Estimating Cardinality for Arbitrarily Large Data Stream With Improved Memory Efficiency

Qingjun Xiao, *Member, IEEE, ACM*, Shigang Chen, *Fellow, IEEE*, You Zhou, and Junzhou Luo, *Member, IEEE, ACM*

Abstract— **Cardinality estimation is the task of determining the number of distinct elements (or the cardinality) in a data stream, under a stringent constraint that the input data stream can be scanned by just one single pass. This is a fundamental problem with many practical applications, such as traffic monitoring of high-speed networks and query optimization of Internet-scale database. To solve the problem, we propose an algorithm named HLL-TailCut, which implements the estimation standard error $1.0/\sqrt{m}$ using the memory units of four or three bits each, whose cost is much smaller than the five-bit memory units used by HyperLogLog, the best previously known cardinality estimator. This makes it possible to reduce the memory cost of HyperLogLog by 20%∼45%. For example, when the target estimation error is 1.1%, state-of-the-art HyperLogLog needs 5.6 kilobytes memory. By contrast, our new algorithm only needs 3 kilobytes memory consumption for attaining the same accuracy. Additionally, our algorithm is able to support the estimation of very large stream cardinalities, even on the Tera and Peta scale.**

Index Terms— **Data streams, cardinality estimation, random hashing.**

I. Introduction

CARDINALITY estimation is the task of determining the number of distinct elements in a data stream, which is presented as a sequence of elements and can be examined by only one pass. This problem has attracted significant attention over the past decades, due to its important role in many application domains, e.g., real-time traffic monitoring in high-speed networks [5], [11]–[14], [20] or in software-defined networks [21], query plan optimization in large-scale database [10], in-network query aggregation in wireless sensor networks [17], and file significance evaluation in P2P systems [18].

Practical Importance: In the domain of online traffic monitoring of high-speed networks, the cardinality estimation problem can be used to detect traffic anomalies, such as network IP/port scan and distributed denial-of-service (DDoS) attacks [11], [12], [20]. For instance, if we treat all the packets originated from a same source IP as a data stream, then we can detect whether this source IP is a network scanner by counting the number of distinct destination IP/port addresses in its outward packet stream. A similar estimator can be used to detect whether a server is under DDoS attack, if we treat all the packets towards a common destination IP as a data stream and estimate the number of distinct source addresses in this stream. For other application examples, a server farm may learn the popularity of its hosted contents by tracking the number of distinct users that request for each file, and an institutional gateway may perform cardinality estimation on outbound URL requests to measure the popularity of external web content for caching priority.

According to a recent paper [10], many data analysis systems developed by Google, including Sawzall, Dremel and PowerDrill, need to estimate the cardinalities of very large data sets (e.g., the number of distinct search queries on google) on a daily basis. As pointed out in [10], cardinality estimation over large data sets presents a challenge in terms of computational resources, and memory in particular; for PowerDrill, a non-negligible fraction of queries historically could not be computed since they exceeded the available memory.

Prior Art: Although the cardinality can be easily computed using space *linear* in the cardinality, for many applications, this is impractical as it requires too much memory. Therefore, a large number of algorithms have been developed to produce an approximate estimation of the cardinality based on a summary or "sketch" of the data stream, whose occupied space in memory is merely on a *sublinear* level. Typical sketch-based algorithms include PCSA [9], MultiresolutionBitmap [7] (a generalization of LinearCounting [19]), MinCount [3], [4], LogLog [6], HyperLogLog [8], and just list a few.

We make a quick comparison of existing cardinality estimators in Table I. In the third column, each register may be a partial machine word of a few bits, independently producing a coarse estimation of the cardinality (or say, a machine word may hold multiple registers). To mitigate the high variation of a single register and improve the estimation accuracy, a number m of registers must be used. The second column presents the relationship between the standard error

Manuscript received September 21, 2017; revised August 4, 2019; accepted October 22, 2019; approved by IEEE/ACM Transactions on Networking Editor E. Yeh. Date of publication March 19, 2020; date of current version April 16, 2020. This work was supported in part by the National Natural Science Foundation of China under Grant 61872080, in part the Jiangsu Provincial Key Laboratory of Computer Networking Technology, in part by the National Science Foundation of the United States under Grant STC-1562485 and Grant CNS-1719222, and in part by the Florida Cybersecurity Center. The preliminary version of this article titled "Better with fewer bits: Improving the performance of cardinality estimation of large data streams" was published in the proceedings of the IEEE INFOCOM, 2017. *(Corresponding author: Qingjun Xiao.)*

Qingjun Xiao is with the School of Cyber Science and Engineering (Jiangsu Provincial Key Laboratory of Computer Networking Technology), Southeast University, Nanjing 210096, China (e-mail: csqjxiao@seu.edu.cn).

Shigang Chen and You Zhou are with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611 USA (e-mail: sgchen@cise.ufl.edu; youzhou@cise.ufl.edu).

Junzhou Luo is with the School of Computer Science and Engineering (Key Lab of Computer Network and Information Integration, Ministry of Education), Southeast University, Nanjing 210096, China (e-mail: jluo@seu.edu.cn).

Digital Object Identifier 10.1109/TNET.2020.2970860

TABLE I
A COMPARISON OF POPULAR CARDINALITY ESTIMATORS

Algorithm	**Std. Err.(σ)**	**Mem Units**	**Mem(σ=2%)**
MinCount	$1.00/\sqrt{m}$	32-bit keys	10000 bytes
PCSA	$0.78/\sqrt{m}$	32-bit registers	6084 bytes
MultiresBitmap	$\approx 4.4/\sqrt{m}$	1 bit	6050 bytes
LogLog	$1.30/\sqrt{m}$	5-bit registers	2641 bytes
HyperLogLog	$1.04/\sqrt{m}$	5-bit registers	1690 bytes
HLL-TailCut	$1.04/\sqrt{m}$	4-bit registers	1352 bytes
HLL-TailCut+	$1.0/\sqrt{m}$	3-bit registers	938 bytes

and the value of m, where m refers to the number of registers (or the number of bits for MultiresolutionBitmap, or the number of memory units used by MinCount). The total memory cost of an estimator is m multiplied by the size of a register (or 1 bit for MultiresolutionBitmap, or 32 bits for MinCount).

In the last column, we list the memory needed by each algorithm to control the standard error around 2% of the actual cardinality, which shows the progress in memory saving over the past decades: If we use PCSA as the initial benchmark, the seminal work of LogLog reduces the memory cost by more than half. The followup HyperLogLog (HLL) further cuts the memory cost by over 30%. Therefore, HLL is the state-of-the-art algorithm and has been widely adopted by IT industries, such as Google [10], Ask.com [16], PostgreSQL, file-sharing P2P systems [18], and network security systems for DDoS and scan detection [8], just to list a few.

It may appear that the cardinality estimators in Table I already have small memory overhead (on the scale of KBs), and meanwhile can provide good estimation accuracy of about 2% error. Further reducing their memory cost does not seem to be a critically important issue. However, many applications need a large quantity of estimators to work simultaneously. Take network traffic anomaly detection as an example. A core router often receives millions of traffic flows in just a few minutes. In order to monitor the behavior of all flows, it has to allocate a cardinality estimator for each flow [11], [12], [20]. For Google's applications, the number of estimators that work simultaneously becomes much larger, greater than one billion under extreme cases [10]. Hence, the total memory overhead, which is the per-estimator memory cost multiplied by the number of estimators, will be a huge value that could easily overwhelm the memory available on devices that maintain these estimators. For example, on a high-speed router, the on-chip SRAM available for online anomaly detection is merely on the scale of MBs [11], [12], [20], and on Google servers, the DRAM available for tracking keyword popularity is also limited, typically on the scale of GBs for a commodity server [10]. As a summary, reducing the memory cost of a single estimator is an important problem with practical value.

