Fine-Grained Probability Counting: Refined LogLog Algorithm

Lun Wang[1], Zekun Cai[1], Hao Wang[1], Jie Jiang[1], Tong Yang[1,2], Bin Cui[1], Xiaoming Li[1]
Department of Computer Science and Engineering, Peking University, China[1]
Collaborative Innovation Center of High Performance Computing, NUDT, China[2]
Email: lun.wang@pku.edu.cn

***Abstract*—Estimating the number of distinct flows, also called the *cardinality*, is an important issue in many network applications, such as traffic measurement, anomaly detection, *etc.* The challenging problem is that a high accuracy should be achieved with line speed and small auxiliary memory. The state-of-the-art, *LogLog* algorithm, uses loglogN_{max} memory, where N_{max} is the priori upper bound for cardinality, and achieves an accuracy of the order of $1/\sqrt{d}$, where d is the number of counters. In this paper, we propose a refined version of *LogLog* algorithm, namely *Refined LogLog*. It achieves a much better accuracy than the original *LogLog* algorithm by using more fine-grained common ratios. The algorithm is validated by a detailed analysis. A self-adaptive version, *Self-Adaptive LogLog*, is also proposed based on *Refined LogLog*, to adapt to cardinalities of different scales automatically. Our experimental results show that *Refined LogLog* outperforms *LogLog* in accuracy by up to 67.0%, and reduces the standard deviation by up to 60.8%.**

***Index Terms*—cardinality estimation; network monitoring**

I. Introduction

A. Background

Determining the number of distinct items, namely *cardinality*, is an important issue in many network applications, such as traffic management [1], [2], anomaly detection [3], *etc.* Many database applications, such as database query optimization [4], require fast and accurate estimation of cardinality as well.

There are mainly two kinds of algorithms for cardinality estimation. The first kind of algorithms is based on *packet sampling* (e.g. *Adaptive Sampling* [5]). However, these algorithms suffer from low accuracy and are unacceptable in fine-grained applications. The second kind of algorithms is based on probabilistic counting. One of the earliest probabilistic counting algorithms is *Linear Counting* [6], proposed by Whang, Zanden, and Taylor. But its requirement for linear space makes it unpractical in real networks, especially when the cardinality to record is large. *Multiresolution Bitmap* [7] and *Adaptive Bitmap* [7] are refined versions of Linear Counting and both achieve higher accuracy. However, Multiresolution Bitmap requires large space, and Adaptive Bitmap has many restrictions on the target datsets.

*Corresponding author: Tong Yang (Email: yang.tong@pku.edu.cn). This work was done by Lun Wang, Zekun Cai, and Hao Wang under the guidance of their mentor: Tong Yang. This work is partially supported by Primary Research & Development Plan of China (2016YFB1000304), National Basic Research Program of China (2014CB340400), NSFC (61472009, 61672061), the Open Project Funding of CAS Key Lab of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences.

Flajolet and Martin proposed another algorithm, namely *Flajolet-Martin sketch* [4], using d bitmaps, each of $\log N_{max}$ bits, to record estimated cardinality, and reaches a standard deviation close to $0.78/\sqrt{d}$. In order to achieve a high accuracy, d should be large, and $d \log N_{max}$ bits is costly for the limited memory in routers or switches. This limitation of FM Sketch motivates another more memory-efficient algorithm called *LogLog* [8], in which each counter only takes up $\log \log N_{max}$ bits. However, this algorithm can only reach a standard deviation close to $1.30/\sqrt{d}$. Detailed descriptions of these two algorithms are given in Section II.

B. Limitation and Proposed Solution

From the above discussion, we can see that these methods either suffer from low accuracy or high memory requirement. The key observation is that for FM sketches and *LogLog*, there are huge memory wastes, because cardinality does not reach 2^{20} or more so the high bits of 32-bit bitmaps or 8-bit counters are not necessarily required (more details are provided in Section II). However, these bits are still there to cater to the 8-bit computer architecture to make the data structures faster.

The key problem lies in the fact that common ratio 2 used by the FM Sketch and LogLog algorithm is coarse-grained. The counters can only record $1, 2, 4, \cdots$, which we call recordable numbers. It not only leads to the waste of memory, but causes a poor stability and low accuracy as well, because when the cardinality is large, the gaps between the accurate numbers and the recordable numbers will be large. For a *LogLog* with $d = 64$, it can only guarantee an approximate error of 16.25%. For a *LogLog* with $d = 32$, it can hardly guarantee any approximate error. To address this issue, we propose a refined version of *LogLog* algorithm, namely *Refined LogLog* in this paper, which can reach a 67% better accuracy than the original version. It uses a fine-grained common ratio like $4/3$ or $8/7$ to narrow the gaps between two recordable values.

Another problem facing *LogLog* is that we need to know a priori upper bound of cardinality to help us choose the appropriate size of bitmaps. In real-world networks, we do not have the knowledge in most cases. In this paper, we propose a *Self-Adaptive LogLog* which can adapt to the cardinality by changing its common ratio dynamically, and thus get rid of the priori knowledge. It provides a high accuracy when the cardinality is small and is capable of storing large cardinality as well.

Contributions:

- We propose two ideas: (1) narrowing the intervals of *LogLog* algorithm by using a geometrically hash functions with smaller common ratio, (2) constructing a *LogLog* algorithm which can adapt to cardinality dynamically.
- We implement a C++ library of *Refined LogLog* and *Self-Adaptive LogLog*. The source codes are released at GitHub [9].
- We do an exhaustive mathematical analysis of *Refined LogLog* and prove that *Refined LogLog* is better than *LogLog* theoretically in terms of accuracy and stability.
- We perform extensive experiments using real-world network traces to test the performance of *Refined LogLog* and *Self-Adaptive LogLog* in terms of accuracy and stability.

