^{n^{(i)}}}$. Then, the expected value of $\frac{1}{q_B^{(i)}}$ can be computed as

$$\mathbb{E}\left(\frac{1}{q_B^{(i)}}\right) = \sum_{j=1}^M \frac{M \binom{n^{(i)}}{j} \sum_{k=0}^{j-1} (-1)^k \binom{j}{k} (j-k)^{n^{(i)}}}{M - j}.$$

We expand the function $\frac{1}{q_B^{(i)}}$ by its Taylor series around $\mathbb{E}(q_B^{(i)})$ as

$$\begin{aligned} \mathbb{E}\left(\frac{1}{q_B^{(i)}}\right) &\approx \mathbb{E}\left(\frac{1}{\mathbb{E}(q_B^{(i)})} - \frac{q_B^{(i)} - \mathbb{E}(q_B^{(i)})}{(\mathbb{E}(q_B^{(i)}))^2} + \frac{(q_B^{(i)} - \mathbb{E}(q_B^{(i)}))^2}{(\mathbb{E}(q_B^{(i)}))^3}\right) \\ &= \frac{1}{\mathbb{E}(q_B^{(i)})} + \frac{\text{Var}(q_B^{(i)})}{(\mathbb{E}(q_B^{(i)}))^3}. \end{aligned}$$

From eqs.(5) and (6) in [48], we easily have $\mathbb{E}(q_B^{(i)}) = e^{-\frac{n^{(i)}}{M}}$ and $\text{Var}(q_B^{(i)}) = \frac{1}{M} e^{-\frac{n^{(i)}}{M}} (1 - (1 + \frac{n^{(i)}}{M}) e^{-\frac{n^{(i)}}{M}})$. Then, we obtain $\mathbb{E}(q_B^{(i)}) \approx e^{\frac{n^{(i)}}{M}} (1 + \frac{1}{M} (e^{\frac{n^{(i)}}{M}} - \frac{n^{(i)}}{M} - 1))$. $\square$

5.2 FreeRS: Parameter-Free Register Sharing

Data Structure. The data structure of FreeRS is shown in Figure 3. FreeRS consists of M registers $R[1], \dots, R[M]$, which are initialized to zero. In addition, FreeRS also uses a hash function $h^*(e)$ to randomly map each user-item pair $e = (s, d)$ to an integer in $\{1, \dots, M\}$ and another function $\rho^*(e)$ that maps e to a random integer in $\{1, 2, \dots\}$ according to a *Geometric*(1/2) distribution. Note that $h^*(e)$ and $\rho^*(e)$ differ from hash functions $h(s)$ and $\rho(s)$ used by vHLL, which map **user** u to a random integer in $\{1, \dots, M\}$ and $\{1, 2, \dots\}$, respectively.

Figure 3: Overview of our method FreeRS.

Update Procedure. The pseudo-code for FreeRS is shown as Algorithm 2. When user-item pair $e = (s, d)$ arrives at time t , FreeRS first computes two random variables $h^*(e)$ and $\rho^*(e)$, and then updates $R^{(t)}[h^*(e)]$ as $R^{(t)}[h^*(e)] \leftarrow \max\{R^{(t-1)}[h^*(e)], \rho^*(e)\}$. Let $q_R^{(t)}$ denote the probability of e changing the value of a register among $R[1], \dots, R[M]$ at time t . Formally, $q_R^{(t)}$ is defined as $q_R^{(t)} = \sum_{j=1}^M P(h^*(e) = j \wedge R^{(t)}[j] > R^{(t-1)}[j]) = \frac{\sum_{j=1}^M 2^{-R^{(t-1)}[j]}}{M}$.

Algorithm 2: The pseudo-code for FreeRS.

```

 $R[1, \dots, M] \leftarrow [0, \dots, 0];$ 
 $\hat{n}_s^{(\text{FreeRS})} \leftarrow 0, s \in S; \quad q_R \leftarrow 1;$ 
foreach  $e = (s, d) \in \Gamma$  do
  if  $\rho^*(e) > R[h^*(e)]$  then
     $\hat{n}_s^{(\text{FreeRS})} \leftarrow \hat{n}_s^{(\text{FreeRS})} + \frac{1}{q_R};$ 
     $q_R \leftarrow q_R + \frac{2^{-\rho^*(e)} - 2^{-R[h^*(e)]}}{M};$ 
     $R[h^*(e)] \leftarrow \rho^*(e);$ 
  end
end

```

Let $\hat{n}_s^{(t, \text{FreeRS})}$ denote the cardinality estimate of user s at time t . When user-item pair $e = (s, d)$ arrives at time t , we update the cardinality estimate of user s as

$$\hat{n}_s^{(t, \text{FreeRS})} \leftarrow \hat{n}_s^{(t-1, \text{FreeRS})} + \frac{\mathbf{1}(R^{(t)}[h^*(e)] \neq R^{(t-1)}[h^*(e)])}{q_R^{(t)}}.$$

For any other user $s' \in S \setminus \{s\}$, we keep its cardinality estimate unchanged, i.e., $\hat{n}_{s'}^{(t, \text{FreeRS})} \leftarrow \hat{n}_{s'}^{(t-1, \text{FreeRS})}$. Similar to $q_B^{(t)}$, we initialize $q_R^{(1)} = 1$ and incrementally compute $q_R^{(t+1)}$ as $q_R^{(t+1)} \leftarrow q_R^{(t)} + \frac{2^{-\rho^*(e)} - 2^{-R[h^*(e)]}}{m} \mathbf{1}(R^{(t)}[h^*(e)] \neq R^{(t-1)}[h^*(e)])$. Hence, the time complexity of FreeRS for processing each user-item pair is also $O(1)$.

Error Analysis. We derive the error of $\hat{n}_s^{(t, \text{FreeRS})}$ as follows:

Theorem 2. The expectation and variance of $\hat{n}_s^{(t, \text{FreeRS})}$ are

$$\mathbb{E}(\hat{n}_s^{(t, \text{FreeRS})}) = n_s^{(t)},$$

$$\text{Var}(\hat{n}_s^{(t, \text{FreeRS})}) = \sum_{i \in T_s^{(t)}} \mathbb{E}\left(\frac{1}{q_R^{(i)}}\right) - n_s^{(t)},$$

where

$$\mathbb{E}\left(\frac{1}{q_R^{(i)}}\right) = \sum_{k_1, \dots, k_M \geq 0} \frac{\sum_{n_1 + \dots + n_M = n^{(i)}} \binom{n^{(i)}}{n_1, \dots, n_M} \prod_{j=1}^M \gamma_{n_j, k_j}}{M^{n^{(i)}-1} \sum_{j=1}^M 2^{-k_j}},$$

with

$$\gamma_{n_j, k_j} = \begin{cases} (1-2^{-k_j})^{n_j} - (1-2^{-k_j+1})^{n_j}, & n_j > 0, k_j > 0 \\ 0, & n_j > 0, k_j = 0 \\ 1, & n_j = 0, k_j = 0. \end{cases}$$

$\mathbb{E}(\frac{1}{q_R^{(i)}})$ is approximately $\frac{1.386n^{(i)}}{M}$ when $n^{(i)} > 2.5M$.

