World Wide Web
https://doi.org/10.1007/s11280-018-0583-0

Fine-grained probability counting for cardinality estimation of data streams

Lun Wang[1] · **Tong Yang**[1] · **Hao Wang**[1] · **Jie Jiang**[1] ·
Zekun Cai[1] · **Bin Cui**[1] · **Xiaoming Li**[1]

Received: 7 March 2018 / Revised: 24 April 2018 / Accepted: 26 April 2018

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 challenge is that high accuracy should be achieved with line speed and small auxiliary memory. Flajolet-Martin algorithm, LogLog algorithm, and HyperLogLog algorithm form a line of work in this area with improving performance. In this paper, we propose refined versions of these algorithms to achieve higher accuracy. The key observations are (1) the "leftmost" hash functions used by these algorithms can be generalized to reach higher accuracy, (2) the amendment coefficient can be highly biased in some certain

This work was done by Lun Wang, Zekun Cai, and Hao Wang under the guidance of their mentor: Tong Yang.

This article belongs to the Topical Collection: *Special Issue on Big Data Management and Intelligent Analytics*
Guest Editors: Junping Du, Panos Kalnis, Wenling Li, and Shuo Shang

✉ Tong Yang
yang.tong@pku.edu.cn; yangtongemail@gmail.com

Lun Wang
lun.wang@pku.edu.cn

Hao Wang
wanghao1996@pku.edu.cn

Jie Jiang
surlavi@pku.edu.cn

Zekun Cai
1400013703@pku.edu.cn

Bin Cui
bin.cui@pku.edu.cn

Xiaoming Li
lxm@pku.edu.cn

[1] Department of Computer Science, Peking University, Beijing, China

streams or datasets so dynamically setting the amendment coefficient instead of using the one derived in pure math can lead to much better accuracy. Experimental results show great improvement of accuracy and stability of the refined versions over original algorithms.

Keywords Cardinality estimation · Probability counting · Network measurement · Data streams

1 Introduction

1.1 Background and motivation

Determining the number of distinct items, namely *cardinality*, is an important issue in many network applications, such as traffic management [7, 10], anomaly detection, etc. Many database applications, such as database query optimization [9], 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* [8]). 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* [33], 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 datasets.

Flajolet and Martin proposed another algorithm, namely *Flajolet-Martin algorithm* [9], 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 algorithm motivates another more memory-efficient algorithm called LogLog algorithm [5], in which each counter only takes up $\log \log N_{max}$ bits. However, LogLog algorithm can only reach a relative accuracy as high as $1.30/\sqrt{d}$. HyperLogLog algorithm is proposed to address the issue and achieves great success by reducing relative accuracy to $1.04/\sqrt{d}$. Detailed descriptions of these algorithms are given in Section 2.

1.2 Limitation of prior art and our proposed solution

From the above discussion, we can see that these methods either suffer from low accuracy or high memory requirement. The key observations are (1) FM algorithm, LogLog algorithm and HyperLogLog algorithm all suffer from large "record gap". For example, a bitmap with leftmost 1 at 18th or 19th position will be regarded as $2^{18} = 262144$ or $2^{19} = 524288$. Then if a stream has a real cardinality of around 30000, then the bitmap will give out pretty rough estimation because the gap of 2 possible records is too large, (2) the amendment coefficient can be highly biased in some streams or datasets. For example, 20 elements with "leftmost" hash positions from 0 to 19 can be recorded as $2^{19} = 524288$ with relative error as high as 26213. Although this example can hardly happen in practice and can be solved using averaging, we observe that in some real-world datasets, the amendment coefficient is still far from good.

The "leftmost" hash functions used by these algorithms can be viewed as a process of consecutive division by 2. In another view, these hash functions can be regarded as geometrically distributed with the common ratio of $1/2$. However, $1/2$ is a too coarse-grained common ratio leading to a huge gap between possible records. For a LogLog algorithm with $d = 64$, it can only guarantee an average error of 16.25%. To address this issue, we propose a generalized version of "leftmost" hash functions, namely geometrically-distributed hash functions. We use it to replace "leftmost" hash functions in FM, LogLog and HyperLogLog algorithms in order to make the gap more fine-grained. Another improvement is that instead of using the amendment coefficient derived by pure math, we dynamically set the coefficient by learning from a small portion of streams or datasets. This may introduce extra overheads in the beginning but will greatly benefit the accuracy in the long term.

Since the paper is expanded from its conference version [30], we want to highlight the new technique contributions here. First, we proposed to dynamically set the amendment coefficient. Second, we add refined version HyperLogLog algorithm. Third, we added a large number of new experiments on real-world datasets to show the efficiency of our refinements.

Our key contributions

- We generalize the "leftmost" hash functions to geometrically-distributed hash functions.
- We propose to dynamically set the amendment coefficient.
- We apply two techniques to Flajolet-Martin, LogLog and HyperLogLog algorithm, and carry out extensive experiments which show great improvement in accuracy and stability.

2 Related work

In this section, we will introduce Flajolet-Martin, LogLog and HyperLogLog algorithm in details. The symbols used are shown in Table 1.

Table 1 Symbols used in the paper

Symbol	Description
n	# of Incoming Elements.
m	# of bitmaps or counters.
w	# of bits of a bitmap
div	dividend used in FM, LogLog and HyperLogLog.
$B[i][j]$	j^{th} bit in i^{th} bitmap in FM.
$\sigma(.)$	the position of leftmost 1 in bits.
$\rho(B[i])$	the position of leftmost 0 in $B[i]$.
$C[i]$	i^{th} counter in LogLog or HyperLogLog.
e	a coming element like a flow in a stream.
$h_u(e)$	Uniformly distributed hash function.
$h_g(e, div)$	geometrically distributed hash value.
//	division operation. The result is truncated to integer.