II. Related Work

A. Flajolet-Martin Sketch

Flajolet and Martin proposed a classic algorithm for approximate cardinality counting, namely FM sketch [4]. FM sketches are widely used in network applications to count flow numbers, such as data dissemination [10], [11] and probabilistic aggregation [12], [13].

As shown in Figure 1, an FM sketch is composed of d bitmaps and each bitmap has w bits. The i^{th} bitmap is denoted by A_i and the jth bit of A_i is denoted by $A_i[j]$. Each bitmap A_i is associated with an independent hash function $h_i(.)$. Specially, $h_i(.)$ maps half of all items to the least significant bit of the i^{th} array, a quarter to bit 1, and so on. The concrete method to generate such geometrically distributed hash functions is discussed in subsection III-A. For each item e, the FM sketch computes d hash functions $h_i(e)$ and sets $A_i[h_i(e)\%w]$ to 1. To answer a query, FM sketch returns $1.2928 \times 2^{\frac{1}{d}\sum_{i=0}^{d-1} L_i}$, where L_i denotes the lowest bit with value 0, as derived in [4].

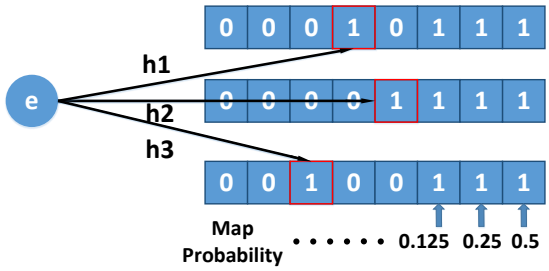


Fig. 1: Structure of a FM Sketch.

B. LogLog Sketch

Durand and Flajolet proposed another classic algorithm, namely *LogLog* algorithm [8], for cardinality estimation. The auxiliary memory required is extremely small, in the order of $\log\log N_{max}$. N_{max} is a **priori** upper bound on cardinality.

As shown in Figure 2, *LogLog* uses d counters and each counter has $w \approx \log\log N_{max}$ bits. The i^{th} counter is denoted by $A[i]$. There is only one hash function $h(.)$. The complete hash value of item e is denoted by $h(e)$. The lowest $k = \log d$ bits of $h(e)$ is denoted by $h_l(e)$ and is used to locate a counter. $h(e)$ is also used to generate geometrically distributed hash values $l_2(h(e))$, through the method discussed in subsection III-A. Then $A[h_l(e)]$ is set to $max\{A[h_l(e)], l_2(h(e))\}$. To answer a query, *LogLog* returns $\alpha_d d 2^{\frac{1}{d}\sum A[i]}$, where $\alpha_d := (\Gamma(-1/d)\frac{1-2^{1/d}}{\log 2})^{-d}$ and $\Gamma(s) := \frac{1}{s}\int_0^\infty e^{-t}t^s dt$, as derived in [8].

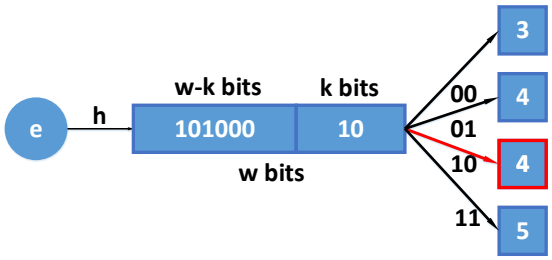


Fig. 2: Structure of *LogLog*.

There are many other data structures which can also be used to augment cardinality extimation such as bloom filter [14], [15], pyramid sketch [16], and cold filter [17], but they are quite different from *Refined LogLog* so we do not find them comparable.

III. Methodology

In this section, we give a detailed description of *Refined LogLog* and *Self-Adaptive LogLog*. First, we introduce the concrete method to generate geometrically distributed hash functions with arbitrary common ratios. Second, we propose the *Refined LogLog* algorithm, which uses more fine-grained common ratio and gives more accurate estimation of cardinality. Third, we propose a dynamic version of *LogLog* algorithm, namely *Self-Adaptive LogLog* which can adapt to different scales of cardinalities without a priori knowledge about upper bound. The symbols we use is defined in Table I.

TABLE I: Symbols used in Section III and IV

Symbol	Description
d	# of bitmaps or counters.
w	# of bits of a bitmap or a counter.
r	rate used in *Refined LogLog* and *Self-Adaptive LogLog*.
μ	$r/(r-1)$
e	An element inserted or queried.
$h(e)$	Uniformly distributed hash function.
udhv	uniformly distributed hash value.
gdhv	geometrically distributed hash value.
A[i]	the i^{th} counter in *Refined LogLog* or *Self-Adaptive LogLog*.
esti	Estimated cardinality.
n	# of Incoming Elements.
$*(i,n,x)$	$i_1 + i_2 + \cdots + i_n = x$
$[z^n]f(z)$	The coefficient of z^n term in $f(z)$.
γ_e	Euler's constant.
//	division operation. The result is truncated to integer.

A. Geometrically Distributed Hash Functions

The most widely used hash functions are uniformly distributed hash functions. These hash functions map the input strings to nearly uniformly distributed binary strings. It is theoretically impossible to find a hash function mapping nonuniform inputs into uniform outputs. But in practice, we can easily find one that is close enough [18].

In cardinality counting algorithms, a geometrically distributed hash function is frequently required. In FM Sketch

and *LogLog*, a geometrically distributed hash function with a common ratio of $1/2$ is generated by the Algorithm 1.