Proof. The proof of the above theorem is similar with that of Theorem 1. Here we derive the formula for $\mathbb{E}(\frac{1}{q_R^{(i)}})$. Using $h^*(\cdot)$, FreeRS randomly splits stream Γ into M sub-streams Γ_j , $1 \leq j \leq M$. Each $R[j]$ tracks the maximum value of function $\rho^*(\cdot)$ for user-item pairs in sub-stream Γ_j . At time i , assume that n_j distinct user-item pairs have occurred in Γ_j . Then, $P(R^{(i)}[j] = k_j | n_j) = \gamma_{n_j, k_j}$. Therefore,

$$P(R^{(i)}[1] = k_1, \dots, R^{(i)}[M] = k_M | n^{(i)}) = \frac{\sum_{n_1 + \dots + n_M = n^{(i)}} \binom{n^{(i)}}{n_1, \dots, n_M} \prod_{j=1}^M \gamma_{n_j, k_j}}{M^{n^{(i)}}}.$$

An exact expression for $\mathbb{E}(\frac{1}{q_R^{(i)}})$ is easily derived. However, it is too complex to analyze. Hence, we introduce a method to approximate $\mathbb{E}(\frac{1}{q_R^{(i)}})$. From [17], we have $\mathbb{E}(\frac{\alpha_M M}{q_R^{(i)}}) = \alpha_M M \mathbb{E}(\frac{1}{q_R^{(i)}}) \approx n^{(i)}$ for $n^{(i)} > 2.5M$. Thus, $\mathbb{E}(\frac{1}{q_R^{(i)}}) \approx \frac{n^{(i)}}{\alpha_M M} \approx \frac{1.386n^{(i)}}{M}$. $\square$

5.3 Discussions

FreeBS vs CSE. FreeBS outperforms CSE in three aspects: (1) FreeBS can estimate cardinalities up to $\sum_{i=1}^M \frac{M}{i} \approx M \ln M$, which is larger than the maximum cardinality $m \ln m$ allowed by CSE; (2) FreeBS exhibits a smaller estimation error than CSE. The expectation and variance of CSE are computed in [41] as

$$\mathbb{E}(\hat{n}_s^{(t, \text{CSE})}) \approx n_s^{(t)} + \frac{1}{2} \left(\mathbb{E}\left(\frac{1}{q^{(t)}}\right) e^{\frac{n_s^{(t)}}{m}} - \frac{n_s^{(t)}}{m} - 1 \right),$$

$$\text{Var}(\hat{n}_s^{(t, \text{CSE})}) \approx m \left(\mathbb{E}\left(\frac{1}{q^{(t)}}\right) e^{\frac{n_s^{(t)}}{m}} - \frac{n_s^{(t)}}{m} - 1 \right),$$

where $q^{(t)} = \frac{U^{(t)}}{M}$ is the fraction of zero bits in the shared bit array at time t . We find that CSE exhibits a large bias when $n_s^{(t)} \gg m$, while FreeBS is unbiased. When m is large (e.g., m approaches to M), FreeBS and CSE perform nearly the same bit setting operations but use their different cardinality estimators. Then, we have $\mathbb{E}(\frac{1}{q^{(t)}}) \approx \mathbb{E}(\frac{1}{q_B^{(t)}})$. From Theorem 2, we have

$$\text{Var}(\hat{n}_s^{(t, \text{CSE})}) \gtrsim m \left(\mathbb{E}\left(\frac{1}{q^{(t)}}\right) \left(1 + \frac{n_s^{(t)}}{m}\right) - \frac{n_s^{(t)}}{m} - 1 \right)$$

$$> n_s^{(t)} \mathbb{E}\left(\frac{1}{q^{(t)}}\right) - n_s^{(t)} \gtrsim \text{Var}(\hat{n}_s^{(t, \text{FreeBS})});$$

3) FreeBS has complexity $O(1)$ to update user cardinality estimates each time it observes a new user-item pair, while CSE has complexity $O(m)$.

FreeRS vs vHLL. FreeRS outperforms vHLL in two aspects: 1) FreeRS has complexity $O(1)$ to update user cardinality estimates each time it observes a new user-item pair, while vHLL has complexity $O(m)$; 2) FreeRS exhibits a smaller estimation error in comparison with vHLL. From Theorem 2, we have $\text{Var}(\hat{n}_s^{(t, \text{FreeRS})}) \leq n_s^{(t)} (\mathbb{E}(\frac{1}{q_R^{(t)}}) - 1) \approx n_s^{(t)} (\frac{n^{(t)}}{M \alpha_M} - 1) < \frac{1.386n^{(t)}n_s^{(t)}}{M}$, while the variance of vHLL is

$$\text{Var}(\hat{n}_s^{(t, \text{vHLL})}) \gtrsim \left(\frac{M}{M-m}\right)^2 \times \frac{1.04^2}{m} \times 2n^{(t)}n_s^{(t)} \frac{m}{M} \left(1 - \frac{m}{M}\right)$$

$$= \frac{2.163n^{(t)}n_s^{(t)}}{M-m} > \frac{2.163n^{(t)}n_s^{(t)}}{M}.$$

FreeBS vs FreeRS. We observe that 1) FreeBS is faster than FreeRS. For each coming user-item pair e , FreeBS only computes $h^*(e)$ to select and set a bit, but FreeRS needs to compute both $h^*(e)$ and $\rho^*(e)$ to select and update a register; 2) Under the same memory usage, we compare the accuracy of FreeBS using M bits and FreeRS using M/w registers, where w is the number of bits in a register. From Theorems 1 and 2, we have

$$\text{Var}(\hat{n}_s^{(t, \text{FreeBS})}) = \sum_{i \in T_s^{(t)}} \mathbb{E}\left(\frac{1}{q_B^{(i)}}\right) - n_s^{(t)},$$

$$\text{Var}(\hat{n}_s^{(t, \text{FreeRS})}) = \sum_{i \in T_s^{(t)}} \mathbb{E}\left(\frac{1}{q_R^{(i)}}\right) - n_s^{(t)},$$

where $\mathbb{E}(\frac{1}{q_B^{(i)}}) \approx e^{\frac{n^{(i)}}{M}}$ and $\mathbb{E}(\frac{1}{q_R^{(i)}}) \approx \frac{1.386wn^{(i)}}{M} < e^{\frac{n^{(i)}}{M}}$ when $\frac{n^{(i)}}{M} \geq 0.772w$. Therefore, FreeRS is more accurate than FreeBS for users not appearing among the first $0.772wM$ distinct user-item pairs presented on stream Γ . Flajolet et al. [17] observe that HLL exhibits a large error for estimating small cardinalities. To

solve this problem, they view a register of HLL as a bit of LPC and estimate the cardinality based on the fraction of registers that equal 0. When $n^{(i)} \ll M/w$, we easily find that $q_R^{(i)}$ is approximately computed as the fraction of registers that equal 0 at time i , and we have $\mathbb{E}(\frac{1}{q_B^{(i)}}) < \mathbb{E}(\frac{1}{q_R^{(i)}})$ because the number of bits in FreeBS is w times larger than that of registers in FreeRS under the same memory usage. It indicates that FreeBS is more accurate than FreeRS for users whose user-item pairs appear early in stream Γ .