Table 2 Algorithms, memory cost and relative accuracy

Algorithm	Memory	(units)	Relative Accuracy
Flajolet-Martin	m	bitmaps (64 bits)	$0.78/\sqrt{m}$
LogLog	m	counters (1 byte)	$1.30/\sqrt{m}$
HyperLogLog	m	counters (1 byte)	$1.04/\sqrt{m}$

2.1 The Flajolet-Martin algorithm

Flajolet and Martin proposed a classic algorithm for approximate cardinality counting, namely Flajolet-Martin algorithm [9]. FM sketches are widely used in network applications to count flow numbers, such as data dissemination [19, 20] and probabilistic aggregation [11, 21]. The memory cost and theoretical accuracy of FM algorithm is shown in Table 2.

As shown in Figure 1, FM algorithm is composed of d bitmaps and each bitmap has w bits. The i^{th} bitmap is denoted by $B[i]$ and the jth bit of $B[i]$ is denoted by $B[i][j]$. Each bitmap $B[i]$ is associated with an independent hash function $h_g^{(i)}(.,.)$. Specially, $h_g^{(i)}(.,.)$ is called "leftmost" hash functions. It maps half of all items to the leftmost bit of the i^{th} array, a quarter to bit 1, and so on. The concrete method to generate such functions is discussed in Section 3.1. For each item e, FM algorithm computes d hash functions $h_g^{(i)}(e, 2)$ and sets $B[i][h_g^{(i)}(e, 2)\%w]$ to 1. To answer a query, FM sketch returns $1.2928 \times 2^{\frac{1}{d}\sum_{i=0}^{d-1}\rho(B[i])}$, where $B[i]$ denotes the lowest bit with value 0, as derived in [9].

2.2 LogLog and HyperLogLog algorithm

Durand and Flajolet proposed another classic algorithm, namely LogLog algorithm [5], 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. Flajolet et al. proposed HyperLogLog algorithm [10] to improve the accuracy of LogLog algorithm. The memory cost and theoretical accuracy of LogLog and HyperLogLog algorithm is shown in Table 2.

As shown in Figure 2, LogLog algorithm uses d counters and each counter has $\log\log N_{max}$ bits. The i^{th} counter is denoted by $C[i]$. There is only one hash function $h_u(.)$. The complete hash value of item e is denoted by $h_U(e)$. The lowest $k = \log d$ bits of $h_u(e)$ is denoted by $h_l(e)$ and is used to locate a counter. The higher bits $h_h(e)$ of $h_u(e)$ is also used to generate "leftmost" hash values $h_g(e, 2))$ with the method discussed in Section 3.1. Then $C[h_l(e)]$ is set to $max\{C[h_l(e)], h_g(e, 2)\}$. To answer a query, LogLog algorithm returns $\alpha_d d 2^{\frac{1}{d}\sum C[i]}$, 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 [5].

HyperLogLog does many refinements on LogLog from practical perspectives. In this paper, we only focus on the replacement of arithmetic mean by harmonic mean. The insertion is the same as LogLog. To answer a query, HyperLogLog returns $\alpha_d d 2^{\frac{d}{\sum 1/C[i]}}$.

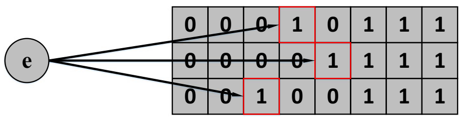


Figure 1 Structure of a FM Sketch

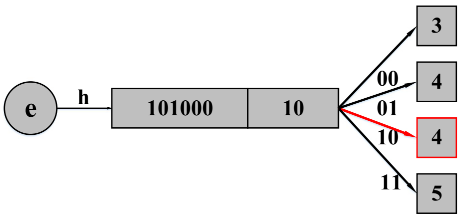


Figure 2 Structure of *LogLog*

There are many more sketches in the literature. Interested readers please refer to [2–4, 6, 13, 17, 18, 25–29, 31, 32, 34].

3 Methodology

In this section, we give a detailed description of our algorithms. First, we introduce how to generalize "leftmost" hash functions to geometrically distributed hash functions. Second, we present how to dynamically set the amendment coefficient. Then, we propose refined FM algorithm, refined LogLog algorithm and refined HyperLogLog algorithm which use geometrically distributed hash functions and dynamic amendment coefficient. The symbols we use are defined in Table 1.

3.1 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 [16].

In cardinality counting algorithms, a "leftmost" hash function is frequently required. The Pseudocode is shown in Algorithm 1. "Leftmost" hash function can be viewed as a process of consecutive division by 2. The hash value is the number of iterations before the remainder is 0.

Algorithm 1 "Leftmost" Hash Function

```
Function LeftMostHash(e):
    $uniform\_hash\_value = h_u(e)$;
    $geometrical\_hash\_value = \sigma(uniform\_hash\_value)$;
    return $geometrical\_hash\_value$;
end
```

Table 3 Derived amendment coefficients

Algorithm	Amendment Coefficient
Flajolet-Martin	1.29281
LogLog	0.39701
HyperLogLog	0.39701

Algorithm 2 Geometrically Distributed Hash Function

```
Function ConsecutiveDivide(uniform_hash _value, div):
    $res = 0$;
    while uniform_hash_value%r != 0 do
        $uniform\_hash\_value = uniform\_hash\_value/div$;
        $++res$;
    end
    return $res$;
end
Function GDHash(e, div):
    $uniform\_hash\_value = h_u(e)$;
    $geometrical\_hash\_value = ConsecutiveDivide(uniform\_hash\_value, div)$;
    return $geometrical\_hash\_value$;
end
```

As shown in Algorithm 2, we extend the algorithm to generate geometrically distributed hash functions with more choices of common ratio. For a binary string, we define a subroutine called $ConsecutiveDivide(.,.)$. We use $ConsecutiveDivide(.,.)$ to replace $\sigma(.)$ in Algorithm 1, and get the extended algorithm 2. We can see that the hash values should be a geometrically distributed array with common ratio $(div-1)/div$ and first term div.