Our Contribution: This paper will present a new cardinality estimator named HLL-TailCut. As shown at the bottom row of Table I, when comparing with the state-of-the-art HyperLogLog, our algorithm can reduce the memory consumption of a single estimator again by 20%$\sim$45%. A great contribution is that we reduce the size of each register from 5 bits to 4 bits (or 3 bits) without degrading the accuracy in cardinality estimation, which represents an extreme in compactness that has not been achieved before. Our technique is called *long tail cutoff* that compresses the information across all registers and meanwhile reduces the variance among the registers, which in turn reduces the standard error in cardinality estimation. Consequently, not only do we have smaller-size registers, but also use fewer registers to attain the same accuracy if compared with the previous algorithms [6], [8], [9]. Moreover, unlike HyperLogLog which has limited operating range within 10^9, our algorithm can support the counting of data streams at the Tera or Peta scale. It has no estimation bias on the entire measurement range, even when handling small cardinalities.

The rest of the paper is organized as follows. Section II reviews the development history of cardinality estimation algorithms. Section III elaborates the best known algorithm called HyperLogLog to motivate our work. Section IV revisits the problem of cardinality estimation and solves it by maximum likelihood estimation. Section V proposes a technique of long tail cutoff to reduce the memory of HyperLogLog, which however causes negative estimation bias. Section VI proposes our HLL-TailCut+ algorithm based on maximum likelihood estimation to address the negative bias issue. Section VII evaluates the performance of proposed algorithm by simulations. Section VIII concludes the paper.

II. RELATED WORK

Cardinality estimation problem is to count the number of distinct elements in a stream, wherein each element is allowed to appear more than once. A key challenge is that the stream of elements can be scanned by just one pass to obtain the result, due to the constraint of limited processing time or memory.

Linear-Space Solutions: A naive solution for this problem is to use a hash table to memorize all the elements seen so far, in order to filter the duplicated ones. This solution has the advantage of knowing the *exact* cardinality. But it needs memory linear to the stream cardinality, which in most applications, is far too large to be kept in available memory.

A well-known algorithm that can *approximate* the stream cardinality is LinearCounting (LC) [19]. It distributes all the stream elements uniformly among a bit array, so that each element can be encoded as the index of a bit in the array. Duplicated stream elements will be mapped to the same bit index, and hence are filtered automatically. LC can provide the best accuracy among all the known cardinality estimators, however under a strict condition that there is sufficient memory space roughly linear to the cardinality [16]. Otherwise, its accuracy will degrade severely. Since our interest is to estimate very large cardinality values on Giga or Tera scale, LC is no longer attractive, as it requires too much memory.

Sublinear-Space Solutions: Researchers have developed a whole range of algorithms that requires only sublinear memory space [3], [4], [6]–[9]. A frequently used method for reducing memory cost is sampling. An example is Multiresolution-Bitmap [7] that designs a sequence of LC structures, whose sampling probabilities decrease exponentially. Another example is MinCount [3], which records only the k smallest hash values for a stream of data items. For both algorithms, their memory efficiency is worse than LogLog and HyperLogLog, as reported by a comparison study [2].

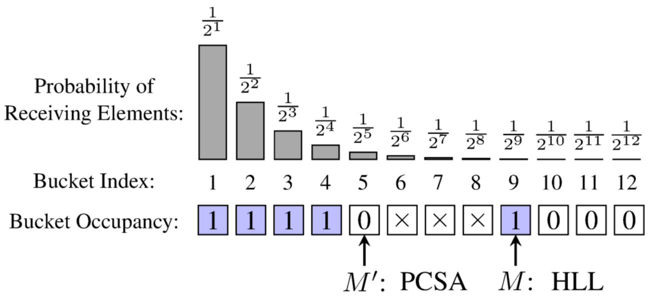
PCSA (Probabilistic Counting with Stochastic Averaging) also prepares a sequence of sampled subsets, but it reduces their sampling probability *exponentially*, until the probability becomes so small that a sampled subset has no data [9]. For the ease of understanding, the sequence of sampled subsets is depicted in Fig. 1 as a sequence of buckets, whose probability of receiving stream elements reduces by the series $2^{-1}, 2^{-2}, 2^{-3}, \dots, 2^{-w}$. To record whether each bucket has received any stream elements, PCSA allocates a bit array in memory: If a bucket receives nothing and remains empty, its corresponding bit is zero; Otherwise, the bit is one. The $\times$ mark in Fig. 1 represents that a bit is either zero or one.

By maintaining the state of this bit array upon stream element arrivals, PCSA always knows the index of the leftmost empty bucket, which is denoted in Fig. 1 by the symbol M'. Such a bit array is called a *register*, which can give an independent estimation of stream cardinality as $2^{M'}$. Hence, if a PCSA register is given w bits memory, the range of its estimated cardinality is as large as 2^w, which is a key advantage of PCSA. Of course, the estimation by a single register will be highly inaccurate. For improving the accuracy, PCSA uses a technique called *stochastic averaging* that allocates multiple registers to produce independent estimations, and returns the average value of their estimations.

The memory efficiency of PCSA still leaves much space for improvement: Its register size must be $\log_2\ n_{\max} + O(1)$ bits, where $n_{\max}$ is the upper bound of measured cardinality. In contrast, a follow-up algorithm called LogLog reduces the memory per register to only $\log_2 \log_2\ n_{\max} + O(1)$ bits [6]. Such significant memory compression is because, instead of maintaining the state of entire bit array like PCSA, LogLog records only the index of the rightmost non-empty bucket, which is denoted by the symbol M in Fig. 1.

HyperLogLog (HLL) is a variant of LogLog for improving accuracy [8]. Both of them depend on the observation of position M shown in Fig. 1, but they adopt different methods for aggregating the estimation results by a set of registers. LogLog uses geometric averaging, while HLL uses harmonic mean, and its purpose is to mitigate the impact of *outlier registers* with abnormally large estimations, thereby appreciably increasing the quality of estimations. As shown in Table I, the expected error of HLL is $1.04/\sqrt{m}$, which is much smaller than that of LogLog $1.30/\sqrt{m}$. In a word, HyperLogLog is the state-of-the-art algorithm.

After years of development, it appears to be very difficult to further compress the memory cost of a cardinality estimator. However, our HLL-TailCut estimator can save memory cost of HLL again by 20%∼45%, based on a long tail cutting technique to be proposed in this paper. Our estimator can reduce the size of a register to four bits (or three bits), which is much smaller than the five-bit register used by HLL, and meanwhile it provides the expected relative error of $1.0/\sqrt{m}$. Therefore, our HLL-TailCut+ algorithm can both reduce the per-register memory cost, and discard large outliers to improve accuracy.