Algorithm 1: Geometrically Distributed Hash Function with Common Ratio 1/2

```
Function Ghash1/2(e):
    udhv = h(e);
    gdhv = the lowest bit's position of udhv with value 1;
    return gdhv;
end
```

We extend the algorithm to generate geometrically distributed hash functions with more choices of common ratio. For a binary string, we define $Lowbit(x, i)$ in Algorithm 2. We use $Lowbit(x, i)$ to replace finding the lowest bit operation in Algorithm 1, and get the extended algorithm 3. We can see that $P\{Ghash(e, i) = v\} = (1 - 1/i)^v$, so we get a geometrically distributed hash function with common ratio $1 - 1/i$. We can set the i and get different common ratios.

Algorithm 2: Subroutine of Geometrically Distributed Hash Function

```
Function Lowbit(x, r):
    res = 0;
    while x%r!=0 do
        x = x/r;
        ++res;
    end
    return res;
end
```

Algorithm 3: Geometrically Distributed Hash Function with More Choices of Common Ratio

```
Function Ghash(e, r):
    udhv = h(e);
    gdhv = Lowbit(udhv, r);
    return gdhv;
end
```

One problem with function $Lowbit(x, i)$ is that the modulo operation is time-consuming and may become a bottleneck for $Ghash(e, i)$. However, if we pick i as 2^k, then we can use shift operation to replace the modulo operation and get an accelerated version of $Lowbit(x, i)$. Thus, we choose to use hash functions with common ratio as $1-(1/2)^k$ in this paper.

B. Refined LogLog Algorithm

Hyper-parameters: *Refined LogLog* is composed of counters initialized to 0. To initialize a *Refined LogLog*, three parameters need tp be determined: 1) the number of counters: d; 2) the size of each counter in terms of bits: w; 3) the rate: r. The selection of these parameters will determine the capacity, accuracy and memory efficiency of the *Refined LogLog*.

Insertion: As shown in Algorithm 4, when an item e is inserted, we calculate the hash value $h(e)$. We use $h(e)\%d$ to locate a counter. If $Ghash(h(e)//d)$ is larger than the counter value, the counter value will updated to $Ghash(h(e)//d)$. Otherwise, the counter will keep its value.

Query: As defined in Algorithm 4, when answering a query, *Refined LogLog* will return $\alpha_{d,r} d(\frac{r}{r-1})^{\frac{1}{d}\sum A[i]}$, where $\alpha_{d,r} := (\Gamma(-1/d)\frac{1-(\frac{r}{r-1})^{1/d}}{\log \frac{r}{r-1}})^{-d}$ and $\Gamma(s) := \frac{1}{s}\int_0^\infty e^{-t}t^s dt$. The detailed derivation is given in Section IV.

Algorithm 4: Refined LogLog

```
struct {
    counter A[d];
    int r;
    Function Insert(e):
        udhv = h(e);
        index = udhv % d; gdhv = Ghash(udhv//d, r);
        A[index] = max{A[index], gdhv};
    end
    Function Query():
        float sum = 0;
        for i in d do
            sum = sum + A[i];
        end
        return α_{d,r} d r (r/(r-1))^{sum};
    end
} RefinedLogLog;
```

C. Self-Adaptive LogLog

Hyper-parameters: To initialize a *Self-Adaptive LogLog*, four parameters need to be determined: 1) the number of counters: d; 2) the size of each counter in terms of bits: w; 3) initial rate: r 4) evolving step size: s.

Insertion: As shown in Algorithm 5, the insertion procedure is similar to *Refined LogLog*. The differences are that it uses an adaptive rate, and when a counter overflows, *Self-Adaptive LogLog* needs to evolve to a new rate to expand the capacity.

Query: The query procedure is similar to *Refined LogLog*, using current value of the adaptive rate.

Evolve: Overflow happens when, during an insertion, the inserted counter cannot hold the new value. This phenomenon indicates that the capacity of the current *Self-Adaptive LogLog* is not big enough to record the cardinality. To address this issue, we propose an "evolve operation". When evolving, the rate of *Self-Adaptive LogLog* is reduced by s, thus increasing the capacity. Then we need to store the previously recorded cardinality into the evolved *Self-Adaptive LogLog*. We first execute a query operation before evolution and get the estimated cardinality: $esti$. Assuming the rate before evolution is r, we can easily find the integer: γ, which satisfies $(\frac{r}{r-1})^{\gamma-1} \leq esti \leq (\frac{r-s}{r-1-s})^{\gamma}$. Then it solves the equation group:

Algorithm 5: Self-Adaptive LogLog

```
struct {
    counter A[d];
    int r;
    int s;
    Function Insert(e):
        /* Same as RefinedLogLog 4        */
        ...;
        if any counter overflows then
            Evolve();
        end
    end
    Function Query():
        /* Same as RefinedLogLog 4        */
        ...;
    end
    Function Evolve():
        r = r - s;
        int estimate = Query();
        Solve the corresponding equation group 1;
        Set the counters as the roots;
    end
} SelfAdaptiveLogLog;
```

$$\begin{cases} x(\frac{r}{r-1})^{\gamma-1} + y(\frac{r-s}{r-1-s})^{\gamma} = esti \\ x + y = 1 \end{cases} \tag{1}$$

We have to use σ/d and $1-\sigma/d$ ($\sigma = 0, 1 \cdots d$) to approximate x and y, so we can randomly set σ counters to $\gamma-1$ and the other counters to γ. For example, to store 7 in a *Refined LogLog* with 32 counters and the common ratio of $4/3$. We set 8 counters to 6 and 24 counters to 7. Figure 3 shows the complete evolving operation.

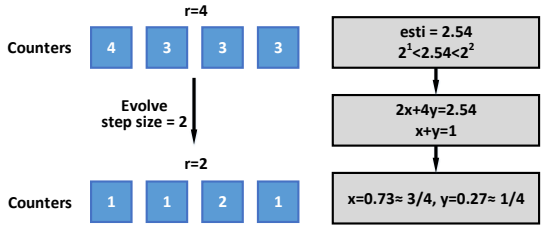


Fig. 3: An Evolving Process.