One problem with function $ConsecutiveDivide(.,.)$ is that the modulo operation is time-consuming and may become a bottleneck for $GDHash(.,.)$. However, if we pick div as 2^k, then we can use shift operation to replace the modulo operation and get an accelerated version of $ConsecutiveDivide(x,i)$. Thus, we choose to use geometrically distributed hash functions with $div = 4$ in this paper.

3.2 Dynamic amendment coefficient

In order to get unbiased estimate of cardinality, the proposers of FM, LogLog and HyperLogLog derived an amendment coefficient with pure math as shown in Table 3. However, we observe in practice that these coefficients can be highly biased when facing certain streams or datasets. We propose to use a small portion of the stream or dataset to learn the coefficients instead of using the derived ones. The coefficients we use in the experiments are shown in Table 4.

Table 4 Learned amendment coefficient when $n = 1000$

Dataset/Algorithm	FM	LogLog	HyperLogLog
Synthetic	0.829	4.81	1.95
Self-Collected	0.608	3.541	1.333
CAIDA	0.806	4.680	1.723
Penn Treebank	0.771	9.693	4.743

3.3 Refined algorithms

3.3.1 Refined Flajolet-Martin algorithm

Refined Flajolet-Martin algorithm is composed of bitmaps initialized to 0. To initialize a refined FM, three parameters need to be determined: 1) the number of bitmaps: d; 2) the size of each bitmap in terms of bits: w; 3) the dividend used in geometrically distributed hash functions: div. The selection of these parameters will determine the capacity, accuracy and memory efficiency of the refined FM algorithm.

Insertion As shown in Algorithm 3, when an item e is inserted, for each bitmap, we calculate a uniform hash value $h_u(e)$. Then we calculate the geometrical hash value and set the corresponding bit to 1.

Query As defined in Algorithm 3, when answering a query, each bitmap refined FM will return $amend \times div(\frac{div-1}{div})^{pos} B[i]$. The whole algorithm will return the average of these estimates.

Algorithm 3 Refined Flajolet-Martin Algorithm

```
struct {
    bitmap $B[d]$;
    int $div$;
    Function Insert(e):
        $uniform\_hash\_value = h_u(e)$;
        for each bitmap $B[i]$ do
            $geometrical\_hash\_value = h_g^{(i)}(uniform\_hash\_value, div)$;
            $B[i][geometrical\_hash\_value] = 1$;
        end
    end
    Function Query():
        float $sum = 0$;
        for $i$ in $d$ do
            $pos = \rho(B[i])$;
            $sum = sum + amend \times div(\frac{div-1}{div})^{pos} B[i]$;
        end
        return $sum/d$ ;
    end
} RefinedFM;
```

3.3.2 Refined LogLog algorithm

Refined LogLog algorithm is composed of counters initialized to 0. To initialize a refined LogLog, three parameters need to be determined: 1) the number of counters: d; 2) the size of each counter in terms of bits: w; 3) the dividend: div. 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 $h_u(e)$. We use $h_u(e)\%d$ to locate a counter. If $GDHash(h_u(e)//d)$ is larger than the counter value, the counter value will be updated to $GDHash(h_u(e)//d)$. Otherwise, the counter will keep its value.

Query As defined in Algorithm 4, when answering a query, refined LogLog will return $amend_{d,r}div \times d(\frac{div}{div-1})^{sum/d}$, where sum is the sum of all counters.

Algorithm 4 Refined LogLog Algorithm

```
struct {
    counter $C[d]$;
    int $div$;
    Function Insert(e):
        $uniform\_hash\_value = h_u(e)$;
        $index = uniform\_hash\_value \% d$; $geometrical\_hash\_value =$
        $GDHash(uniform\_hash\_value//d, div)$;
        $C[index] = \max\{C[index], geometrical\_hash\_value\}$;
    end
    Function Query():
        float $sum = 0$;
        for $i$ in $d$ do
            $sum = sum + C[i]$;
        end
        return $amend \times div \times d(\frac{div}{div-1})^{sum/d}$;
    end
} RefinedLogLog;
```

3.3.3 Refined HyperLogLog algorithm

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

Insertion As shown in Algorithm 5, when an item e is inserted, we calculate the hash value $h_u(e)$. We use $h_u(e)\%d$ to locate a counter. If $GDHash(h_u(e)//d)$ is larger than the counter value, the counter value will be updated to $GDHash(h_u(e)//d)$. Otherwise, the counter will keep its value.

Query As defined in Algorithm 5, when answering a query, refined HyperLogLog will return $amend_{d,r}div \times d(\frac{div}{div-1})^{d/sum}$, where sum is the sum of the inverses of all counters.

Algorithm 5 Refined HyperLogLog Algorithm

```
struct {
    counter $C[d]$;
    int $div$;
    Function Insert(e):
        $uniform\_hash\_value = h_u(e)$;
        $index = uniform\_hash\_value \ \% \ d$; $geometrical\_hash\_value =$
        $GDHash(uniform\_hash\_value//d, div)$;
        $C[index] = \max\{C[index], geometrical\_hash\_value\}$;
    end
    Function Query():
        float $sum = 0$;
        for $i$ in $d$ do
            $sum = sum + 1/C[i]$;
        end
        return $amend \times div \times d(\frac{div}{div-1})^{d/sum}$;
    end
} RefinedHyperLogLog;
```

4 Experimental results

4.1 Experimental setup

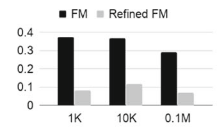
Basic settings We run the experiments on a Dell Inspiron-15 5000 series Notebook PC with Intel(R) Core(TM) i7- 4510U CPU @2.00GHz 2.60GHz, 8.00 GB memory and Ubuntu 14.04 LTS Desktop system. All the codes are open-sourced at Github [22].