Probability of Receiving Elements:
$\frac{1}{2^1}$ $\frac{1}{2^2}$ $\frac{1}{2^3}$ $\frac{1}{2^4}$ $\frac{1}{2^5}$ $\frac{1}{2^6}$ $\frac{1}{2^7}$ $\frac{1}{2^8}$ $\frac{1}{2^9}$ $\frac{1}{2^{10}}$ $\frac{1}{2^{11}}$ $\frac{1}{2^{12}}$
Bucket Index: 1 2 3 4 5 6 7 8 9 10 11 12
Bucket Occupancy: 1 1 1 1 0 × × × 1 0 0 0
M': PCSA
M: HLL

Fig. 1. Observation used by PCSA and HyperLogLog.

III. Traditional HyperLogLog

In this section, we introduce the traditional HyperLogLog (HLL) algorithm by details, and then identify its inadequacies, which motivate the design of our own algorithms.

A. Basic Idea of HyperLogLog

For the ease of understanding, we firstly explain the estimation procedure of a single HLL register. As shown in Fig. 1, when this register receives a stream of elements, it distributes these elements exponentially among a sequence of buckets, i.e., the probability for the buckets to receive elements reduces exponentially by the series $2^{-1}, 2^{-2}, 2^{-3}, \dots$.

For implementing this exponential distribution, a hash function h is applied to each stream element e. Let us focus on the binary representation of a hash value $h(e)$. The probability of observing the bit pattern $0^{\rho-1}1$ at its beginning is $1/2^{\rho}$, where ρ is one plus the number of leading zeros. For instance, if the hash value $h(e)$ has no leading zeros, then $\rho(1\dots) = 1$, and the probability of observing the bit pattern is $1/2^1$. If there are three leading zeros, then $\rho(0001\dots) = 4$, and the chance of observing the bit pattern is $1/2^4$. Therefore, we can simply regard the symbol ρ as the index of the bucket a stream element e has been mapped to.

A HLL register will record the largest ρ value for all its input elements, or say, the register will record the position of the rightmost non-empty bucket, which is denoted by M in Fig. 1. Because the probability for this bucket to receive elements is $1/2^M$, intuitively, a good estimation for the number of elements the register receives could be 2^M.

However, the cardinality estimation 2^M by a single register is highly variant. For mitigating the high variance, a technique called *stochastic averaging* is adopted: The input data stream $\mathcal{S}$ is pseudorandomly split into m substreams and then fed into m registers. Each register counts the cardinality of its input substream independently. When needed, their results are aggregated to estimate the cardinality of the data stream $\mathcal{S}$.

B. Detailed Algorithm Procedure

Suppose we have allocated m registers $M_0, M_1, \ldots, M_{m-1}$. The procedure of HyperLogLog can be divided into two parts: an online component that processes each stream element and records critical information into the set of registers, and an offline analysis component that recovers the stream cardinality information from the register set.

Online Component: For an element in stream $\mathcal{S}$, we apply the hash function h to it[1], and the resultant hash value is denoted by x. For the binary representation of x, let j be its initial p bits, where $p = \log_2\ m$ or $m = 2^p$, and let x' be its remaining bits:

$$x = h(e), \quad j = \langle x_1 x_2 \cdots x_p \rangle, \ x' = \langle x_{p+1} x_{p+2} \cdots \rangle.$$

The integer j decides that the register M_j receives this stream element. The integer x' is a hash value that updates M_j:

$$M_j := \max\big(M_j,\, \rho(x')\big), \tag{1}$$

where $:=$ is the assignment operator, and $\max(a, b)$ is a function that returns the greater value of its two parameters. As stated before, $\rho(x')$ is one plus the number of leading zeros in the binary format of x', for instance, $\rho(0001\ldots){=}4$. Hence, when the jth substream is nonempty, the register M_j records the index of the *rightmost nonempty bucket* as in Fig. 1.

Offline Analysis Component: Each register M_j in the register set with $0 \leq j < m$ can give an estimation 2^{M_j} for the cardinality of its substream. For aggregating the substream cardinalities, HLL uses the normalized harmonic mean:

$$\hat{n} = \alpha_m \cdot m^2 \cdot \big(\textstyle\sum_{0 \leq j < m} 2^{-M_j}\big)^{-1}, \tag{2}$$

where α_m is a bias correction constant: $\alpha_{16} = 0.673$, $\alpha_{32} = 0.697$, $\alpha_{64} = 0.709$, $\alpha_m = 0.7213/(1{+}1.079/m)$ if $m \geq 128$.

This formula can produce unbiased estimation results only when the cardinality n is large enough. HyperLogLog has proposed to use LinearCounting when $\hat{n}$ by (2) is smaller than $2.5m$. The formula of LinearCounting is $\hat{n} = -m \log(z/m)$, where z is the number of registers M_j that are equal to zeros.

C. Shortcomings of HyperLogLog

HyperLogLog is an excellent algorithm that provides the relative standard error $\frac{1.04}{\sqrt{m}}$ at the cost of $5m$ bits memory. Its high accuracy and memory compactness have triggered extensive adoption in IT industries, e.g., Google [10], Ask.com [16] and PostgreSQL. However, this algorithm still possesses two inadequacies which open doors to further improvements.

[1]The hash function h recommended by a technical blog [1] is Murmur3 hash or City hash functions, which performs better than Jenkins and Spooky hash. Moreover, since the stream cardinality could be as large as 10^9, the hash function must be 64 bits long (rather than 32 bits), in order to make negligibly small the chance of hash collisions among different stream elements [10].

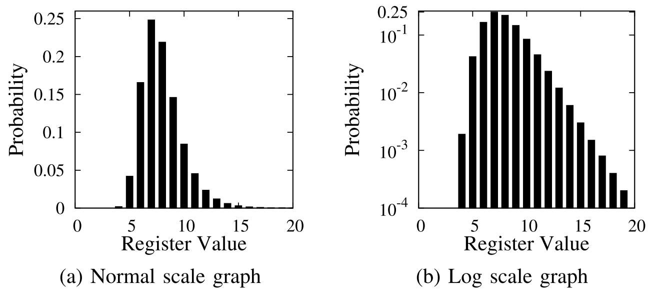


Fig. 2. Probability of register values, when $m = 512$ and load factor $\frac{n}{m} = 100$.

Threat of Outliers: As mentioned before, the observation used by HyperLogLog, which is the value of each register M_j, is highly variant. To give an impression of the high variance, we illustrate in Fig. 2 the probabilistic distribution for a register to carry an arbitrary value k. The plot (a) is drawn in normal scale, and the plot (b) is drawn by log scale for Y-axis. The mathematical formula of this probabilistic distribution will be described later in Eq. (3). Here, the two plots show that it is a *right-skewed distribution* with a long tail stretching out to the right side of the peak. Note that this property of M_j distribution has no relation with the input stream data. It originates from the uniform distribution of hash function h.

The registers whose value strongly deviates from the peak are called *outliers*, which are most likely to exist on the right-side long tail of the distribution as illustrated in Fig. 2(b). In order to mitigate the impact of the outliers existing on the right tail that have abnormally large register values, HLL adopts harmonic average to aggregate the estimation results of a register set. Our intuition is to completely remove the impact of large outliers, by cutting off the right-side long tail on such a histogram, which contains plenty of outliers instead of useful information. It may appear that the outlier rejection can be easily implemented by discarding the registers whose values are much larger than the average. However, the difficulty is that, as the stream cardinality n increases, the register value distribution will move right as a whole. So does the tail cutoff point which separates the main part of the distribution and the rightside long tail with outliers. Then, the problem is how to maintain the cutoff point dynamically as more elements arrive.