IV. Analysis

In this section, we give a detailed derivation of the expectation and standard deviation of *Refined LogLog*. The derivation is inspired by [8]. We also give a rough discussion about the evolving operation and its influence on the accuracy. The theoretical analysis shows that *Refined LogLog* not only outperforms *LogLog* by an exponential ratio in terms of accuracy, but are more stable in terms of standard deviation.

A. Preliminary Knowledge

Poissonization and Depoissonization: Poissonization is a calculating technique, using Poisson process to replace the original input. During the replacement, a sequence characterizing the Bernoulli model is mapped to a complex variable's generating function, which characterizes the Poisson model. In *Refined LogLog*'s derivation, we first solve the problem in the Poisson domain. Then it is depoissonized to transform the results back to the original Bernoulli model. We refer readers interested in Poissonization and Depoissonization to [19].

Mellin Transform The Mellin transform [20], [21] associated with a function $f(x)$ is defined as the complex function $f*(s)$ where $f*(s) = \int_0^\infty f(x)x^{s-1}dx$, and it lies in the positive real domain. The major application of the Mellin transform is to calculate the asymptotic analysis of sums obeying the general pattern $G(x) = \sum_k \lambda_k g(\mu_k x)$, which are called harmonic sums connected to the expression $\int_0^\infty \lambda(\kappa)g(\mu(\kappa)x)d\kappa$. In *Refined LogLog*'s derivation, Mellin transform is used to calculate asymptotic analysis of expectation and standard deviation of the estimated value.

B. Theoretical Analysis for Refined LogLog

Theorem 4.1: The asymptotic expectation ε_n and standard deviation ν_n of *Refined LogLog* is shown as below. n is the total number of incoming elements, and $\mu = r/(r-1)$

$$\varepsilon_n = (r-1)n\Gamma(\frac{-1}{d})(\frac{1-\mu^{\frac{1}{d}}}{\log\mu})^{-d}$$
$$\nu_n = (r-1)^2n^2[(\frac{\log\mu}{\Gamma(\frac{-2}{d})\mu^{\frac{2}{d}}})^d - (\frac{\log\mu}{\Gamma(\frac{-1}{d})(1-\mu^{\frac{1}{d}})})^{2d}]$$

Theorem 4.1 points out that the expectation of *Refined LogLog* is unbiased and the standard deviation ($1.06/\sqrt{d}$ when rate = 4, common ratio = 3/4) is theoretically smaller than *LogLog* with standard deviation of $1.30/\sqrt{d}$.

We define $Z := d\mu^{\frac{1}{d}\sum_j M^{(j)}}$. $E_n(Z)$ represents the expectation of it, and $V_n(Z)$ represents the variance of it. According to the definition of expectation, we get the following equation.

$$E_n(Z) = \sum_k (d\mu^{\frac{k}{d}} P(\sum_j M^{(j)} = k)) \tag{2}$$

When a counter receives an element, suppose the corresponding hash value is Y, then we know $P(Y \le k) = \frac{1}{\mu^{k-1}}$. When a counter receives v elements, we define the maximum hash value of the v elements as M. Because hash values of v elements are supposed to be independent, $P_\nu(M \le k) = (1-\frac{1}{\mu^k})^\nu$. Then we get $P_\nu(M=k) = (1-\frac{1}{\mu^k})^\nu - (1-\frac{1}{\mu^{k-1}})^\nu$. The bivariate expotential generative function of $P_\nu(M=k)$ is $G(z,u)$.

$$\begin{aligned} G(z,u) :&= \sum_{v,k} P_\nu(M=k)u^k\frac{z^\nu}{\nu!} \\ &= \sum_k u^k(e^{z(1-\mu^{-k})} - e^{z(1-\mu^{1-k})}) \end{aligned} \tag{3}$$

The probability expression in 2 can be derived as following. For convenience, we use the notation $*(i,n,x)$ to represent $i_1 + i_2 + \cdots + i_n = x$.

$$P(\sum_j M^{(j)} = k) = \frac{1}{d^n} \sum_{*(i,d,n)} \sum_{*(j,d,k)} \frac{n!}{i_1!i_2!..i_d!} (\prod_{l=1}^{d} P_{i_l}(M = j_l)) \tag{4}$$

Then we use the bivariate exponential generating function to calculate the combination of d counters. $[z^n]f(z)$ represents the coefficient of z^n term in $f(z)$.

$$\begin{aligned} G(z,u)^d &= \sum_{\nu,k} \sum_{*(i,d,\nu)} \sum_{*(j,d,k)} \prod_{l=1}^{d} P_{i_l}(M = j_l) u^{j_l} \frac{z^{j_l}}{i_l!} \\ &= \sum_{\nu,k} (\sum_{*(i,d,\nu)} \sum_{*(j,d,k)} \prod_{l=1}^{d} P_{i_l}(M = j_l) \frac{1}{i_l!}) u^k z^\nu \\ [z^n]G(\frac{z}{d},u)^d &= \sum_{k} \sum_{*(i,d,n)} \sum_{*(j,d,k)} \prod_{l=1}^{d} P_{i_l}(M = j_l) \frac{1}{i_l!} u^k \frac{1}{d^n} \end{aligned} \tag{5}$$

According to Equation 2, 5, 4, we can finally get the expression of expectation.

$$E_n(Z) = dn![z^n]G(\frac{z}{d}, \mu^{\frac{1}{d}})^d \tag{6}$$

In the same way , we can get the expectation of Z^2.

$$E_n(Z^2) = d^2 n![z^n]G(\frac{z}{d}, \mu^{\frac{2}{d}})^d \tag{7}$$

Then we can get the variance of Z.