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

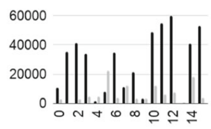
- *AAE:* Average absolute error, defined as the average value of absolute error over the dataset number, where absolute error is the absolute value of the difference between accurate value and estimated value.
- *ARE:* Average relative error, defined as the average value of relative error over the dataset number, where relative error is the absolute value of the difference between accurate value and estimated value divided by the accurate value.

Datasets

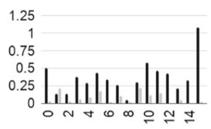
- *Synthetic Strings:* This dataset contains randomly generated strings within length of 128. The elements in the context are the next string, and the cardinality is the distinct string number.
- *Self-Collected Traces:* This dataset contains traffic traces collected from a tier-1 router. We identify flows using the standard 5-tuple. The traces approximately obey zipfian distribution. The elements in the context are random strings, and the cardinality is the distinct flow number.



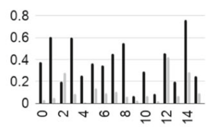
(a) Summary of ARE Experiments



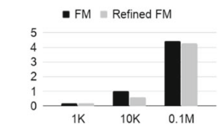
(b) Relative Error of 1K Synthetic



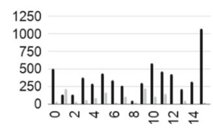
(c) Relative Error of 10K Synthetic



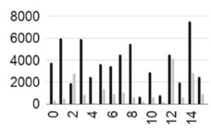
(d) Relative Error of 0.1M Synthetic



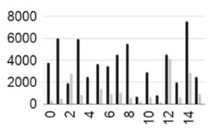
(e) Summary of AAE Experiments



(f) Absolute Error of 1K Synthetic



(g) Absolute Error of 10K Synthetic



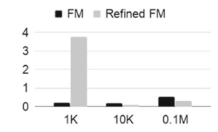
(h) Absolute Error of 0.1M Synthetic

Figure 3 Experimental results of refined Flajolet-Martin algorithms on synthetic strings

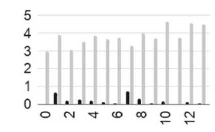
– *CAIDA:* CAIDA dataset contains anonymous passive traffic traces from CAIDA's equinix-chicago monitor on high-speed Internet backbone links. The elements in the context are the coming packets, and the cardinality is the distinct flow number.
– *Penn TreeBank:* The Penn Treebank dataset [23] selected 2,499 stories from a three year Wall Street Journal (WSJ) collection of 98,732 stories for syntactic annotation. The elements in the context are words, and the cardinality is the distinct word number.

4.2 Experimental results

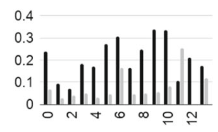
Figures 3, 4, 5, 6, 7, 8, 9, 10, 11 12, 13 and 14 show all the experimental results. The first figure of each line shows a summary–the average of the following 3 figures. The following three figures show the $n = 1000$, $n = 10000$, and $n = 100000$.



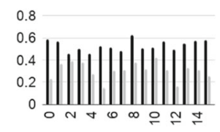
(a) Summary of ARE Experiments



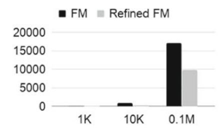
(b) Relative Error of 1K Traces



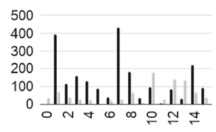
(c) Relative Error of 10K Traces



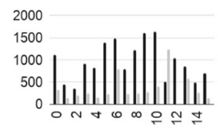
(d) Relative Error of 0.1M Traces



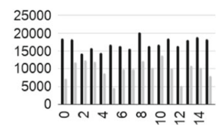
(e) Summary of AAE Experiments



(f) Absolute Error of 1K Traces

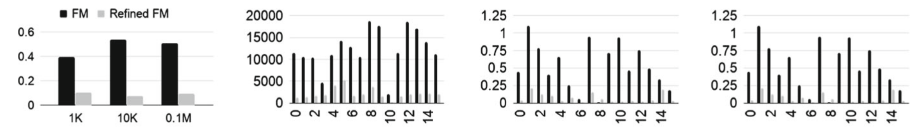


(g) Absolute Error of 10K Traces

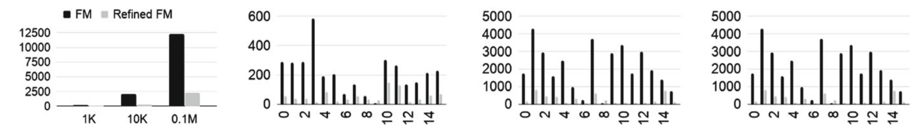


(h) Absolute Error of 0.1M Traces

Figure 4 Experimental results of refined Flajolet-Martin algorithms on self-collected traces



(a) Summary of ARE Experiments (b) Relative Error of 1K CAIDA (c) Relative Error of 10K CAIDA (d) Relative Error of 0.1M CAIDA



(e) Summary of AAE Experiments (f) Absolute Error of 1K CAIDA (g) Absolute Error of 10K CAIDA (h) Absolute Error of 0.1M CAIDA

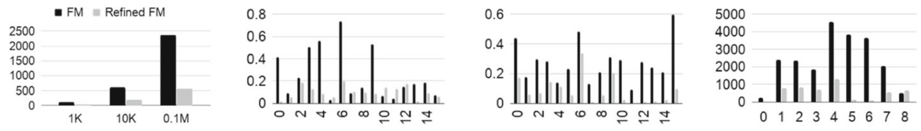
Figure 5 Experimental results of refined Flajolet-Martin algorithms on CAIDA

4.2.1 *Refined Flajolet-Martin algorithm*

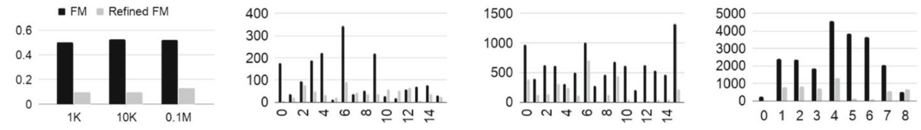
The experimental results of refined Flajolet-Martin algorithm is shown in Figures 3, 4, 5 and 6.