Limitations. The limitation of all above methods to estimate all users' cardinalities over time is that they require to store users together with their estimated cardinalities. We consider this limitation in the following cases: **case 1)** When exploiting all these methods to compute user cardinalities in graph streams consisting of a limited number of users of interest (e.g., monitoring the network connection counts of hosts in a campus), the number of users is not large enough. Therefore, storing all users and their cardinalities requires a limited amount of memory space and so it is practical. **case 2)** When estimating user cardinalities for graph streams consisting of a large number of users of interest (e.g., monitoring the network connection counts of hosts on edge routers between internal and external networks), it is impractical to record all users and cardinalities due to limited memory resources. However, many applications such as anomaly detection only focus on users with extremely large cardinalities (i.e., super spreaders). Therefore, we can implement sketch methods such as Space Saving [35] and Unbiased Space Saving [43] to store only a small number of users and their cardinalities.

6 EXTENSION: SUPER SPREADER DETECTION

For estimating cardinalities in graph streams over time, above methods all require to set an extra counter for each user to keep track of its corresponding cardinality. However, many real-world datasets contain millions of or more users, which will cost extremely large memory resources. Meanwhile, real-world datasets are always with skewed distribution (i.e., there are few users with large cardinalities), and in many cases such as anomaly detection, we mainly focus on monitoring abnormal users with large cardinalities rather than all users. As mentioned above, existing frequency estimation methods cannot be directly used for complex super spreader detection due to edge duplicates in graph streams, while existing super spreader detection methods exhibit large estimation errors. Therefore, in this section, we extend both our methods and combine them with the *Unbiased Space Saving* method [43] to monitor top- K super spreaders in real-time. Unbiased Space Saving almost implements the data structure and update procedure of *Space Saving* method [35], but replaces the minimum element with a varied probability to make it unbiased. The basic idea of our methods is that we use FreeBS/FreeRS to handle edge duplicates in graph streams and accurately estimate user cardinalities, and then utilize Unbiased Space Saving method to monitor top- K super spreaders with small time complexity.

6.1 FreeBS-SSD: FreeBS for Super Spreader Detection

Data Structure. The pseudo-code for FreeBS-SSD is shown as Algorithm 3. FreeBS-SSD consists of two data structures, a one-dimension bit array B of length M used to record connected items and estimate cardinalities for all users over time, and a *Stream-Summary* data structure H with l buckets used to store users with large cardinalities. Specifically, the Stream-Summary data

Algorithm 3: The pseudo-code for FreeBS-SSD.

```

 $B[1, \dots, M] \leftarrow [0, \dots, 0]; \quad H[1, \dots, l] \leftarrow [\emptyset, \dots, \emptyset];$ 
 $\hat{n}_s^{(\text{FreeBS-SSD})} \leftarrow 0, s \in S; \quad m_0 \leftarrow M;$ 
foreach  $e = (s, d)$  in  $\Gamma$  do
    if  $B[h^*(e)] == 0$  then
         $B[h^*(e)] \leftarrow 1;$ 
         $q_B \leftarrow \frac{m_0}{M};$ 
        if  $s \in H$  then
             $\hat{n}_s^{(\text{FreeBS-SSD})} \leftarrow \hat{n}_s^{(\text{FreeBS-SSD})} + \frac{1}{q_B};$ 
            rerank buckets in  $H;$ 
        else
            if  $H$  is full then
                 $p_t \leftarrow \frac{1/q_B}{\hat{n}_{\min}^{(\text{FreeBS-SSD})} + 1/q_B};$ 
                 $r \leftarrow \text{Rand}(0, 1);$ 
                if  $r \leq p_t$  then
                    replace  $s_{\min}$  with  $s;$ 
                     $\varepsilon_s \leftarrow \hat{n}_{\min}^{(\text{FreeBS-SSD})};$ 
                     $\hat{n}_s^{(\text{FreeBS-SSD})} \leftarrow \hat{n}_{\min}^{(\text{FreeBS-SSD})} + \frac{1}{q_B};$ 
                    rerank buckets in  $H;$ 
                else
                     $\hat{n}_{\min}^{(\text{FreeBS-SSD})} \leftarrow \hat{n}_{\min}^{(\text{FreeBS-SSD})} + \frac{1}{q_B};$ 
                    rerank buckets in  $H;$ 
                end
            else
                 $\hat{n}_s^{(\text{FreeBS-SSD})} \leftarrow \hat{n}_s^{(\text{FreeBS-SSD})} + \frac{1}{q_B};$ 
                 $\varepsilon_s \leftarrow 0;$ 
                insert  $s$  into  $H$  and rerank buckets in  $H;$ 
            end
        end
    end
     $m_0 \leftarrow m_0 - 1;$ 
end

```

/ $\hat{n}_{\min}^{(\text{FreeBS-SSD})}$ is the minimum counter, $s_{\min}$ is the user in this bucket. */*

structure is a doubly linked list, where each bucket is initialized as $\emptyset$ and contains three parameters, user s , counter $\hat{n}_s^{(\text{FreeBS-SSD})}$ used to record its estimated cardinality, and its maximum over-estimation ε_s . Note there should be a sufficient number of buckets to record super spreaders, i.e., the number of buckets is larger than that of super spreaders. Meanwhile, all buckets in H are ranked according to values of their counters.

Update Procedure. When a user-item pair $e = (s, d)$ arrives at time t , FreeBS-SSD first updates the bit array B as described in Algorithm 1, and then updates Stream-Summary H . In detail, when user s is already monitored in H , i.e., the bucket of user s is contained in H , we directly update its corresponding counter as $\hat{n}_s^{(t, \text{FreeBS-SSD})} \leftarrow \hat{n}_s^{(t-1, \text{FreeBS-SSD})} + \frac{1}{q_B^{(t)}}$, where $q_B^{(t)} = \frac{m_0^{(t-1)}}{M}$ is the probability of a user-item pair e changing a bit in B from 0 to 1 at time t . Then we rerank buckets in H according to values of their counters. Because all buckets in H are always-sorted, we only need to move the updated bucket forward into a position where its counter is smaller than that of the prior bucket and larger than the next bucket. In other words, we usually need to move the updated bucket forward a bucket to keep H ranked. On the contrary, when the arriving user s is not monitored in H , we consider two different scenarios: scenario 1) H is full and there

are no empty buckets left, we choose to update the bucket with the minimum counter. Let $\hat{n}_{\min}^{(t, \text{FreeBS-SSD})}$ denote the minimum counter in H at time t , and its corresponding user is represented as $s_{\min}$. We use a probability $p_t = \frac{1/q_B^{(t)}}{\hat{n}_{\min}^{(t, \text{FreeBS-SSD})} + 1/q_B^{(t)}}$ and compare it with a random number r in $(0, 1)$. If this condition is met, we update the bucket as $s_{\min} \leftarrow s, \varepsilon_s \leftarrow \hat{n}_{\min}^{(t, \text{FreeBS-SSD})}, \hat{n}_s^{(t, \text{FreeBS-SSD})} \leftarrow \hat{n}_{\min}^{(t, \text{FreeBS-SSD})} + \frac{1}{q_B^{(t)}}$, and then rerank buckets in H according to their counters. Otherwise, we update the minimum counter as $\hat{n}_{\min}^{(t, \text{FreeBS-SSD})} \leftarrow \hat{n}_{\min}^{(t, \text{FreeBS-SSD})} + \frac{1}{q_B^{(t)}}$, and rerank buckets in H ; scenario 2) There are still empty buckets in H , we initialize the bucket of user s and insert it into a suitable position according to its counter. Specially, the over-estimation of user s is initialized as $\varepsilon_s \leftarrow 0$. Therefore, the time complexity of FreeBS-SSD to update the buckets in H for each arriving user-item pair is $O(1)$.