$$\begin{aligned} V_n(Z) &= E_n Z^2 - (E_n Z)^2 \\ &= d^2 n![z^n]G(\frac{z}{d}, \mu^{\frac{2}{d}})^d - (dn![z^n]G(\frac{z}{d}, \mu^{\frac{1}{d}})^d)^2 \end{aligned} \tag{8}$$

Then we use Poisson model [19] and Mellin transform [20] to get the calculable asymptotic expression of $E_n(Z)$ and $V_n(Z)$. Suppose $f(z) := \sum_n E_n(Z) \frac{z^n}{n!}$, then $\varepsilon_n := e^{-\lambda} f(\lambda) = \sum_{\lambda \in N} E_n(Z) e^{-\lambda} \frac{\lambda^n}{n!}$ gives the corresponding expectation under Poisson model, which means that $E_n(Z) \sim \varepsilon_n$. According to the definition of ε_n, we can find that $\varepsilon_n = dG(\frac{n}{d}, \mu^{\frac{1}{d}})^d e^{-n}$. In the same way, $V_n(Z) \sim \nu_n := d^2 G(\frac{n}{d}, \mu^{\frac{2}{d}})^d e^{-n} - (dG(\frac{n}{d}, \mu^{\frac{1}{d}})^d e^{-n})^2$.

According to the Mellin transform [20], when $n \to \infty$,

$$\begin{aligned} \varepsilon_n &= [(\Gamma(\frac{-1}{d}) \frac{1-\mu^{\frac{1}{d}}}{\log \mu})^{-d} + \epsilon_n] \cdot n \\ \nu_n &= [(\Gamma(\frac{-2}{d}) \frac{1-\mu^{\frac{2}{d}}}{\log \mu})^{-d} - (\Gamma(\frac{-1}{d}) \frac{1-\mu^{\frac{1}{d}}}{\log \mu})^{-2d} + \eta_n] \cdot n^2 \end{aligned} \tag{9}$$

$|\epsilon_n|$ and $|\eta_n|$ are bounded by 10^{-6} so we can ignore them.

Define $\alpha_{d,r} := (\Gamma(\frac{-1}{d}) \frac{1-(1/r)^{\frac{1}{d}}}{\log 1/r})^{-d} = e^{-\gamma_e} \sqrt{r} - (\frac{1}{r}\pi^2 + \log^2 \frac{1}{r})/(24\frac{d}{r})$ (γ_e is Euler's constant). When $n \to \infty$, $\frac{1}{n} E_n(E) = \frac{1}{n} E_n(Z)/\alpha_{d,r} = 1 + \theta_{1,n,r} + o(1)$, where $|\theta_{1,n,r}| < 10^{-6}$, and $\sqrt{\frac{1}{n} V_n(E)} = \frac{\beta_{d,r}}{\sqrt{d}} + \theta_{2,n,r} + o(1)$, where $|\theta_{2,n,r}| < 10^{-6}$.

$$\begin{aligned} \nu_n &= (\Gamma(\frac{-2}{d}) \frac{1-\alpha^{\frac{2}{d}}}{\log \alpha})^{-d} - (\Gamma(\frac{-1}{d}) \frac{1-\alpha^{\frac{1}{d}}}{\log \alpha})^{-2d} \\ &= (\Gamma(\frac{-1}{d/2}) \frac{1-\alpha^{\frac{1}{d/2}}}{\log \alpha})^{-2(d/2)} - (\Gamma(\frac{-1}{d}) \frac{1-\alpha^{\frac{1}{d}}}{\log \alpha})^{-2d} \\ &= \alpha^2_{d/2,1/\alpha} - \alpha^2_{d,1/\alpha} \end{aligned} \tag{10}$$

Suppose $A = e^{-\gamma_e}\sqrt{r}$, and $B = (\frac{\pi^2}{r} + \log^2 \frac{1}{r})/24$, we get $\beta_{d,r} = \sqrt{2B - \frac{3B^2}{A} d}$, when $n \to \infty$. $\beta_{\infty,r} = \sqrt{(\frac{\pi^2}{r} + \log^2 \frac{1}{r})/12}$. When $r = 3/4$ and $r = 1/2$, $\beta_{\infty,r} \approx 1.06 < 1.30$.

The last tricky part is that the geometrically hash functions we use to generate a geometrically distributed array has a first term $1-\mu$ different from common ration μ, However, this can be easily dealt with if we consider the array as a geometrically distributed array with μ as both first term and common ratio times a constant $r-1$. Then we can get the final asymptotic expectation and variance as shown in 4.1.

V. Experimental Evaluation

A. Experimental Setup

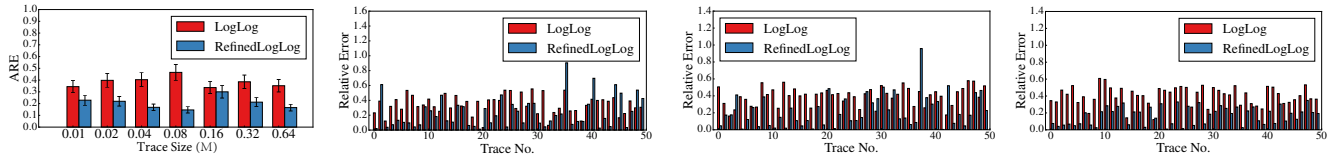
Metrics: We define four metrics to evaluate our algorithm's performances.

- *AAE:* Average absolute error, defined as the average value of absolute error over the dataset number, where absolute error is the absolute value of the difference between accurate value and estimated value.
- *ARE:* Average relative error, defined as the average value of relative error over the dataset number, where relative error is the absolute value of the difference between accurate value and estimated value divided by the accurate value.
- *Insertion Time:* defined as the total time of all insertion operations for a data set.
- *Capacity:* defined as the maximal cardinality of different elements a data structure can record.