Synthetic strings As shown in Figure 3, our experimental results show that refined Flajolet-Martin algorithm can reduce AAE and ARE by 77.88%, 68.47%, 76.15% on synthetic strings datasets with size 1K, 10K, and 0.1M, compared to the original Flajolet-Martin algorithm.

Self-collected traces As shown in Figure 4, our experimental results show that refined Flajolet-Martin algorithm can reduce the AAE and ARE by 57.03%, 64.18%, 42.77% on self-collected traces datasets with size 1K, 10K, and 0.1M, compared to the original Flajolet-Martin algorithm.

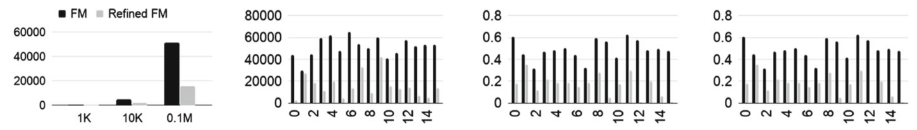


(a) Summary of ARE Experiments (b) Relative Error of 1K PTB (c) Relative Error of 10K PTB (d) Relative Error of 0.1M PTB

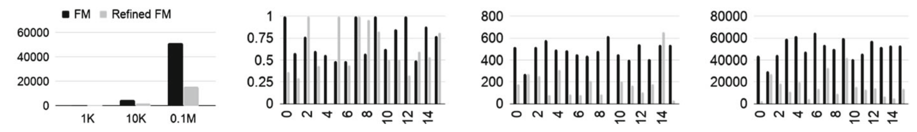


(e) Summary of AAE Experiments (f) Absolute Error of 1K PTB (g) Absolute Error of 10K PTB (h) Absolute Error of 0.1M PTB

Figure 6 Experimental results of refined Flajolet-Martin algorithms on Penn TreeBank



(a) Summary of ARE Experiments (b) Relative Error of 1K Synthetic (c) Relative Error of 10K Synthetic (d) Relative Error of 0.1M Synthetic



(e) Summary of AAE Experiments (f) Absolute Error of 1K Synthetic (g) Absolute Error of 10K Synthetic (h) Absolute Error of 0.1M Synthetic

Figure 7 Experimental results of refined LogLog algorithms on synthetic strings

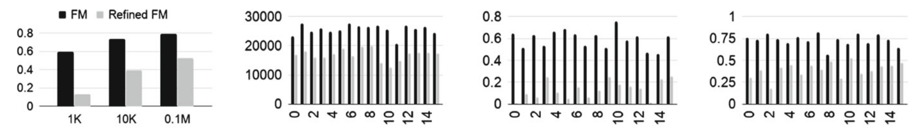
CAIDA As shown in Figure 5, our experimental results show that refined Flajolet-Martin algorithm can reduce the AAE and ARE by 74.39%, 85.76%, 81.94% on CAIDA datasets with size 1K, 10K, and 0.1M, compared to the original Flajolet-Martin algorithm.

Penn Treebank As shown in Figure 6, our experimental results show that refined Flajolet-Martin algorithm can reduce the AAE and ARE by 59.36%, 68.41%, 76.03% on Penn Treebank datasets with size 1K, 10K, and 0.1M, compared to the original Flajolet-Martin algorithm.

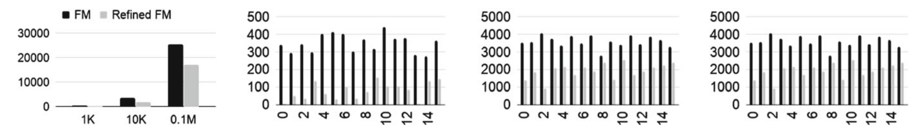
4.2.2 Refined LogLog algorithm

The experimental results of Refined LogLog algorithm are shown in Figures 7, 8, 9 and 10.

Synthetic strings As shown in Figure 7, our experimental results show that refined LogLog algorithm can reduce the AAE and ARE by 62.35%, 65.86%, 69.34% on synthetic strings datasets with size 1K, 10K, and 0.1M, compared to the original LogLog algorithm.

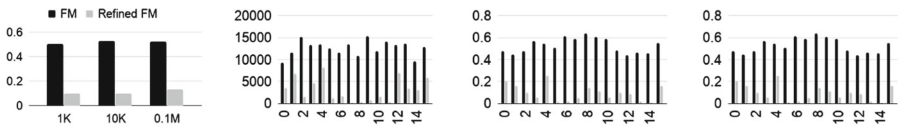


(a) Summary of ARE Experiments (b) Relative Error of 1K Traces (c) Relative Error of 10K Traces (d) Relative Error of 0.1M Traces

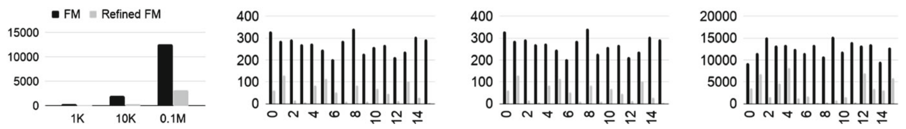


(e) Summary of AAE Experiments (f) Absolute Error of 1K Traces (g) Absolute Error of 10K Traces (h) Absolute Error of 0.1M Traces