Inefficient Register Encoding: The second inadequacy of HyperLogLog is its inefficient encoding of each register state. The number of bits given to each register is a central problem for the design of a cardinality estimator. If a register can be given a smaller number of bits, more registers can be created from a pre-allocated memory block to support more accurate cardinality estimation. For HyperLogLog, the size of a register is five bits long, so that the cardinality estimation by a single register can be up to $2^{2^5} \approx 4 \times 10^9$. With the surging demand to process Internet-scale big data, a cardinality estimator needs to support the counting at the tera- or peta-scale. A recent paper proposed to expand the register size to six bits, to support this large cardinality scenario [10]. On the contrary, we propose to reduce the register size to four bits or even to three bits, and meanwhile support the same large operating range.

Our inspiration comes from the Fig. 2(b), where only the sixteen highest bars between 4 and 19 have their probabilities larger than 0.01%. It implies that, when the number of registers m is on the scale of thousands, the spread of a register set (i.e., the largest register value minus the smallest register value) is less than sixteen in most cases. From the perspective of information theory, it is redundant to use five bits to encode each register. Four bits may be sufficient for most cases.

Moreover, in Fig. 2(a), only for the eight highest bars between 5 and 12, their probabilities are greater than 2%. These eight bars are the most informative part of a histogram, and other bars are more prone to contain outliers, which implies the possibility of abandoning the rightside tail for outlier rejection and giving each register only three bits.

Conclusion: Our idea is that the memory per register may be compressed to four bits or even three bits. Due to the smaller register size, we can precisely record the frequencies only for a few highest bars of the register value distribution in Fig. 2. The outlier bars in the rightside long tail and far from the peak have to be truncated. Later in Section V-C, we will quantify the information loss due to tail truncation, by analyzing the probability of "overflow" event for a register. Thanks to the smaller register size, we can allocate a larger number of registers from the same memory budget, which drives down the estimation error. Hence, what we have proposed is *lossy compression of registers*, and the challenge is how to avoid its side effect.

IV. MLE-Based HyperLogLog

HyperLogLog has a minor inadequacy: When estimating small cardinalities between $2m$ and $5m$, its formula in (2) is strongly biased, as shown later by Fig. 8(a) in the simulation section. This is the region where HyperLogLog makes a switch between LinearCounting and the closed-form formula in (2). To ensure unbiased cardinality estimations in the entire measurement range, we will apply an alternative estimator to fine tune the result, if the estimated cardinality by (2) falls in the biased region. Our alternative formula is based on MLE (maximum likelihood estimation). The analysis in this section is the theoretical foundation of our algorithm to present later.

A. Maximum Likelihood Estimator

In this subsection, we will present a maximum likelihood estimator for the number of distinct elements in a data stream.

The following theorem presents the probabilistic distribution for a HLL register to take a k value, which is shown in Fig. 2.

Theorem 1 (Probability of Register Value): Given m HyperLogLog registers and suppose the cardinality is n, the probability for the jth register M_j to take a certain value k is

$$Pr\{M_j = k\} = \begin{cases} \left(1-\frac{1}{m}\right)^n & \text{if } k=0,\\ (1-\frac{1}{m2^k})^n - (1-\frac{1}{m2^{k-1}})^n & \text{if } k \geq 1. \end{cases} \tag{3}$$

Proof: Check Appendix VIII for a detailed proof. □

For an arbitrary non-negative k value, let N_k be the number of registers, among the m registers, which carry the k value. If observing exactly N_k registers carrying a particular k value, the probability of this observation is $Pr\{M_j = k\}^{N_k}$, assuming these registers are mutually independent. Then, the combined probability of making the observations $N_0, N_1, \ldots, N_\infty$ for all the k values from zero to infinity is as follows, under the condition that the stream cardinality is n.

$$Pr\{N_0, N_1, \ldots, N_\infty \mid n\} = \frac{m!}{N_0!N_1!\ldots N_\infty!} \prod_{k=0}^{\infty} Pr\{M_j = k\}^{N_k}$$

However, it is impossible to measure the number of registers N_k for an arbitrarily large k value up to infinity, because each register are given limited memory space (typically 5 bits). We use a symbol $\mathcal{K}$ to characterize a register's up-bound capacity of recording k value. For example, if each register is given 5 bits, then it can record a limited range of k values starting from 0 up to $2^5 - 1 = 31$, and in this case, $\mathcal{K} = 32$. If each register is of 4 bits, then the threshold $\mathcal{K} = 2^4 = 16$. Considering this upper limit of recording k values, the above probability function needs to be modified, assuming only the availability of the observations $N_0, N_1, \ldots, N_{\mathcal{K}-1}$.

$$\begin{aligned} &Pr\{N_0, N_1, \ldots, N_{\mathcal{K}-1} \mid n\} \\ &\qquad \approx \frac{m!}{N_0!N_1!\ldots N_{\mathcal{K}-1}!} \prod_{k=0}^{\mathcal{K}-1} Pr\{M_j = k\}^{N_k} \end{aligned}$$

This probability function is also called the *likelihood* of unknown parameter n, when given the observations about the number of registers carrying each value: $N_0, N_1, \ldots, N_{\mathcal{K}-1}$.

$$\begin{aligned} &\mathcal{L}(n \mid N_0, N_1, \ldots, N_{\mathcal{K}-1}) \\ &\qquad \approx \frac{m!}{N_0!N_1!\ldots N_{\mathcal{K}-1}!} \prod_{k=0}^{\mathcal{K}-1} Pr\{M_j = k\}^{N_k} \end{aligned} \tag{4}$$

This likelihood function of cardinality n has to be an approximation, due to the limited recording capacity of HLL registers. This is in fact a shortcoming of HyperLogLog (HLL): A HLL register records the position of the rightmost non-empty bucket as shown in Figure 1. When the cardinality is too large, this position may exceed the capacity of a register and needs to round down to some smaller value. We call it the "overflow" event - the recording value exceeds the capacity of a HLL register. Hence, the largest register value in (4) is $\mathcal{K} - 1$ rather than infinity. But the approximation error of (4) can be safely neglected. This is because we apply the maximum likelihood method only to address the strong bias problem of HyperLogLog between $2m$ and $5m$, so that HyperLogLog will have unbiased estimation in the entire measurement range. When the cardinality is that small, the "overflow" events of HLL registers happen very rarely. The overflow probability will be modelled later by the Eq. (13) as $Pr\{M_j \geq \mathcal{K}\}$.

Applying the well-known *maximum likelihood estimation*, we can find the best n value that maximizes this log-likelihood function, and we use the symbol $\hat{n}$ to denote this optimized estimation of the stream cardinality n.

$$\hat{n} = \arg\max_n \log \mathcal{L}(n \mid N_0, N_1, \ldots, N_{\mathcal{K}-1}) \tag{5}$$

B. Gradient Ascent Solution for MLE

In this subsection, we present our solution to the MLE optimization problem in (5). Although it is viable to solve this problem symbolically by finding the closed-form root to the equation $\frac{\partial}{\partial n}\log\mathcal{L}(n \mid N_0, N_1, \ldots, N_{\mathcal{K}-1}) = 0$, this solution will be complex and have low flexibility (We will demonstrate this point in the next section, when the symbol $\mathcal{K}$ is configured to some other value smaller than 2^5). Therefore, we choose to solve this optimization problem numerically.