Traffic Traces: The traffic traces used in the experiments are collected from a tier-1 router. We identify flows using the standard 5-tuple. The traces approximately obey zipfian distributions [22]. The elements in the context is the coming packets, and the cardinality is the distinct flow number. There are 7 sets of traces, having 0.01M, 0.02M,0.04M, 0.08M, 0.16M, 0.32M, 0.64M incoming packets.

B. Experimental Results

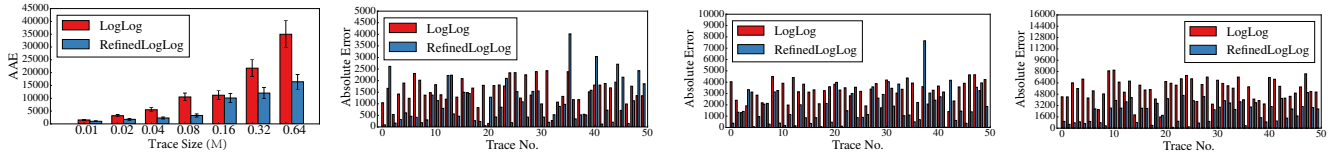
ARE: As shown in Figure 4(a), our experimental results show that *Refined LogLog can reduce the ARE by* $[19.6\%, 67.5\%]$, *with a mean of* 44.1%, compared to the *LogLog*. It also *reduces the standard deviation of ARE by up to* 60.8%*, with a mean of* 33.6%. The results of relative error of each trace in the 7 sets are shown in Figure 4(b) to 4(h). We can see that *Refined LogLog* outperforms *LogLog* in most of the traces no matter what the trace size is.



(a) Summary of ARE Experiments (b) Relative Error of 0.01M Traces (c) Relative Error of 0.02M Traces (d) Relative Error of 0.04M Traces

(e) Relative Error of 0.08M Traces (f) Relative Error of 0.16M Traces (g) Relative Error of 0.32M Traces (h) Relative Error of 0.64M Traces

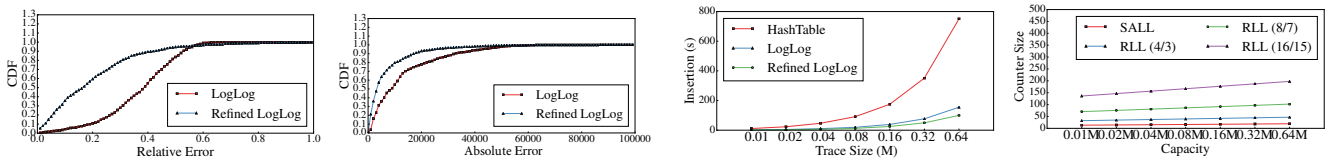
Fig. 4: Relative Error Experimental Results on 7 Set of Traces.



(a) Summary of AAE Experiments (b) Absolute Error of 0.01M Traces (c) Absolute Error of 0.02M Traces (d) Absolute Error of 0.04M Traces

(e) Absolute Error of 0.08M Traces (f) Absolute Error of 0.16M Traces (g) Absolute Error of 0.32M Traces (h) Absolute Error of 0.64M Traces

Fig. 5: Absolute Error Experimental Results on 7 Set of Traces.



(a) CDF of Relative Error (b) CDF of Absolute Error (c) Insertion Time v.s. #Packets (d) Counter Size v.s. Capacity

Fig. 6: Experimental results of CDF, Insertion Time and Capacity.

AAE: As shown in Figure 5(a), our experimental results show that *Refined LogLog can reduce the AAE by* $[19.7\%, 67.6\%]$, *with a mean of* 44.2%, compared to the *LogLog*. It also *reduces the standard deviation of ARE by up to* 61.0%*, with a mean of* 30.0%. The results of absolute error of each trace in the 7 sets are shown in Figure 5(b) to 5(h). We can see that *Refined LogLog* outperforms *LogLog* in most of the traces no matter what the trace size is.

The above two sets of experimental results show that *Refined LogLog* can bring stable improvements in terms of relative error, absolute error, and standard deviation of the estimation, regardless of the dataset size, which conforms with the theoretical analysis. This characteristic indicates that *Refined LogLog* can provide a high accuracy in real-world network environment with different scales of cardinality.

CDF: As shown in Figure 6(a), *over 80% of Refined LogLog's estimated cardinality have a relative error which is smaller than 0.25, while the statistical quantity drops sharply to 20% using LogLog*. As shown in Figure 6(b), *over 79% of Refined LogLog's estimated cardinality have a absolute error which is smaller than 10000, while the statistical quantity drops sharply to 55% using LogLog.*

We can see that *Refined LogLog* is more stable than *LogLog* because more estimated cardinalities have smaller errors. This conforms with the theoretical analysis.

Total Insertion Time vs. Dataset Size: As shown in Figure 6(c), *Refined LogLog can provide a stable speedup of 78% compared with classic hash table and 35.5% compared with LogLog* . This result indicates that our *Refined LogLog* fits the high-speed network environment as well.

Counter Size vs. Capacity: As shown in Figure 6(d), *our Self-Adaptive LogLog can store a cardinality of 0.64M with less than 16 bits, while Refined LogLog will take up to 200 bits*. The numbers between the legends are common ratio used by *Refined LogLog*. The result shows that *Self-Adaptive LogLog* has a far bigger capacity and can adapt to various network environments.

VI. Conclusion

Due to the high requirement of speed and memory in network applications, estimating the cardinality has always been a challenging and important task for algorithm scientists. The state-of-the-art either suffers from low accuracy [5] or specific requirement for datasets [7]. The most widely used data structures for cardinality estimating is FM Sketch [4], *LogLog* [8]. They are widely used in the real networks and other applications [23]–[28] , and have many variants [2], [29], [30].