Figure 8 Experimental results of refined LogLog algorithms on self-collected traces



(a) Summary of ARE Experiments (b) Relative Error of 1K CAIDA (c) Relative Error of 10K CAIDA (d) Relative Error of 0.1M CAIDA



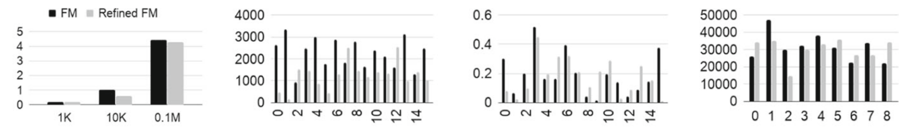
(e) Summary of AAE Experiments (f) Absolute Error of 1K CAIDA (g) Absolute Error of 10K CAIDA (h) Absolute Error of 0.1M CAIDA

Figure 9 Experimental results of refined LogLog algorithms on CAIDA

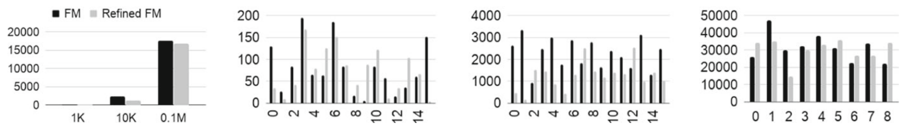
Self-collected traces As shown in Figure 8, our experimental results show that refined LogLog algorithm can reduce the AAE and ARE by 77.11%, 46.05%, 33.56% on self-collected traces datasets with size 1K, 10K, and 0.1M, compared to the original LogLog algorithm.

CAIDA As shown in Figure 9, our experimental results show that refined LogLog algorithm can reduce the AAE and ARE by 80.91%, 81.47%, 74.69% on CAIDA datasets with size 1K, 10K, and 0.1M, compared to the original LogLog algorithm.

Penn Treebank As shown in Figure 10, our experimental results show that refined LogLog algorithm can reduce the AAE and ARE by 7.43%, 44.68%, 4.48% on Penn Treebank datasets with size 1K, 10K, and 0.1M, compared to the original LogLog algorithm.

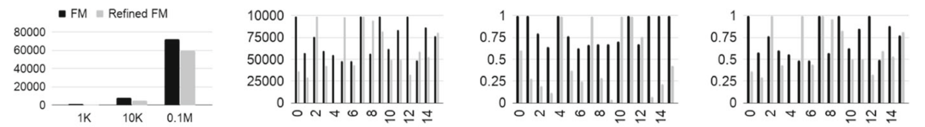


(a) Summary of ARE Experiments (b) Relative Error of 1K PTB (c) Relative Error of 10K PTB (d) Relative Error of 0.1M PTB

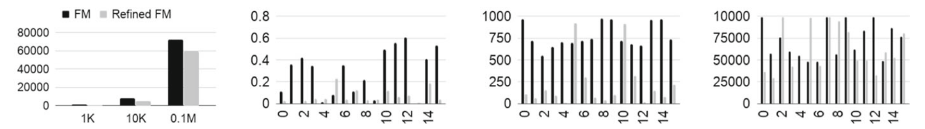


(e) Summary of AAE Experiments (f) Absolute Error of 1K PTB (g) Absolute Error of 10K PTB (h) Absolute Error of 0.1M PTB

Figure 10 Experimental results of refined LogLog algorithms on Penn TreeBank



(a) Summary of ARE Experiments (b) Relative Error of 1K Synthetic (c) Relative Error of 10K Synthetic (d) Relative Error of 0.1M Synthetic



(e) Summary of AAE Experiments (f) Absolute Error of 1K Synthetic (g) Absolute Error of 10K Synthetic (h) Absolute Error of 0.1M Synthetic

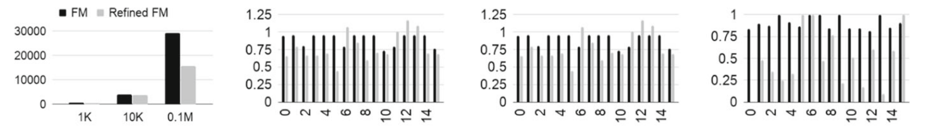
Figure 11 Experimental results of refined HyperLogLog algorithms on synthetic strings

4.2.3 Refined HyperLogLog algorithm

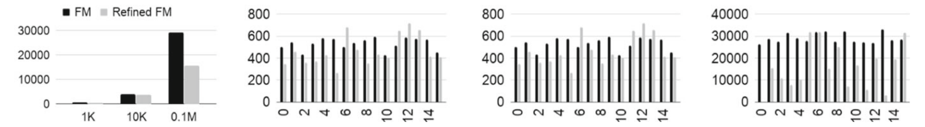
The experimental results of Refined HyperLogLog algorithm are shown in Figure 11, 12, 13 and 14. We can tell that except a few exceptions, refined algorithms show a stable improvement in accuracy compared the original algorithms.

Synthetic strings As shown in Figure 11, our experimental results show that refined HyperLogLog algorithm can reduce the AAE and ARE by 71.29%, 42.47%, 17.75% on synthetic strings datasets with size 1K, 10K, and 0.1M, compared to the original HyperLogLog algorithm.