Error Analysis According to the data structure and update procedure of FreeBS-SSD, we have the following theorem.

Theorem 3. For any user s , FreeBS-SSD gives an unbiased estimation of its cardinality $n_s^{(t, \text{FreeBS-SSD})}$ at any time t .

Proof. Here we simplify $\hat{n}_s^{(t, \text{FreeBS-SSD})}$ as $\hat{n}_s^{(t)}$ to ease exposition. There are two scenarios that $\hat{n}_s^{(t)}$ will be changed by the arriving user: 1) user s is the next arriving user at time $t+1$, the estimated cardinality is updated as $\hat{n}_s^{(t+1)} \leftarrow \hat{n}_s^{(t)} + \frac{1}{q_B^{(t+1)}}$. As mentioned in Theorems 1, this increment is an unbiased estimation; 2) user s is not the next arriving user at time $t+1$ but its counter $\hat{n}_s^{(t)}$ is the minimum one in H . In this case, $\hat{n}_s^{(t)}$ is changed with probability $p_t = \frac{1/q_B^{(t)}}{\hat{n}_s^{(t)} + 1/q_B^{(t)}}$, i.e., $\hat{n}_s^{(t+1)} = \hat{n}_s^{(t)}$ happens with probability $1 - p_t$, and the expected increment to be updated is computed as

$$\mathbb{E}(\hat{n}_s^{(t+1)} - \hat{n}_s^{(t)}) = (\hat{n}_s^{(t)} + \frac{1}{q_B^{(t)}})(1 - \frac{1/q_B^{(t)}}{\hat{n}_s^{(t)} + 1/q_B^{(t)}}) - \hat{n}_s^{(t)} = 0.$$

Therefore, FreeBS-SSD shows an unbiased estimation of its cardinality $n_s^{(t, \text{FreeBS-SSD})}$ at any time t . $\square$

6.2 FreeRS-SSD: FreeRS for Super Spreader Detection

Data Structure. The pseudo-code for FreeRS-SSD is shown as Algorithm 4. FreeRS-SSD consists of a register array of M registers $R[1], \dots, R[M]$ to record connected items and estimate cardinalities for all users, and a Stream-Summary data structure H of l buckets to store users with large cardinalities.

Update Procedure. When a user-item pair $e = (s, d)$ arrives at time t , FreeRS-SSD follows the same procedure of FreeRS to update the register array, and implements a similar procedure as FreeBS-SSD to update buckets in H . Specifically, when user s is contained in H , we directly update its counter as $\hat{n}_s^{(t, \text{FreeRS-SSD})} \leftarrow \hat{n}_s^{(t-1, \text{FreeRS-SSD})} + \frac{1}{q_R^{(t)}}$, where $q_R^{(t)} \leftarrow q_R^{(t-1)} + \frac{2^{-\rho^*(e)} - 2^{-R[h^*(e)]}}{M}$ denotes the probability of changing the value of a register among $R[1], \dots, R[M]$. Then we move the updated bucket into a position of H according to its counter to keep H ranked. When user s is not monitored in H , we also consider the following scenarios: scenario 1) H is full, then we update the bucket with the minimum counter as $s_{\min} \leftarrow s, \varepsilon_s \leftarrow \hat{n}_{\min}^{(t, \text{FreeRS-SSD})}, \hat{n}_s^{(t, \text{FreeRS-SSD})} \leftarrow \hat{n}_{\min}^{(t, \text{FreeRS-SSD})} + \frac{1}{q_R^{(t)}}$ with the probability $p_t = \frac{1/q_R^{(t)}}{\hat{n}_{\min}^{(t, \text{FreeRS-SSD})} + 1/q_R^{(t)}}$, and rerank buckets in H according to their counters. Otherwise, we only update the bucket

Algorithm 4: The pseudo-code for FreeRS-SSD.

```

 $R[1, \dots, M] \leftarrow [0, \dots, 0]; \quad H[1, \dots, l] \leftarrow [\emptyset, \dots, \emptyset];$ 
 $\hat{n}_s^{(\text{FreeRS-SSD})} \leftarrow 0, s \in S; \quad q_R \leftarrow 1;$ 
foreach  $e = (s, d) \in \Gamma$  do
  if  $\rho^*(e) > R[h^*(e)]$  then
    if  $s \in H$  then
       $\hat{n}_s^{(\text{FreeRS-SSD})} \leftarrow \hat{n}_s^{(\text{FreeRS-SSD})} + \frac{1}{q_R};$ 
      rerank buckets in  $H$ ;
    else
      if  $H$  is full then
         $p_t \leftarrow \frac{1/q_R}{\hat{n}_{\min}^{(\text{FreeRS-SSD})} + 1/q_R};$ 
         $r \leftarrow \text{Rand}(0, 1);$ 
        if  $r \leq p_t$  then
          replace  $s_m$  with  $s$ ;
           $\varepsilon_s \leftarrow \hat{n}_{\min}^{(\text{FreeRS-SSD})};$ 
           $\hat{n}_s^{(\text{FreeRS-SSD})} \leftarrow \hat{n}_{\min}^{(\text{FreeRS-SSD})} + \frac{1}{q_R};$ 
          rerank buckets in  $H$ ;
        else
           $\hat{n}_{\min}^{(\text{FreeRS-SSD})} \leftarrow \hat{n}_{\min}^{(\text{FreeRS-SSD})} + \frac{1}{q_R};$ 
          rerank buckets in  $H$ ;
        end
      else
         $\hat{n}_s^{(\text{FreeRS-SSD})} \leftarrow \hat{n}_s^{(\text{FreeRS-SSD})} + \frac{1}{q_R};$ 
         $\varepsilon_s \leftarrow 0;$ 
        insert  $s$  into  $H$  and rerank buckets in  $H$ ;
      end
    end
     $q_R \leftarrow q_R + \frac{2^{-\rho^*(e)} - 2^{-R[h^*(e)]}}{M};$ 
     $R[h^*(e)] \leftarrow \rho^*(e);$ 
  end
end

```

as $\hat{n}_{\min}^{(t, \text{FreeRS-SSD})} \leftarrow \hat{n}_{\min}^{(t, \text{FreeRS-SSD})} + \frac{1}{q_R^{(t)}}$, and rerank buckets in H ; scenario 2) There are still empty buckets left in H , we initialize the counter of user s , and its over-estimation is also set as $\varepsilon_s \leftarrow 0$. Then we move the bucket into a suitable position in H according to the value of its counter. Also, the time complexity of updating the buckets in H for each arriving user-item pair is $O(1)$.