Motivated by FM Sketch [4] and *LogLog* [8], we observe that the key problem is the common ratio–2 is too coarse-grained for the cardinality estimation. In this paper, we propose to use a much more fine-grained common ratio to replace 2, and reach a higher accuracy and stability. We also propose *Self-Adaptive LogLog*, which can accommodate various network environments. We do an exhaustive analysis of *Refined LogLog* , and the theoretical results show that *Refined LogLog* beats *LogLog* in terms of accuracy and stability. We also give a rough discussion about the mathematical properties of *Self-Adaptive LogLog*, but more detailed work about it needs to be done. We perform extensive experiments and the experimental results show that *Refined LogLog* significantly reduces the fluctuation and reach a much better accuracy. Furthermore, *Refined LogLog* is also faster in insertion by approximately 35% than *LogLog*, which fits well with the high-speed network environments. There are extensive work to do about *Refined LogLog* and *Self-Adaptive LogLog*, like more detailed mathematical analysis and further experiments. We are convinced that *Refined LogLog* and *Self-Adaptive LogLog* will be widely used in network applications in the near future.

References

[1] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high speed links," in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. ACM, 2003, pp. 153–166.

[2] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm," in *Analysis of Algorithms 2007 (AofA07)*, 2007, pp. 127–146.

[3] C. Estan and G. Varghese, *New directions in traffic measurement and accounting*. ACM, 2002, vol. 32, no. 4.

[4] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *Journal of computer and system sciences*, vol. 31, no. 2, pp. 182–209, 1985.

[5] P. Flajolet, "On adaptive sampling," *Computing*, vol. 43, no. 4, pp. 391–400, 1990.

[6] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications," *ACM Transactions on Database Systems (TODS)*, vol. 15, no. 2, pp. 208–229, 1990.

[7] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high speed links," in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. ACM, 2003, pp. 153–166.

[8] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *European Symposium on Algorithms*. Springer, 2003, pp. 605–617.

[9] "Source code for refined loglog," https://github.com/spartazhihu/Refined-LogLog.

[10] C. Lochert, B. Scheuermann, and M. Mauve, "Probabilistic aggregation for data dissemination in vanets," in *Proceedings of the fourth ACM international workshop on Vehicular ad hoc networks*. ACM, 2007, pp. 1–8.

[11] C. Lochert, J. Rybicki, B. Scheuermann, and M. Mauve, "Scalable data dissemination for inter-vehicle-communication: Aggregation versus peer-to-peer (skalierbare informationsverbreitung für die fahrzeug-fahrzeug-kommunikation: Aggregation versus peer-to-peer)," *it-Information Technology*, vol. 50, no. 4, pp. 237–242, 2008.

[12] C. Lochert, B. Scheuermann, and M. Mauve, "A probabilistic method for cooperative hierarchical aggregation of data in vanets," *Ad Hoc Networks*, vol. 8, no. 5, pp. 518–530, 2010.

[13] M. Garofalakis, J. M. Hellerstein, and P. Maniatis, "Proof sketches: Verifiable in-network aggregation," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. IEEE, 2007, pp. 996–1005.

[14] T. Yang, A. X. Liu, M. Shahzad, D. Yang, Q. Fu, G. Xie, and X. Li, "A shifting framework for set queries," *Transactions on Networking*, 2017.

[15] T. Yang, A. X. Liu, M. Shahzad, Y. Zhong, Q. Fu, Z. Li, G. Xie, and X. Li, "A shifting bloom filter framework for set queries," *VLDB*, 2016.

[16] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li, "Pyramid sketch: A sketch framework for frequency estimation of data streams," *VLDB*, 2017.

[17] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold filter: A meta framework for faster and more accurate stream processing," *SIGMOD*, 2018.

[18] D. E. Knuth, *The art of computer programming: sorting and searching*. Pearson Education, 1998, vol. 3.

[19] W. Szpankowski, "Analytic poissonization and depoissonization," *Average Case Analysis of Algorithms on Sequences*, pp. 442–519, 2001.

[20] P. Flajolet, X. Gourdon, and P. Dumas, "Mellin transforms and asymptotics: Harmonic sums," *Theoretical computer science*, vol. 144, no. 1-2, pp. 3–58, 1995.

[21] B. Epstein, "Some applications of the mellin transform in statistics," *The Annals of Mathematical Statistics*, pp. 370–379, 1948.

[22] D. M. Powers, "Applications and explanations of zipf's law," in *Proceedings of the joint conferences on new methods in language processing and computational natural language learning*. Association for Computational Linguistics, 1998, pp. 151–160.

[23] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec, "Hadi: Mining radii of large graphs," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 5, no. 2, p. 8, 2011.

[24] A. Sridharan and T. Ye, "Tracking port scanners on the ip backbone," in *Proceedings of the 2007 workshop on Large scale attack defense*. ACM, 2007, pp. 137–144.

[25] C. Lochert, B. Scheuermann, and M. Mauve, "Probabilistic aggregation for data dissemination in vanets," in *Proceedings of the fourth ACM international workshop on Vehicular ad hoc networks*. ACM, 2007, pp. 1–8.

[26] Q. Han, S. Du, D. Ren, and H. Zhu, "Sas: A secure data aggregation scheme in vehicular sensing networks," in *Communications (ICC), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–5.

[27] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *European Symposium on Algorithms*. Springer, 2003, pp. 605–617.

[28] S. Heule, M. Nunkesser, and A. Hall, "Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm," in *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 2013, pp. 683–692.

[29] Y. Chabchoub and G. Hébrail, "Sliding hyperloglog: Estimating cardinality in a data stream over a sliding window," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1297–1303.