Self-collected traces As shown in Figure 12, our experimental results show that refined HyperLogLog algorithm can reduce the AAE and ARE by 12.76%, 8.00%, 45.93% on

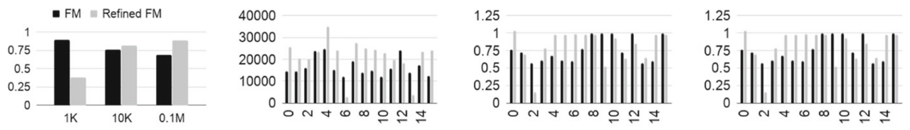


(a) Summary of ARE Experiments (b) Relative Error of 1K Traces (c) Relative Error of 10K Traces (d) Relative Error of 0.1M Traces

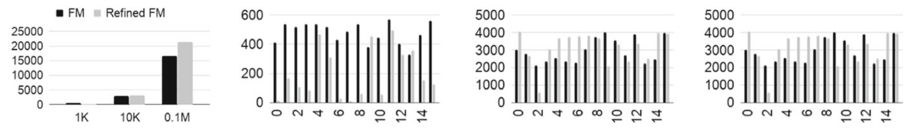


(e) Summary of AAE Experiments (f) Absolute Error of 1K Traces (g) Absolute Error of 10K Traces (h) Absolute Error of 0.1M Traces

Figure 12 Experimental results of refined HyperLogLog algorithms on self-collected traces



(a) Summary of ARE Experiments (b) Relative Error of 1K CAIDA (c) Relative Error of 10K CAIDA (d) Relative Error of 0.1M CAIDA



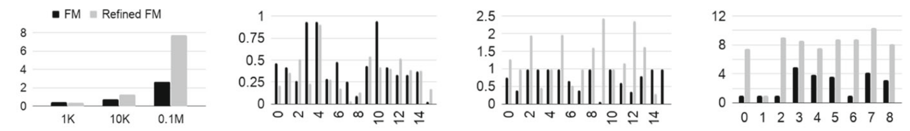
(e) Summary of AAE Experiments (f) Absolute Error of 1K CAIDA (g) Absolute Error of 10K CAIDA (h) Absolute Error of 0.1M CAIDA

Figure 13 Experimental results of refined HyperLogLog algorithms on CAIDA

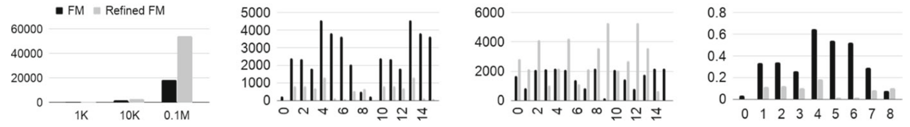
self-collected traces datasets with size 1K, 10K, and 0.1M, compared to the original HyperLogLog algorithm.

CAIDA As shown in Figure 13, our experimental results show that refined HyperLogLog algorithm can reduce the AAE and ARE by 57.88%, -7.53%, -29.18% on CAIDA datasets with size 1K, 10K, and 0.1M, compared to the original HyperLogLog algorithm.

Penn Treebank As shown in Figure 14, our experimental results show that refined HyperLogLog algorithm can reduce the AAE and ARE by 19.48%, -64.92%, -192.76% on Penn Treebank datasets with size 1K, 10K, and 0.1M, compared to the original HyperLogLog algorithm. Note that refined HyperLogLog on CAIDA and Penn Treebank are the only two experiments showing refined algorithm has lower accuracy than original algorithm. This illustrates that although refined algorithms are better in most cases, we cannot guarantee them to be always better than original ones.



(a) Summary of ARE Experiments (b) Relative Error of 1K PTB (c) Relative Error of 10K PTB (d) Relative Error of 0.1M PTB



(e) Summary of AAE Experiments (f) Absolute Error of 1K PTB (g) Absolute Error of 10K PTB (h) Absolute Error of 0.1M PTB

Figure 14 Experimental results of refined HyperLogLog algorithms on Penn TreeBank

5 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 [8] or specific requirement for datasets [7]. The most widely used data structures for cardinality estimating are FM algorithm [9], LogLog algorithm [5] and HyperLogLog algorithm [10]. They are widely used in the real networks and other applications [5, 12, 14, 15, 19, 24] , and have many variants [1, 10, 35].

Motivated by FM [9], LogLog [5] and HyperLogLog [10], we observe that the key problem is (1) the common ratio–2 is too coarse-grained for the cardinality estimation, (2) the amendment coefficient can sometimes be highly biased. 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 to set amendment coefficient dynamically, which can accommodate various network environments.

We perform extensive experiments and the experimental results show that refined FM, LogLog and HyperLogLog significantly reduce the fluctuation and reach a much better accuracy.

There are extensive work to do about refined FM, LogLog and HyperLogLog algorithms, like more detailed mathematical analysis and further experiments. We are convinced that refined FM, LogLog and HyperLogLog will be widely used in network applications in the near future.

Acknowledgments This work is partially supported by Primary Research, Development Plan of China (2016YFB1000304), National Basic Research Program of China (2014CB340405), NSFC (61672061), the OpenProject Funding of CAS Key Lab of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences.