Error Analysis Similar to FreeBS-SSD, we have

Theorem 4. For any user s , FreeRS-SSD gives an unbiased estimation of its cardinality $n_s^{(t, \text{FreeRS-SSD})}$ at any time t .

The proof of the above theorem is similar to that of Theorem 3, and we omit it due to limited pages.

6.3 Super Spreader Detection

Based on FreeBS-SSD and FreeRS-SSD, we detect top- K super spreaders, i.e., the set of users whose cardinalities rank top- K among all users. As mentioned above, each bucket in H contains three parameters user s , counter $\hat{n}_s^{(t, \text{FreeBS-SSD})}$ for FreeBS-SSD (resp. $\hat{n}_s^{(t, \text{FreeRS-SSD})}$ for FreeRS-SSD) and maximum over-estimation ε_s , and all buckets are ranked according to the values of their counters. $|\hat{n}_s^{(t, \text{FreeBS-SSD})} - \varepsilon_s|$ (resp. $|\hat{n}_s^{(t, \text{FreeRS-SSD})} - \varepsilon_s|$) represents the increments after user s is inserted in the Stream-Summary data structure H , which can also be approximately treated as the cardinality of user s . Therefore, we sequentially enumerate all buckets in H , and then report users with top- K largest values of $|\hat{n}_s^{(t, \text{FreeBS-SSD})} - \varepsilon_s|$ (resp. $|\hat{n}_s^{(t, \text{FreeRS-SSD})} - \varepsilon_s|$).

7 EVALUATION

7.1 Datasets

We conduct our experiments on a variety of publicly available real-world datasets, which are summarized in Table 1. Dataset sanjose (resp. chicago) consists of one-hour passive traffic traces collected from the equinix-sanjose (resp. equinix-chicago) data collection monitors during March 20, 2014. Twitter, Flickr, Orkut and LiveJournal are all graph-datasets where each edge represents social relationship between any two users and may occur more than once in these datasets. Figure 4 shows the CCDFs of user cardinalities for all datasets used in our experiments.

Table 1: Summary of datasets used in our experiments.

dataset	#users	max-cardinality	total cardinality
sanjose [1]	8,387,347	313,772	23,073,907
chicago [1]	1,966,677	106,026	9,910,287
Twitter [29]	40,103,281	2,997,496	1,468,365,182
Flickr [36]	1,441,431	26,185	22,613,980
Orkut [36]	2,997,376	31,949	223,534,301
LiveJournal [36]	4,590,650	9,186	76,937,805

Figure 4: CCDFs of user cardinalities.

7.2 Baselines

In this paper, we compare our methods with other state-of-the-arts under the same memory usage. Therefore, we let bit-sharing algorithms FreeBS and CSE [53] have M bits, whereas register-sharing algorithms FreeRS and vHLL [49] have $M/5$ 5-bit registers. Moreover, CSE and vHLL use virtual sketches of size m to record the cardinality for each user, and we vary different m and select the optimal one according to a given metric. LPC [48] and HyperLogLog++ [24] (short for HLL++) build a sketch for each user $s \in S$ to record items that s connects to. Specially, HLL++ is an optimized method of HyperLogLog, which uses 6 bits for each register, and implements bias correction and sparse representation strategies to improve cardinality estimation performance. In our experiments, we let LPC have $\frac{M}{|S|}$ bits and HLL++ have $\frac{M}{6|S|}$ 6-bit registers for each user respectively. In addition, we also compare our methods with a combined method BF+CM (i.e., a Bloom Filter [5] and a Count-Min sketch [9]). In detail, we set each counter in Count-Min as 32 bit, the number of hash functions in the Bloom Filter as 3, and use a parameter μ ($0 < \mu < 1$) to control the memory space of both structures, where μ is the ratio of the memory size of the Bloom Filter to the total memory size. Let w denote the number of counter arrays contained in Count-Min, and then the number of counters in each counter array is $\frac{M(1-\mu)}{32w}$. Later in our experiments, we vary parameters μ and w and select optimal ones under the same memory space.

In this paper, we aim to compute the cardinalities of all users. At each time t , enumerating each occurred user and computing

its cardinality requires time $O(|S^{(t)}|m)$ for methods CSE, vHLL, LPC, and HLL++, where $S^{(t)}$ is the set of occurred users before and include time t . It is computationally intensive and thus is prohibitive for estimating all users' cardinalities over time t . To solve this problem, we allocate each occurred user s a counter $\hat{n}_s$ to keep tracking of its cardinality for CSE, vHLL, LPC, and HLL++. For each edge $e^{(t)} = (s^{(t)}, d^{(t)})$ arriving at time t , we only estimate the cardinality of user $s^{(t)}$ for CSE, vHLL, LPC, and HLL++, and then update its counter $\hat{n}_{s^{(t)}}$ and keep the counters of the other occurred users unchanged, which reduces time complexity from $O(|S^{(t)}|m)$ to $O(m)$. To track all users' cardinalities over time, therefore, all methods FreeRS, FreeBS, CSE, vHLL, LPC, and HLL++ require a counter for each user, and this memory usage of all methods is the same and we do not consider this memory usage in our comparisons.

7.3 Metrics

In our experiments, we use a fine-grained metric *relative standard error* (RSE) to evaluate the performance of estimating the cardinality for any user with a particular cardinality n at time t .

Formally, we define $RSE^{(t)}(n) = \frac{1}{n} \sqrt{\frac{\sum_{s \in S} (\hat{n}_s^{(t)} - n)^2 \mathbf{1}(n_s^{(t)} = n)}{\sum_{s \in S} \mathbf{1}(n_s^{(t)} = n)}}$.

In addition, we also use a metric *average absolute relative error* (AARE) to evaluate the average performance of estimating each user's cardinality with respect to its true value at any time t , and AARE is formally defined as $AARE^{(t)} = \frac{1}{|S|} \sum_{s \in S} \frac{|\hat{n}_s^{(t)} - n_s^{(t)}|}{n_s^{(t)}}$.

Figure 5: (Orkut) Estimated cardinalities vs actual cardinalities for LPC and CSE under the same memory size $M = 5 \times 10^8$ (bits), where the number of bits in the virtual sketch of CSE $m = 1024$.