When maximizing the log-likelihood function in (4), we guide the iterative search for the best n estimated value based on the gradient direction of log-likelihood function. We use the following iterative optimization procedure to obtain an optimized estimation of stream cardinality n:

$$\hat{n}^{(i+1)} = \hat{n}^{(i)} + \eta \cdot \frac{\partial}{\partial \hat{n}^{(i)}} \log\mathcal{L}(\hat{n}^{(i)} \mid N_0, N_1, \ldots, N_{\mathcal{K}-1}), \quad (6)$$

where $\hat{n}^{(i)}$ is the current cardinality estimation, $\hat{n}^{(i+1)}$ is the next-round estimation, $\mathcal{B}$ is the smallest value among all registers, η is the optimization step size equal to $\alpha_m 2^{\mathcal{B}} m$, and $\frac{\partial}{\partial n}\log\mathcal{L}(n \mid N_0, N_1, \ldots, N_{\mathcal{K}-1})$ is the gradient of log-likelihood function whose expression is in Eq. (7). After tens of optimization steps, we will obtain an unbiased cardinality estimation. But this estimation is a floating number. We need to round it to a closest integer, as our final estimation result.

We analyze the derivative of log-likelihood function over n:

$$\begin{aligned} &\frac{\partial}{\partial n}\log\mathcal{L}(n \mid N_0, N_1, \ldots, N_{\mathcal{K}-1}) \\ &\quad = \frac{\partial}{\partial n}\sum_{k=0}^{\mathcal{K}-1} N_k \log Pr\{M_j = k\} \\ &\quad = \sum_{k=0}^{\mathcal{K}-1} N_k \frac{\frac{\partial}{\partial n} Pr\{M_j = k\}}{Pr\{M_j = k\}}, \end{aligned} \quad (7)$$

where $\frac{\partial}{\partial n} Pr\{M_j = k\}$ for any k value is in given Theorem 2.

Theorem 2 (Derivative of Register Probability): For the probability of a HLL register to carry an arbitrary value k, its first-order derivative is as follows.

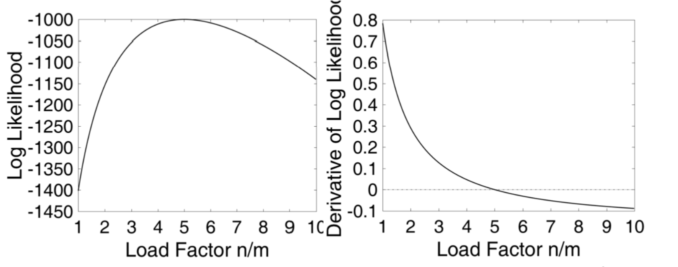
$$\frac{\partial}{\partial n} Pr\{M_j = k\} = \begin{cases} (1 - \frac{1}{m})^n \log(1 - \frac{1}{m}) & \text{if } k = 0 \\ (1 - \frac{1}{m2^k})^n \log(1 - \frac{1}{m2^k}) - \\ (1 - \frac{1}{m2^{k-1}})^n \log(1 - \frac{1}{m2^{k-1}}) & \text{if } k \geq 1 \end{cases} \quad (8)$$

Proof: Directly follow the conclusion in Theorem 1. □

We will prove that our steepest ascent method in (6) always converges to the global optimal solution. Firstly, we analyze the second-order derivative $\frac{\partial^2}{\partial n^2}\log\mathcal{L}$. From (7), we have

$$\begin{aligned} &\frac{\partial^2}{\partial n^2}\log\mathcal{L}(n \mid N_0, N_1, \ldots, N_{\mathcal{K}-1}) \\ &\quad = \sum_{k=0}^{\mathcal{K}-1} N_k \frac{\left(\frac{\partial^2}{\partial n^2} Pr\{M_j = k\}\right) Pr\{M_j = k\} - \left(\frac{\partial}{\partial n} Pr\{M_j = k\}\right)^2}{Pr\{M_j = k\}^2}. \end{aligned} \quad (9)$$

It is easy to very, when $k = 0$, the nominator is equal to $(1 - \frac{1}{m})^n \log^2(1 - \frac{1}{m})(1 - \frac{1}{m})^n - (1 - \frac{1}{m})^{2n}\log^2(1 - \frac{1}{m}) = 0$. When $k \geq 1$, the nominator is equal to $(a^n \log^2\ a - b^n \log^2\ b)(a^n - b^n) - (a^n \log a - b^n \log b)^2 = -(\log a - \log b)^2\ a^n b^n$, where $a = 1 - \frac{1}{m2^k}$, and $b = 1 - \frac{1}{m2^{k-1}}$. As a result, the second-order derivative $\frac{\partial^2}{\partial n^2}\log\mathcal{L}$ is always negative, indicating that the first-order derivative $\frac{\partial}{\partial n}\log\mathcal{L}$ is a monotonic function with respect to the cardinality n. This implies that there is at most one solution that can make $\frac{\partial}{\partial n}\log\mathcal{L}$ equal to zero, and the log-likelihood function will have only one local optima.



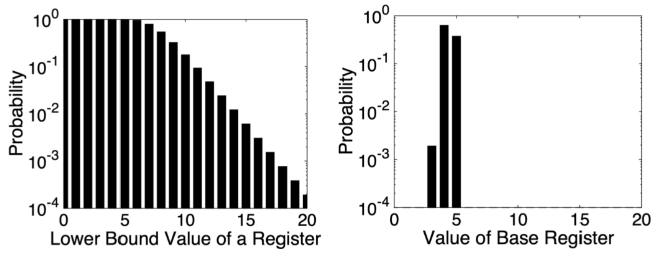
(a) Log-likelihood $\log\mathcal{L}(n \mid \ldots)$ (b) Derivative of log-likelihood $\frac{\partial}{\partial n}\log\mathcal{L}$

Fig. 3. Likelihood of cardinality estimate $\hat{n}$, when $m = 512$ and $n = 5m$.

To illustrate the convergence, we set $m = 512$ and $n = 5m$, and we draw the log-likelihood function $\log\mathcal{L}(n \mid N_0, N_1, \ldots, N_{\mathcal{K}-1})$ and its derivative $\frac{\partial}{\partial n}\log\mathcal{L}$ in Figure 3. For simplicity, we suppose each observation N_k, $0 \leq k < 32$, is equal to its expected value $m \cdot Pr\{M_j = k\}$ as defined in (3). The plot (a) shows that $\log\mathcal{L}$ has only one local maximum. Our steepest ascent method will converge to this global optima. The plot (b) shows that the derivative of $\log\mathcal{L}$ is monotonic, and the equation $\frac{\partial}{\partial n}\log\mathcal{L} = 0$ has only one solution $n = 5m$.

To speed up the convergence rate, we can obtain a good initial guess of the cardinality n by the closed-form formula in (2). Recall that this cardinality estimation is biased when it is between $2m$ and $5m$, and our key purpose is to mitigate this bias problem of HyperLogLog using the steepest ascent method. Hence, we use the gradient direction in (7) to fine tune the cardinality guess to an optimized position with no bias. Since we apply the MLE method only in the small region $n \in (2m, 5m)$, we can neglect its shortcoming of higher computational cost than the closed-form formula (2).