References

1. Chabchoub, Y., Hébrail, G.: Sliding hyperloglog: estimating cardinality in a data stream over a sliding window. In: IEEE International Conference on Data Mining Workshops (ICDMW), pp 1297–1303. IEEE (2010)
2. Dai, H., Shahzad, M., Liu, A.X., Zhong, Y.: Finding persistent items in data streams. Proc. VLDB Endow. **10**(4), 289–300 (2016)
3. Dai, H., Zhong, Y., Liu, A.X., Wang, W., Li, M.: Noisy bloom filters for multi-set membership testing. In: ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science, pp. 139–151 (2016)
4. Dai, H., Meng, L., Liu, A.X.: Finding persistent items in distributed, datasets. In: Proceedings of the 37th Annual IEEE International Conference on Computer Communications (INFOCOM) (2018)
5. Durand, M., Flajolet, P.: Loglog counting of large cardinalities. In: European Symposium on Algorithms, pp. 605–617. Springer (2003)
6. Estan, C., Varghese, G.: New directions in traffic measurement and accounting. ACM, **32**(4) (2002)
7. Estan, C., Varghese, G., Fisk, M.: Bitmap algorithms for counting active flows on high speed links. In: Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement, pp. 153–166. ACM (2003)
8. Flajolet, P.: On adaptive sampling. Computing **43**(4), 391–400 (1990)
9. Flajolet, P., Martin, G.N.: Probabilistic counting algorithms for data base applications. J. Comput. Syst. Sci. **31**(2), 182–209 (1985)
10. Flajolet, P., Fusy, É., Gandouet, O., Meunier, F.: Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. Anal. Algor. **2007**(AofA07), 127–146 (2007)
11. Garofalakis, M., Hellerstein, J.M., Maniatis, P.: Proof sketches: Verifiable in-network aggregation. In: IEEE 23rd International Conference on Data Engineering, 2007. ICDE 2007, pp. 996–1005. IEEE (2007)

12. Han, Q., Du, S., Ren, D., Zhu, H.: Sas: a secure data aggregation scheme in vehicular sensing networks. In: IEEE International Conference on Communications (ICC), pp. 1–5. IEEE (2010)
13. Han, J., Zheng, K., Sun, A., Shang, S., Wen, J.-R.: Discovering neighborhood pattern queries by sample answers in knowledge base. In: 2016 IEEE 32nd International Conference on Data Engineering (ICDE), pp. 1014–1025. IEEE (2016)
14. Heule, S., Nunkesser, M., Hall, A.: 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, pp. 683–692. ACM (2013)
15. Kang, U., Tsourakakis, C.E., Appel, A.P., Faloutsos, C., Leskovec, J.: Hadi: mining radii of large graphs. ACM Trans. Knowl. Discov. Data (TKDD) **5**(2), 8 (2011)
16. Knuth, D.E.: The art of computer programming: sorting and searching, vol. 3. Pearson Education (1998)
17. Li, Z., Xiao, F., Wang, S., Pei, T., Li, J.: Achievable rate maximization for cognitive hybrid satellite-terrestrial networks with af-relays. IEEE Journal on Selected Areas in Communications (2018)
18. Liu, J., Zhao, K., Sommer, P., Shang, S., Kusy, B., Jurdak, R.: Bounded quadrant system: Error-bounded trajectory compression on the go. In: IEEE 31st International Conference onData Engineering (ICDE), pp. 987–998. IEEE (2015)
19. Lochert, C., Scheuermann, B., Mauve, M.: Probabilistic aggregation for data dissemination in vanets. In: Proceedings of the Fourth ACM International Workshop on Vehicular ad hoc Networks, pp. 1–8. ACM (2007)
20. Lochert, C., Rybicki, J., Scheuermann, B., Mauve, M.: 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 **50**(4), 237–242 (2008)
21. Lochert, C., Scheuermann, B., Mauve, M.: A probabilistic method for cooperative hierarchical aggregation of data in vanets. Ad Hoc Netw. **8**(5), 518–530 (2010)
22. Open-source codes, https://github.com/spartazhihu/Fine-Grained-Probability-Counting-Algorithms
23. Penn tree bank dataset, https://catalog.ldc.upenn.edu/ldc99t42
24. Sridharan, A., Ye, T.: Tracking port scanners on the ip backbone. In: Proceedings of the 2007 Workshop on Large Scale Attack Defense, pp. 137–144. ACM (2007)
25. Tong, Y., Chen, L., Cheng, Y., Yu, P.S.: Mining frequent itemsets over uncertain databases. Proc. VLDB Endow **5**(11), 1650–1661 (2012)
26. Tong, Y., Chen, L., Ding, B.: Discovering threshold-based frequent closed itemsets over probabilistic data. In: 2012 IEEE 28th International Conference on Data Engineering (ICDE), pp. 270–281. IEEE (2012)
27. Tong, Y., Chen, L., Yu, P.S.: Ufimt: an uncertain frequent itemset mining toolbox. In: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1508–1511. ACM (2012)
28. Tong, Y.-X., Chen, L., She, J.: Mining frequent itemsets in correlated uncertain databases. J. Comput. Sci. Technol. **30**(4), 696–712 (2015)
29. Tong, Y., Zhang, X., Chen, L.: Tracking frequent items over distributed probabilistic data. World Wide Web **19**(4), 579–604 (2016)
30. Wang, L., Cai, Z., Wang, H., Jiang, J., Yang, T., Cui, B., Li, X.: Fine-grained probability counting. Refined loglog algorithm. IEEE Bigcomp (2018)
31. Wei, Z., Liu, X., Li, F., Shang, S., Du, X., Wen, J.-R.: Matrix sketching over sliding windows. In: Proceedings of the 2016 International Conference on Management of Data, pp. 1465–1480. ACM (2016)
32. Wei, S.W.S.S.Z., He, X., Xiao, X., Wen, J.R.: Topppr: top-k personalized pagerank queries with precision guarantees on large graphs. In: SIGMOD. ACM (2018)
33. Whang, K.-Y., Vander-Zanden, B.T., Taylor, H.M.: A linear-time probabilistic counting algorithm for database applications. ACM Trans. Database Syst. (TODS) **15**(2), 208–229 (1990)
34. Yang, B., Guo, C., Jensen, C.S., Kaul, M., Shang, S.: Stochastic skyline route planning under time-varying uncertainty. In: 2014 IEEE 30th International Conference on Data Engineering (ICDE), pp. 136–147 (2014)
35. Zhao, Y., Guo, S., Yang, Y.: Hermes: an optimization of hyperloglog counting in real-time data processing. In: 2016 International Joint Conference on Neural Networks (IJCNN), pp. 1890–1895. IEEE (2016)