7.4 Accuracy of User Cardinality Estimation

Whenever an element $e = (s, d)$ arrives, we implement all methods to estimate the cardinality of user s respectively. We first present the experimental results of methods LPC and CSE in the dataset Orkut in Figure 5, where the memory size $M = 5 \times 10^8$ bits and the number of bits in the virtual sketch of CSE $m = 1024$. The experimental results in other datasets are similar. Points close to the diagonal line $\hat{n}_s^{(t)} = n_s^{(t)}$ indicate better estimation accuracy. From Figure 5, we observe that LPC and CSE have limited estimation ranges and LPC exhibits extremely bad estimation performance especially for larger cardinalities. Under the fixed memory size, each user in the dataset Orkut uses only 167 bits for LPC and its maximum estimation range is much smaller than other methods. Therefore we omit the experimental results of LPC in the later experiments. Meanwhile, we vary the parameters of different methods respectively and use the metric AARE to compare their estimation performance in the dataset sanjose, which is shown

Figure 6: (sanjose) AAREs of our methods in comparison with CSE, vHLL, HLL++, and BF+CM for different parameters.

in Figure 6. AAREs of all methods decrease as the total memory usage increases from 1×10^8 to 5×10^8 bits (Figure 6 (a)), and both our methods FreeBS and FreeRS have better performance than other methods. When we use $M = 1 \times 10^8$ bits to estimate user cardinalities for dataset sanjose that contains 8,387,347 users, each user of methods CSE/FreeBS has about 12 bits and each user of vHLL/HLL++/FreeRS has only about 2 registers on average. In this case, there occur many “noisy” bits/registers contaminated by other users in the sharing methods CSE/vHLL/FreeBS/FreeRS, and the number of each user’s registers of HLL++ is so limited that it cannot guarantee the estimation accuracy. In addition, we notice that AAREs of CSE and vHLL first decrease and then increase when m increases, and both methods maintain the best performance when $m = 1024$ (Figure 6 (b)). This is consistent with our analysis in Section 4.2.3. Increasing m can increase the estimation performance of CSE/vHLL to some extent but it introduces more “unused” bits/registers for users with small cardinalities, which makes it easy for these bits/registers to be contaminated by other users. Furthermore, we observe that BF+CM is more accurate when $\mu = 0.2$ and $w = 2$ (Figures 6 (c) and (d)) but its AARE is larger than other methods except CSE because it costs large memory space to handle edge duplicates in graph streams and introduces complex errors to Count-Min sketch.

Based on above experiments, we fix the total memory usage as $M = 5 \times 10^8$ bits, and set $m = 1024$ for CSE/vHLL and $\mu = 0.2, w = 2$ for BF+CM. We use the metric RSE to compare the estimation errors of our methods with other methods at a fine-grained level, which is shown in Figure 7. We see that our methods FreeBS and FreeRS are up to 10,000 times more accurate than CSE, vHLL, HLL++ and BF+CM. Bit sharing method FreeBS (resp. CSE) is more accurate than register sharing method FreeRS (resp. vHLL) for characterizing users with small cardinalities. For users with large cardinalities, register sharing method FreeRS (resp. vHLL) outperforms bit sharing method FreeBS (resp. CSE). Specially, the RSE of CSE first decreases and then increases as the actual cardinality increases. This is because CSE has a small estimation range, i.e., $m \ln m$, which is more consistent with the original paper [53]. Meanwhile, HLL++ is more accurate than CSE and vHLL for small cardinalities due to its bias correction strategy. Because each user uses fewer registers for HLL++ than vHLL, HLL++ performs larger estimation errors

Figure 7: (All datasets) Cardinality estimation accuracy (RSE), where memory size $M = 5 \times 10^8$ (bits), and $m = 1024$ for CSE/vHLL, and $\mu = 0.2, w = 2$ for BF+CM.

Figure 8: (sanjose) Cardinality estimation accuracy (AARE) and memory usage (bits/user) of cardinality estimation over time, where memory size $M = 5 \times 10^8$ (bits), $m = 1024$ for CSE/vHLL, and $\mu = 0.2, w = 2$ for BF+CM.

than vHLL for large cardinalities. Also, when using Count-Min to record user cardinalities, users with small cardinalities are more easily affected by other users and thus identified as users with large cardinalities. Therefore, RSE of BF+CM is large for small cardinalities. Besides, Figure 8 (a) shows AAREs of all methods for estimating user cardinalities over time and Figure 8 (b) presents the memory usage per user. We notice that the estimation errors of all methods increase when more and more elements arrive in the graph stream while both our methods FreeBS and FreeRS have better performance than other methods. Also, the memory usage per user decreases as the volume of graph streams increases and each user costs about 60 bits at the end of the graph stream.

7.5 Runtime

For CSE, vHLL, LPC and HLL++, its runtime is mainly determined by the number of bits/registers in each user’s (virtual) sketch, because estimating a user’s cardinality needs to enumerate m bits/registers in the user’s sketch, which requires large time

Figure 9: Runtime of our methods FreeBS and FreeRS in comparison with CSE, vHLL, LPC and HLL++ for different m (i.e., the number of bits/registers in the (virtual) sketch) of a user.

Figure 10: (sanjose) Accuracy of detecting top-100 super spreaders for different λ with the same memory size, where $M = 5 \times 10^8$ (bits), $m = 1024$ for CSE/vHLL, and $\mu = 0.2, w = 2$ for BF+CM.

Figure 11: (All datasets) Accuracy of detecting top-100 super spreaders, where $M = 5 \times 10^8$ (bits), $\lambda = \frac{1}{6}$ for our methods, $m = 1024$ for CSE/vHLL, and $\mu = 0.2, w = 2$ for BF+CM.

complexity $O(m)$. On the contrary, our methods FreeBS and FreeRS have low time complexity $O(1)$ to process each coming element and update the counter for its user. In our experiments, we vary the number of bits/registers in the sketch m and record the average time required for processing each element and updating the user cardinality of all six methods. The experimental results are shown in Figure 9. As m increases, the runtime of all four methods CSE, vHLL, LPC and HLL++ increases. Our methods FreeBS and FreeRS are substantially faster than other four methods for different m . Also, we notice that CSE is faster than vHLL and FreeBS is faster than FreeRS, this is because register sharing based methods perform more operations than bit sharing methods for processing each element.

7.6 Accuracy of Super Spreader Detection

For detecting super spreaders, we compare our methods with CSE, vHLL, HLL++, BF+CM, and a reversible sketch-based method *Vector Bloom Filter* (VBF) [31] under the same memory size. VBF

Figure 12: (sanjose) Accuracy of detecting top-100 super spreaders over time, where $M = 5 \times 10^8$ (bits), $\lambda = \frac{1}{6}$ for our methods, $m = 1024$ for CSE/vHLL, and $\mu = 0.2, w = 2$ for BF+CM.

consists of 5 2-dimensional bit arrays and 6 hash functions, where each bit array contains $4096 \times m$ bits initialized to 0. Whenever an element $e = (s, d)$ arrives, VBF uses a 32-bit string to represent user s and divides it into 8 sub-strings without overlapping, i.e., $s = s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8$. In this case, the first four hash values are computed as $h_1(s) = s_1 s_2 s_3, h_2(s) = s_3 s_4 s_5, h_3(s) = s_5 s_6 s_7, h_4(s) = s_7 s_8 s_1$, and the fifth hash function randomly maps user s into an integer in $\{0, \dots, 4095\}$. All these hash functions are used to find the row index in each bit array. The sixth hash function is used to generate the column index in all bit arrays for each arriving element. Then the corresponding bit in each bit array is set to 1. After, VBF enumerates all hash values in the first four bit arrays to reconstruct users based on their overlapping parts, and estimates their corresponding cardinalities with all five bit arrays. Therefore, we can easily find top- K super spreaders.