In (6), we set the optimization step size as $\eta = \alpha_m 2^{\mathcal{B}} m$ for the following reason. On the one hand, the step size should not be too small, which would otherwise cause the convergence to be slow. Since each register satisfies the inequality $M_j \geq \mathcal{B}$, we know $\alpha_m 2^{\mathcal{B}} m$ is proportional to HyperLogLog's cardinality estimation $\hat{n}$, according to its estimation formula in (2). Such an adaptive step size prevents slow convergence for large cardinalities. On the other hand, the step size should not be too big, which would otherwise degrade accuracy. We know that the gradient at the optimal point is zero. When our cardinality guess moves close to the optima, the amplitude of log-likelihood gradient $\frac{\partial}{\partial n}\log\mathcal{L}(n \mid N_0, N_1, \ldots, N_{\mathcal{K}-1})$ approaches to zero. For example, it is smaller than 0.1 in our experiment of Fig. 3(b). We also know that the step size $\eta = \alpha_m 2^{\mathcal{B}} m$ is at least four times smaller than the cardinality estimation $\hat{n}$, since the base $\mathcal{B}$, whose probability distribution is shown in Fig. 4(b), is no larger than the peak minus two,



(a) Probability of register lower bound. (b) Probability of base register $\mathcal{B}$.

Fig. 4. When $m = 512$ and $n/m = 100$, the probability for a HLL register to take a value no less than a bound k, and the probability of base register.

as shown by the register value distribution in Fig. 2(a). Then, the multiplication result of the gradient and the step size will be small as we step towards the maxima; for instance, it is no more than 1/40 of the cardinality under estimation in our experiment, which can help converge to an accurate solution.

V. HLL-TailCut Algorithm

In this section, we reduce the size of each HLL register from five bits to four bits. We call the new algorithm HLL-TailCut, which applies the long tail cutoff technique to the register value distribution in Fig. 2. Compared with HLL, HLL-TailCut has two advantages: It is 20% more memory efficient, and it supports the counting of Tera- or Peta-scale cardinalities.

A. Base Register and Offset Registers

Our basic idea is to use a shared base register for storing the smallest value among the set of HyperLogLog registers, so that the m registers only need to store their offsets relative to the base register. Intuitively, the offset stays in a much smaller range than 2^5 (see Fig. 2) and can be encoded by less than five bits. In following, we explain how to maintain the base register and the m offset registers, upon the arrival of stream elements.

Let $\mathcal{B}$ be a base register that records the smallest value of the HLL register set. Hence, we have $\mathcal{B} \geq 0$.

$$\mathcal{B} = \min_{0 \leq j < m} M_j$$

With this base $\mathcal{B}$, each register $\tilde{M}_j$ is now called an offset register that records only its offset relative to $\mathcal{B}$. When a stream element arrives, we need to memorize its occurrence by updating the offset register $\tilde{M}_j$. To serve this purpose, we modify the register updating Eq. (1) as

$$\tilde{M}_j := \max\left(\tilde{M}_j,\ \rho(x') - \mathcal{B}\right), \tag{10}$$

where $\tilde{M}_j$ is the jth offset register to which an arrival element is mapped, $\rho(x')$ is the index of the bucket chosen by the element, and $\rho(x') - \mathcal{B}$ is the offset of the bucket index from $\mathcal{B}$.

Handle Overflow of Offset Register: We define each offset register to be four bits long. Thus, an offset register's recording capacity $\mathcal{K}$ is $2^4 = 16$, implying that the recorded offset value must be smaller than $\mathcal{K} = 16$. However, occasionally, the offset values $\rho(x') - \mathcal{B}$ of some stream elements are at least $\mathcal{K}$. We use the term "overflow" to refer to the attempts of updating the offset register $\tilde{M}_j$ to the $\mathcal{K}$ value or above.

In order to address the overflow event, we scan the m offset registers to find the smallest offset value, which is denoted as $\Delta\mathcal{B}$. If $\Delta\mathcal{B}$ is non-zero, it tells that the shared base register $\mathcal{B}$ can be increased by this amount to reduce the offset value stored in each offset register. We call this operation "base register update": As $\mathcal{B}$ increases by $\Delta\mathcal{B}$, each offset register $\tilde{M}_j$ must be decreased by $\Delta\mathcal{B}$, since they record offsets to $\mathcal{B}$.

Thanks to this base updating operation, we have a base register $\mathcal{B}$ that increases dynamically as more stream element arrives. Essentially, our purpose is to use a set of truncated registers $\mathcal{B} + \tilde{M}_j$ to record the cardinality information. Although each offset register $\tilde{M}_j$ with a limited number of bits must be smaller than $\mathcal{K}$, the base register $\mathcal{B}$ can grow arbitrarily large. Therefore, we can count very large cardinalities on Tera or Peta scale without giving more bits to offset registers.

After the increase of the base register by $\Delta\mathcal{B}$, the new offset value $\rho(x') - \mathcal{B}$ in (10) may become smaller than $\mathcal{K}$. If it is true, the overflow event disappears. Otherwise, the overflow problem can not be resolved, and the jth offset register has to be truncated by the cutoff bound $\mathcal{K}$ as follows. Later in Section V-C, we will theoretically analyze the probability of unresolvable overflows.

$$\tilde{M}_j := \max\left(\tilde{M}_j,\ \min(\rho(x') - \mathcal{B},\ \ \mathcal{K} - 1)\right) \tag{11}$$

For a fraction of stream elements, their $\rho(x')$ may be smaller than the current value of the base register $\mathcal{B}$. In this case, the new offset variable $\rho(x') - \mathcal{B}$ will be negative, and Eq. (11) will leave the offset register $\tilde{M}_j$ unchanged. Our handling of this case is consistent with the basic idea of HyperLogLog: Each HLL register M_j maintains the position of the rightmost non-empty bucket in Fig. 1, so that each HLL register can give an independent cardinality estimation 2^{M_j}. Our HLL-TC is to approximate a HLL register M_j by a truncated register $\mathcal{B} + \tilde{M}_j$, where the base $\mathcal{B}$ keeps track of the smallest value among all HLL registers. When an update $\rho(x')$ is smaller than $\mathcal{B}$, it is of course smaller than any HLL register. Eq. (1), which updates each HLL register M_j upon the arrival of stream elements, shows that each HLL register will be unchanged. Hence, the base $\mathcal{B}$ that tracks the minimum of all HLL registers will stay the same, and so does the offset $\tilde{M}_j$. In summary, we can safely ignore the case $\rho(x') \leq \mathcal{B}$. This won't affect the cardinality estimation result.

B. Pseudocode of HLL-TailCut

In this subsection, we describe the procedure of the HLL-TailCut algorithm (abbreviated as HLL-TC), which can be divided into two parts: an online component that updates the base register $\mathcal{B}$ and offset registers $\tilde{M}_j$, $0 \leq j < m$, upon the arrival of stream elements, and an offline component that estimates the stream cardinality n using these registers.