[30] Y. Zhao, S. Guo, and Y. Yang, "Hermes: An optimization of hyperloglog counting in real-time data processing," in *Neural Networks (IJCNN), 2016 International Joint Conference on*. IEEE, 2016, pp. 1890–1895.

[31] "Intuition of poissonization," https://math.stackexchange.com/questions/721195/poissonization-and-intuition.

[32] "Definition of mellin transform," https://en.wikipedia.org/wiki/Mellin_transform.

APPENDIX A. EXPONENTIAL GENRATING FUNCTION

In this appendix, we give the definition of generating function, exponential generating function, multivariable generating function and multivariable exponential generating function. Because they can usually be represented in a close form, they are often used to store large information about sequence, and solve combination problems. We use generating function and multivariable exponential generating function to derive the expectation and variance of *Refined LogLog* estimation in Section IV.

Definition 1: Generating function of a sequence a_n is a function, whose coefficient of x^n term is a_n.

$$G_{a_n}(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_i x^i + \cdots$$

Definition 2: Exponential generating function of a sequence a_n is a function, whose coefficient of x^n term is $a_n/n!$.

$$G_{a_n}(x) = a_0 + a_1 x + a_2 x^2/2! + \cdots + a_i x^i/i! + \cdots$$

Definition 3: Multivariable generating function of multi-dimensional sequences $a_{n0,n1,\ldots,n_k}$ is a function, whose coefficient of $x_0^{n0} x_1^{n1} \cdots x_k^{nk}$ term is $a_{n0,n1,\ldots,n_k}$. Bivariate generating function is multivariable generating function with $k = 2$.

$$\begin{aligned} G_{a_{n,m}}(x,y) = & a_{0,0} & + a_{0,1} y & & + a_2 0, 2y^2 & & + \cdots \\ & a_{1,0} x & + a_{1,1} xy & & + a_2 1, 2xy^2 & & + \cdots \\ & a_{2,0} x^2 & + a_{2,1} x^2 y & & + a_2 2, 2x^2 y^2 & & + \cdots \\ & \cdots & & & & & \end{aligned}$$

Definition 4: Multivariable exponential generating function of multi-dimensional sequences $a_{n0,n1,\ldots,n_k}$ is a function, whose coefficient of $x_0^{n0} x_1^{n1} \cdots x_k^{nk}$ term is $a_{n0,n1,\ldots,n_k}/(n_0! n_1! \cdots n_k!)$. Bivariate generating function is multivariable generating function with $k = 2$.

$$\begin{aligned} G_{a_{n,m}}(x,y) = & a_{0,0} & + a_{0,1} y & & + a_2 0, 2\frac{y^2}{2!} & & + \cdots \\ & a_{1,0} x & + a_{1,1} xy & & + a_2 1, 2x\frac{y^2}{2!} & & + \cdots \\ & a_{2,0} \frac{x^2}{2!} & + a_{2,1} \frac{x^2}{2!} y & & + a_2 2, 2\frac{x^2}{2!}\frac{y^2}{2!} & & + \cdots \\ & \cdots & & & & & \end{aligned}$$

APPENDIX B. POISSONIZATION AND DEPOISSONIZATION

In this appendix, we give the deinition of poissonization and depoissonization. Then we give a intuitive explanation.

Definition 5: Let $a_k (k \in N)$ be a sequence of independent variables with Bernoulli distribution $B(p)$. Let N be a random variable, and define $N' := \sum_{k=1\cdots N} a_k$, and $N'' := \sum_{k=1\cdots N}(1 - a_k)$. If N obeys Poisson distribution $P(\lambda)$, then N' and N'' are independent random variables, and obey Poisson distribution $P(\lambda p)$ and $P(\lambda(1-p))$. Conversely, if N' and N'' are independent, then N obeys Poisson distribution.

An intuitive example [31] of this is that suppose that N represents the number of customers who have arrived to a store up to time t, and that a_k is an indication of whether the kth customer is male. Then N' counts up the number of male customers and N'' counts up the number of female customers. We can know that if the arriving customers obey Poisson distribution, then the arriving male and female customers both obey poissonization independently, vice versa.

APPENDIX C. MELLIN TRANSFORM

In this appendix, we give the definition of Mellin transform and another version of definition in probability theory [32].

Definition 6: In mathematics, the Mellin transform is an integral transform that may be regarded as the multiplicative version of the two-sided Laplace transform. This integral transform is closely connected to the theory of Dirichlet series, and is often used in number theory, mathematical statistics, and the theory of asymptotic expansions; it is closely related to the Laplace transform and the Fourier transform, and the theory of the gamma function and allied special functions.

The Mellin transform of a function f is

$$\{Mf\}(s) = \phi(s) = \int_0^\infty x^{s-1} f(x) dx$$

The inverse transform is

$$\{M^{-1}\phi\}(x) = f(x) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} x^{-s} \phi(s) ds$$

Definition 7: Let X be a random variable, $X^+ = max\{X, 0\}$ be the positive part of it, and $X^- = max\{X, 0\}$ be the negative part of it, then the Mellin transform of X is defined as

$$M_X(s) = \int_0^\infty x^s dF_{X^+}(x) + \gamma \int_0^\infty x^s dF_{X^-}(x),$$

where γ is a formal indeterminate with $\gamma^2 = 1$. This transform exists for all s in some complex strip $D = \{s : a \le Re(s) \le b\}$, where $a \le 0 \le b$.

Mellin transform is an important tool in probability theory, the Mellin transform because it provides a convenient method to study the distributions of products of random variables. The distribution function F_X of a random variable X is uniquely determined by Mellin transform $M_X(s)$. The significance of the Mellin transform in probability theory lies in the fact that if X and Y are two independent random variables, then the Mellin transform of their products is equal to the product of the Mellin transforms of X and Y.

$$M_{XY}(s) = M_X(s) M_Y(s)$$