In our experiments, we use two metrics *false negative ratio* (FNR) and *false positive ratio* (FPR) to evaluate super spreader detection performance. The memory size of CSE, vHLL, HLL++, and BF+CM consists of the sketch size and user-counter pairs, while the memory size of our methods includes the bit/register array and the Stream-Summary data structure. Each bucket in the Stream-Summary data structure contains three parameters, user, counter and maximum over-estimation, and we assign 32 bits to represent each of them respectively. Therefore, the total memory size is $M + 64|S|$ bits and the number of columns in each bit array of VBF is $\frac{M+64|S|}{4096 \times 5}$, where $|S|$ is the number of users. Also, we use a variable λ to control the ratio of the memory size of the Stream-Summary data structure to all memory size. Figure 10 shows the accuracy of detecting top-100 super spreaders in dataset sanjose, and we omit the similar results in other datasets. We notice that FNRs and FPRs of our methods are much smaller than other methods and increase as λ increases from $\frac{1}{6}$ to $\frac{1}{2}$. This is because that the accuracy of user cardinality estimation of our methods decreases when there is smaller memory size given for the bit/register array. Later in our experiments, we fix $\lambda = \frac{1}{6}$. Figure 11 shows the accuracy of detecting top-100 super spreaders for all datasets when all elements arrive. We see that FreeBS-SSD and FreeRS-SSD outperform other methods in all datasets. Specially, due to limited estimation range, CSE exhibits large estimation errors in datasets Twitter and Orkut. Although VBF reduces the memory usage of user-counter pairs using reversible sketches, its estimation accuracy is seriously affected by false users reversed but not actually occurred in graph streams. Also, it is computationally expensive to reverse each possible user, which is not practicable to detect super spreaders in real-time. Figure 12 shows FNRs and FPRs of all methods for detecting top-100 super spreaders over time. We observe that the number of super spreaders not detected increases over time and thus FNRs of all methods increase. The number of users that are misidentified

as super spreaders and the number of all users in graph streams increases simultaneously, and thus it is complex to analyze the trend of FPRs of all methods. Meanwhile, we notice that both our methods outperform other methods for detecting super spreaders.

8 CONCLUSIONS

In this paper, we develop two novel streaming algorithms FreeBS and FreeRS to accurately estimate user cardinalities over time. Compared to existing bit/register sharing methods using (i.e., selecting) only m bits/registers for each user, FreeBS/FreeRS enables that the number of bits/registers used by a user dynamically increases as its cardinality increases over time and each user can use all shared bits/registers. It is therefore capable to estimate user cardinalities over a large range. Besides, both our algorithms exploit dynamic properties of shared bits/registers to significantly improve estimation accuracy. They are simple yet effective, and sharply reduce time complexity of computing all user cardinalities to $O(1)$ each time they observe a new user-item pair. FreeBS is more accurate than FreeRS for users whose user-item pairs appear early in the graph stream, and FreeRS outperforms FreeBS for users whose user-item pairs arrive late. Furthermore, we extend both methods to accurately and fast detect top- K super spreaders. We conduct experiments on real-world datasets and experimental results demonstrate that our methods significantly outperform other methods in terms of accuracy and computational time.

ACKNOWLEDGMENT

The research presented in this paper is supported in part by National Key R&D Program of China (No.2016YFE0206700), Shenzhen Basic Research Grant (JCYJ20170816100819428), National Natural Science Foundation of China (61922067, U1736205, 61902305, 61828302), MoE-CMCC “Artificial Intelligence” Project (MCM20190701), Natural Science Basic Research Plan in Shaanxi Province of China (2019JM-159), Natural Science Basic Research Plan in ZheJiang Province of China (LGG18F020016).

REFERENCES

- [1] Internet traces 2014, center for applied internet data analysis (caida). http://www.caida.org/data/passive/passive_2014_dataset.xml.
- [2] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, 1964.
- [3] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *RANDOM*, pages 1–10, 2002.
- [4] R. B. Basat, G. Einziger, R. Friedman, and Y. Kassner. Randomized admission policy for efficient top-k and frequency estimation. In *INFOCOM*, pages 1–9, 2017.
- [5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.
- [6] J. Cao, Y. Jin, A. Chen, T. Bu, and Z.-L. Zhang. Identifying high cardinality internet hosts. In *INFOCOM*, pages 810–818, 2009.
- [7] A. Chen, J. Cao, L. Shepp, and T. Nguyen. Distinct counting with a self-learning bitmap. *JASS*, 106(495):879–890, 2011.
- [8] W. Chen, Y. Liu, and Y. Guan. Cardinality change-based early detection of large-scale cyber-attacks. In *INFOCOM*, pages 1788–1796, 2013.
- [9] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *JAL*, 55(1):58–75, 2005.
- [10] G. Cormode, S. Muthukrishnan, and D. Srivastava. Finding hierarchical heavy hitters in data streams. In *VLDB*, pages 464–475, 2003.
- [11] G. Cormode, S. Muthukrishnan, and D. Srivastava. Diamond in the rough: Finding hierarchical heavy hitters in multi-dimensional data. In *SIGMOD*, pages 155–166, 2004.
- [12] M. Durand and P. Flajolet. *Loglog Counting of Large Cardinalities*, pages 605–617. Springer Berlin Heidelberg, 2003.
- [13] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *SIGCOMM*, pages 323–336, 2002.
- [14] C. Estan and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *SIGCOMM*, pages 137–148, 2003.
- [15] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. In *SIGCOMM*, pages 182–209, 2003.
- [16] P. Flajolet. On adaptive sampling. *Computing*, 43(4):391–400, 1990.
- [17] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA*, pages 127–146, 2007.
- [18] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *JCSS*, 31(2):182–209, 1985.
- [19] P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, pages 541–550, 2001.
- [20] F. Giroire. Order statistics and estimating cardinalities of massive data sets. *Discrete Applied Mathematics*, 157(2):406 – 427, 2009.
- [21] J. Gong, T. Yang, H. Zhang, H. Li, S. Uhlig, S. Chen, L. Uden, and X. Li. Heavykeeper: an accurate algorithm for finding top-k elephant flows. In *ATC*, pages 909–921, 2018.
- [22] X. Guan, P. Wang, and T. Qin. A new data streaming method for locating hosts with large connection degree. In *GLOBECOM*, pages 1–6, 2009.
- [23] H. Harmouch and F. Naumann. Cardinality estimation: an experimental survey. *VLDB*, 11(4):499–512, 2017.
- [24] S. Heule, M. Nunkesser, and A. Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *EDBT*, pages 683–692, 2013.
- [25] D. Horvitz and D. Thompson. A generalization of sampling without replacement from a finite universe. *JASS*, 47(260):663–685, 1952.
- [26] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang. Sketchvisor: Robust network measurement for software packet processing. In *SIGCOMM*, pages 113–126, 2017.
- [27] N. Kamiyama, T. Mori, and R. Kawahara. Simple and adaptive identification of superspreaders by flow sampling. In *INFOCOM*, pages 2481–2485, 2007.
- [28] D. M. Kane, J. Nelson, and D. P. Woodruff. An optimal algorithm for the distinct elements problem. In *PODS*, pages 41–52, 2010.
- [29] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.
- [30] Y. Li, R. Miao, C. Kim, and M. Yu. Flowradar: A better netflow for data centers. In *NSDI*, pages 311–324, 2016.
- [31] W. Liu, W. Qu, J. Gong, and K. Li. Detection of superpoints using a vector bloom filter. *TIFS*, 11(3):514–527, 2016.
- [32] Y. Liu, W. Chen, and Y. Guan. Identifying high-cardinality hosts from network-wide traffic measurements. *TDSC*, 13(5):547–558, 2012.
- [33] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *SIGCOMM*, pages 101–114, 2016.
- [34] J. Lumbroso. An optimal cardinality estimation algorithm based on order statistics and its full analysis. In *AOFA*, pages 489–504, 2010.
- [35] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, pages 398–412, 2005.
- [36] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *IMC*, pages 29–42, 2007.
- [37] W. Pan, Z. Li, Y. Zhang, and C. Weng. The new hardware development trend and the challenges in data management and analysis. *TDSE*, 3(3):263–276, 2018.
- [38] R. Schwellen, Z. Li, Y. Chen, Y. Gao, A. Gupta, Y. Zhang, P. A. Dinda, M.-Y. Kao, and G. Memik. Reversible sketches: enabling monitoring and analysis over high-speed data streams. *TON*, 15(5):1059–1072, 2007.
- [39] X. Shi, D.-M. Chiu, and J. C. Lui. An online framework for catching top spreaders and scanners. *Computer Networks*, 54(9):1375–1388, 2010.
- [40] Y. Tang and S. Chen. Online identification of multi-attribute high-volume traffic aggregates through sampling. In *ICC*, pages 977–981, 2005.
- [41] J. Tao, P. Wang, X. Guan, and W. Hu. Al-bitmap: Monitoring network traffic activity graphs on high speed links. *Inf. Sci.*, 408:162–175, 2017.
- [42] D. Ting. Streamed approximate counting of distinct elements: Beating optimal batch methods. In *KDD*, pages 442–451, 2014.
- [43] D. Ting. Data sketches for disaggregated subset sum and frequent item estimation. In *SIGMOD*, pages 1129–1140, 2018.
- [44] G. Vargas-Solar, J.-L. Zechinelli-Martini, and J.-A. Espinosa-Oviedo. Big data management: What to keep from the past to face future challenges? *TDSE*, 2(4):328–345, 2017.
- [45] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum. New streaming algorithms for fast detection of superspreaders. In *NDSS*, pages 1–28, 2004.