We present the pseudocode of the online component in Algorithm 1. We use the term "truncated register" to refer to the sum of base register $\mathcal{B}$ and offset register $\tilde{M}_j$. The Algorithm 1 essentially maintains a set of truncated registers

Algorithm 1 Online Component of HLL-TailCut

```
1  initialize $\mathcal{B}$ and $\tilde{M}_j$ to zero, for each $j \in [0, m)$
2  foreach element $e$ in data stream $\mathcal{S}$ do
3      $x := h(e), \quad j := \langle x_1\ x_2 \cdots x_b \rangle, \quad x' := \langle x_{b+1} x_{b+2} \cdots \rangle$
4      if $\rho(x') - \mathcal{B} \geq \mathcal{K}$ then        // detect overflow
5          $\Delta\mathcal{B} := \min_{0 \leq j < m} \tilde{M}_j$
6          if $\Delta\mathcal{B} > 0$ then
7              $\mathcal{B} := \mathcal{B} + \Delta\mathcal{B}$   // update base register
8              foreach $j \in [0, m)$ do $\tilde{M}_j := \tilde{M}_j - \Delta\mathcal{B}$
9
10     $\tilde{M}_j := \max\big(\tilde{M}_j,\ \min(\rho(x') - \mathcal{B},\ \mathcal{K} - 1)\big)$
```

$\mathcal{B} + \tilde{M}_j$, $0 \leq j < m$. Different from the HLL register M_j maintained by (1), these truncated registers chop off the long tail of a histogram (like in Fig. 2) by the bound $\mathcal{B} + \mathcal{K}$, and the chopped part is stacked above the $(\mathcal{B}+\mathcal{K}-1)$th bar, due to the line 10. Thus, the resultant histogram will exhibit an edge peak distribution with a spike close to the tail truncation point.

The computational complexity of the online component of our HLL-TC does not increase when the cardinality to estimate is very large. The most expensive operation of Algorithm 1 is the lines 5∼9, which scan the m offset registers to find the smallest offset value and update $\mathcal{B}$. However, we perform this operation only when an arrival element triggers the overflow event at line 4. Its probability is $2^{\mathcal{B}+\mathcal{K}-1}$, which reduces exponentially as the increase of base register $\mathcal{B}$. Therefore, the computational cost of Algorithm 1 actually reduces when the cardinality of the data stream grows very large.

Since the Algorithm 1 no longer maintains the HLL register M_j, we need to modify the offline estimation equation in (2), using the newly designed base register $\mathcal{B}$ and offset registers $\tilde{M}_j$. A straightforward solution is to replace M_j by the jth truncated register $\mathcal{B} + \tilde{M}_j$.

$$\hat{n} = \alpha_m \cdot m^2 \cdot \left(\textstyle\sum_{0 \leq j < m} 2^{-(\mathcal{B}+\tilde{M}_j)}\right)^{-1} \tag{12}$$

In this formula, we use the harmonic averaging to aggregate the cardinality estimation results $2^{\mathcal{B}+\tilde{M}_j}$ of different registers $\mathcal{B}+\tilde{M}_j$, $0 \leq j < m$. The harmonic mean technique is inherited from HyperLogLog [8], which is to mitigate the impact of large outliers in a register value distribution (see the right-side long tail of Fig. 2(b)). Therefore, our HLL-TC algorithm can provide a reliable cardinality estimation when there are a group of outliers far away from the peak.

If the cardinality estimate $\hat{n}$ by (12) is between $2m$ and $5m$, we will use the MLE estimator in (5) to fine tune the estimated result and make it unbiased. If $\hat{n}$ by (12) is smaller than $2m$, LinearCounting is more accurate: $\hat{n} = -m\log(z/m)$, where z is the number of registers $\mathcal{B} + \tilde{M}_j$ that are equal to zeros.

C. Theoretical Analysis of HLL-TC

This subsection will prove our HLL-TC in (12) can produce unbiased cardinality estimations. An intuitive explanation is, when each offset register is given four-bits memory and the cutoff bound $\mathcal{K}$ is 16, the truncated long tail in the register value distribution has negligibly small probability mass.

Previously, we have given in (3) the probability for a HLL register to carry an arbitrary k value. In the following theorem, we present $Pr\{M_j \geq k\}$, the probability for a register to carry a value of at least k, which is called the *tail probability*.

Theorem 3 (Probability of Smallest Register): The probability for a register to carry a value of at least k is

$$Pr\{M_j \geq k\} = \begin{cases} 1 & \text{if } k = 0 \\ 1 - (1 - \dfrac{1}{m2^{k-1}})^n & \text{if } k \geq 1, \end{cases} \tag{13}$$

Proof: Directly derived from Theorem 1. □

From the tail probability in (13), we can analyze the value distribution of the base register $\mathcal{B}$:

$$Pr\{\mathcal{B} = b\} = Pr\{\mathcal{B} \geq b\} - Pr\{\mathcal{B} \geq b+1\}, \text{ where} \tag{14}$$
$$Pr\{\mathcal{B} \geq b\} = Pr\{\forall j,\ M_j \geq b\} = \textstyle\prod_{0 \leq j < m} Pr\{M_j \geq b\}$$
$$= \begin{cases} 1 & \text{if } b = 0, \\ \left(1 - (1 - \frac{1}{m2^{b-1}})^n\right)^m & \text{if } b \geq 1. \end{cases}$$

Suppose the number of registers m is 512 and the load factor $n/m = 100$. Under this parameter setting, Fig. 4(a) plots the tail distribution of HLL register values in (13), and Fig. 4(b) plots the value distribution of base register in (14). We have two findings about Figure 4.

Firstly, that the base register $\mathcal{B}$ is quite stable, and often alternates between two neighboring values. As shown in Fig. 4(b), in most cases, $\mathcal{B}$ takes one of the two values, i.e., 4 and 5 when $m = 512$ and $\frac{n}{m} = 100$. The conclusion is similar when we examine other settings of the number of registers m and the load factor $\frac{n}{m}$.

Secondly, in Fig. 4(a), the tail probability $Pr\{M_j \geq k\}$ reduces exponentially as the k value grows. We define the "overflow" probability as the chance for a register to take a value at least $\mathcal{B} + \mathcal{K}$, after an estimator with m registers accepts a data stream with cardinality n. An offset register with $\lceil \log_2(\mathcal{K}) \rceil$ bits can only record a value smaller than $\mathcal{K}$ and has to be truncated. Suppose each offset register is given four bits. Then, the tail cutoff bound is $\mathcal{B}+16$, and the overflow probability is $Pr\{M_j \geq \mathcal{B} + 16\}$. In Fig. 4(a), if the base $\mathcal{B}$ is 4, the overflow probability $Pr\{M_j \geq 4+16\} = Pr\{M_j \geq 20\}$ is as small as 0.02%. Since our HLL-TC assumes four bits per register, the overflow probability is negligibly small, and a truncated register $\mathcal{B} + \tilde{M}_j$ can closely approximate a HLL register M_j with no obvious information loss. As a result, we can use the closed-form equation in (12) similar to HyperLogLog to estimate the stream cardinality.

HLL-TC has two advantages compared with HyperLogLog: Memory cost can be reduced by 20% under the same accuracy constraint, and Tera/Peta-scale large cardinalities can be accurately estimated. HLL-TC inherits a good attribute called "composable" from HyperLogLog: Suppose multiple HLL-TC estimators are deployed at distributed sites. For each estimator, its base register $\mathcal{B}$ and offset registers $\tilde{M}_j$ can be transmitted to a central server, which has abundant resources to recover the set of HyperLogLog registers $\mathcal{B} + \tilde{M}_j$ for each estimator. Then, the server can merge the multiple estimators

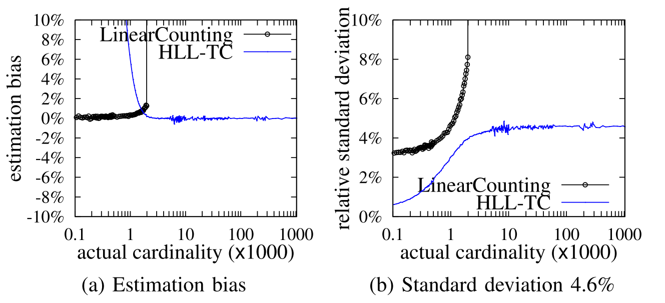


Fig. 5. Performance of HLL-TailCut allocated with 512 offset registers, each of which is given four bits memory.