- [46] P. Wang, X. Guan, T. Qin, and Q. Huang. A data streaming method for monitoring host connection degrees of high-speed links. *TIFS*, 6(3-2):1086–1098, 2011.
- [47] P. Wang, X. Guan, D. Towsley, and J. Tao. Virtual indexing based methods for estimating node connection degrees. *Computer Networks*, 56(12):2773–2787, 2012.
- [48] K. Whang, B. T. Vander-zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *TDS*, 15(2):208–229, 1990.
- [49] Q. Xiao, S. Chen, M. Chen, and Y. Ling. Hyper-compact virtual estimators for big network data based on register sharing. In *SIGMETRICS*, pages 417–428, 2015.
- [50] Q. Xiao, Y. Zhou, and S. Chen. Better with fewer bits: Improving the performance of cardinality estimation of large data streams. In *INFOCOM*, pages 1–9, 2017.
- [51] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi, and X. Li. Heavyguardian: Separate and guard hot items in data streams. In *KDD*, pages 2584–2593, 2018.
- [52] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *SIGCOMM*, pages 561–575, 2018.
- [53] M. Yoon, T. Li, S. Chen, and J. K. Peir. Fit a spread estimator in small memory. In *INFOCOM*, pages 504–512, 2009.
- [54] Q. Zhao, A. Kumar, and J. Xu. Joint data streaming and sampling techniques for detection of super sources and destinations. In *SIGCOMM*, pages 77–90, 2005.

Peng Jia received the B.S. in automation engineering from Xi'an Jiaotong University, Xi'an, China, in 2016. He is currently a graduate student with MOE Key Laboratory for Intelligent Networks and Network Security at Xi'an Jiaotong University. His research interests include Internet traffic measurement and modeling, abnormal detection, and online social network measurement.

Pinghui Wang received the B.S. degree in information engineering and the Ph.D. degree in automatic control from Xi'an Jiaotong University, Xi'an, China, in 2006 and 2012 respectively. He was a postdoctoral researcher with the Department of Computer Science and Engineering at The Chinese University of Hong Kong and School of Computer Science at McGill University, QC, Canada, and was a researcher with Huawei Noah's Ark lab, Hong Kong. He is currently a professor with MOE Key Laboratory for

Intelligent Networks and Network Security at Xi'an Jiaotong University, and is also with Shenzhen Research Institute of Xi'an Jiaotong University, Shenzhen, China. His research interests include Internet traffic measurement and modeling, traffic classification, abnormal detection, and online social network measurement.

Yuchao Zhang received the B.S. in computer science from Xi'an Jiaotong University, Xi'an, China, in 2019. He is currently a graduate student with MOE Key Laboratory for Intelligent Networks and Network Security at Xi'an Jiaotong University. His research interests include Internet traffic measurement and modeling, online social network measurement, and deep learning.

Xiangliang Zhang received the Ph.D. degree in computer science from INRIA-Universite Paris-Sud, France in 2010. She is currently an associate professor and directs the Machine Intelligence and knowledge Engineering Laboratory in King Abdullah University of Science and Technology (KAUST), Saudi Arabia. Her main research interests and experiences are in machine learning, data mining, and cloud computing.

Jing Tao received the B.S and M.S degrees in automatic control from Xi'an Jiaotong University, Xi'an, China, in 2001 and 2006 respectively. He is currently a teacher in Xi'an Jiaotong University. His research interests include Internet traffic measurement and modeling, traffic classification, abnormal detection, and botnet.

Jianwei Ding is a senior engineer in the Science and Technology on Communication Security Laboratory at 30th Research Institute of China Electronics Technology Group Corporation. He received the Ph.D degree in computer science and technology department from Tsinghua University, China, in 2016. His major research interests include dark web threat intelligence, malware analysis, and intelligence security analysis.

Xiaohong Guan received the B.S. (1982) and M.S. degrees (1985) in automatic control from Tsinghua University, Beijing, China, and the Ph.D. degree in electrical engineering from the University of Connecticut, Storrs, US (1993). He is currently a professor at the Systems Engineering Institute, Xi'an Jiaotong University, Xi'an, China. He has been with the Systems Engineering Institute, Xi'an Jiaotong University since 1995, and was appointed Cheung Kong Professor of Systems Engineering in 1999, and dean

of the School of Electronic and Information Engineering in 2008. He has been the director of the Center for Intelligent and Networked Systems since 2001, Tsinghua University, and served as head of the Department of Automation, 2003-2008. He has been elected Fellow of IEEE.

Don Towsley holds a B.A. in Physics (1971) and a Ph.D. in Computer Science (1975) from University of Texas. He is currently a Distinguished Professor at the University of Massachusetts in the Department of Computer Science. He was Program Co-chair of the joint ACM SIGMETRICS and PERFORMANCE 92 conference and the Performance 2002 conference. He is a member of ACM and ORSA, and Chair of IFIP Working Group 7.3. He has received the 2007 IEEE Koji Kobayashi Award, the 2007 ACM

SIGMETRICS Achievement Award, the 1998 IEEE Communications Society William Bennett Best Paper Award, and numerous best conference/workshop paper awards. He has been elected Fellow of both the ACM and IEEE.