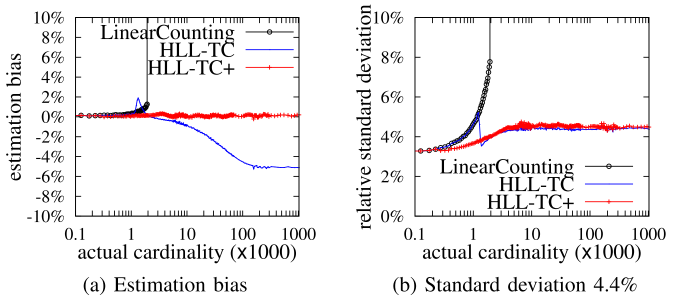


Fig. 6. Performance of HLL-TailCut configured with 512 offset registers, each of which is given three bits memory.

to determine the union cardinality of the data streams at distributed sites [8].

However, when each offset register is given only three bits, the overflow probability $Pr\{M_j \geq 4+8\} = Pr\{M_j \geq 12\}$ grows to 4.6%. The high overflow rate will cause estimation bias problem in Fig. 6(a), which is compensated by HLL-TC+ in Section VI using maximum likelihood estimation.

D. Accuracy Evaluation of HLL-TailCut

We empirically evaluate how the estimated result by (12) is affected, when we truncate the right-side tail of a HLL histogram by the bound $\mathcal{B}+\mathcal{K}$ when $\mathcal{K}=16$. Our evaluation results in Fig. 5 verify that the tail cutoff across $\mathcal{B}+16$ causes negligibly small bias to the cardinality estimation by (12).

Subfigure (a) illustrates the estimation bias $E(\hat{n}-n)/n$, where n is the actual cardinality and $\hat{n}$ is the estimated value. Subfigure (b) depicts the relative standard deviation of estimated results $\sqrt{Var(\hat{n})}/E(\hat{n})$. We illustrate both the results of LinearCounting and HLL-TC, which are configured with the same number of memory units: LinearCounting is given $m=512$ bits, and HLL-TC is given $m=512$ offset registers. Plot (a) shows that HLL-TC in (12) can produce unbiased estimations, when the cardinality n is larger than $5m=2560$. It also shows LinearCounting generates unbiased estimations for cardinalities smaller than 2 $m=1024$. Hence, the two formulas are complementary with each other. Later in simulation section, we will illustrate why it is better to use the MLE estimator in (5) when $2m<n<5m$. It is more obvious when the number of register m is a few thousands.

The expected relative error of HLL-TC is the same with the error of HyperLogLog [8]: $\frac{1.04}{\sqrt{m}}$, where m is the number of offset registers. The expected relative error is mainly determined by the number of offset registers m that can be created from the allocated memory. It has no strong relation with the stream cardinality n to estimate. For example, suppose the memory cost of HLL-TC is 257 bytes and the base register occupies just one byte. Then, 512 offset registers can be created, and the relative estimation error is expected to be $\frac{1.04}{\sqrt{512}} \approx 4.6\%$. This is consistent with the experimental result in Fig. 5(b). When $m=512$, the standard deviation of HLL-TC is 4.6%, but it converges to 4.6% only when the cardinality n is larger than $10m=5120$. When the number of offset registers m increases to 4096, the memory cost of HLL-TC is about 2 kilo bytes, and the relative error is $\frac{1.04}{\sqrt{4096}} \approx 1.6\%$.

VI. HLL-TailCut+ Algorithm

In this section, we reduce the size of each offset register to three bits, and save the memory cost by over 40% than HLL.

A. Bias Problem of the Naive HLL-TailCut

When the offset register size reduces to three bits, we assign the cutoff bound $\mathcal{K}$ to $2^3=8$, and then we can reuse the online component in Algorithm 1 to maintain the base register $\mathcal{B}$ and each offset register $\tilde{M}_j$, upon the arrival of stream elements. However, the offline analysis component in (12) used by the naive HLL-TailCut has a serious "estimation bias" problem, which will be identified and explained as follows.

HLL-TC adopts an estimation equation in (12) similar to HyperLogLog. For this solution, we illustrate its experimental results in Fig. 6. The subfigure (a) shows that HLL-TC with cutoff bound $\mathcal{K}=8$ produces the estimation bias of -5.2%. This is because, when the offset register is three bits and $\mathcal{K}$ reduces to eight, the percentage of registers truncated by (11), called overflow probability, will greatly increases to about 5%. Thus, a non-negligible fraction of offset registers are truncated.

To make things worse, in Fig. 6(a), the bias of HLL-TC exaggerates to -5.2% by a non-linear curve, implying that we cannot compensate such bias simply by applying a constant corrector to the biased estimation result.

Interestingly, the standard deviation of HLL-TC decreases from 4.6% shown in Fig. 5(b) to 4.4% shown in Fig. 6(b). This is because more outliers in the long tail are discarded, as the tail cutoff bound changes from 16 to 8. More aggressive outlier rejection brings a small degree of accuracy gain.

B. Probabilistic Model of Truncated Register

To address the negative bias problem of HLL-TC, we will propose a HLL-TailCut+ algorithm, which modifies the MLE-based HyperLogLog algorithm discussed in Section IV. We have already depicted its performance in Fig. 6. Plot (a) shows that it can provide unbiased estimations in the entire range, and plot (b) shows that it has comparable accuracy with HLL-TC.

Before introducing this new algorithm, in this subsection, we analyze the probability of a truncated register $\mathcal{B}+\tilde{M}_j$

to exhibit an arbitrary k value. The key difficulty for this analysis is that, during the execution time of the online component, the shared register $\mathcal{B}$ is not a fixed value, but gradually increases as the register set receives more and more stream elements.

We begin by defining several notations. Let b be the final value which the base register $\mathcal{B}$ has been updated to. Note that b is typically a small value even for a very large stream at the scale of 10^9. For example, let the number of registers m be 1024. When the stream size n equals $10^2\ m$, the b value alternates between 4 and 5, as in Fig. 4(a). When n increases to $10^6\ m$ (about 10^9), the b value grows only to 17 or 18.

As the base register $\mathcal{B}$ undergoes the step-by-step increase in the range of $[0, b]$, the register set consisting of m offset registers will receive different numbers of stream elements.

- When $\mathcal{B}$ is equal to 0, we assume that the register set receives n_0 distinct stream elements.
- When $\mathcal{B}$ is equal to 1, the register set receives n_1 stream elements that are distinct from the previous n_0 elements.
- …
- When $\mathcal{B}$ is equal to b, the register set receives n_b stream elements that are distinct from the elements $n_0, n_1, \ldots, n_{b-1}$ received when $\mathcal{B}$ is equal to the previous values.

The purpose of our problem is to estimate the total cardinality n of the data stream, which is equal to $n_0 + n_1 + \ldots + n_b$.

Let $\tilde{M}_j^{(0)}, \tilde{M}_j^{(1)}, \ldots, \tilde{M}_j^{(b)}$ be the values of the jth offset register, when the base register $\mathcal{B}$ is fixed to $0, 1, \ldots, b$ and the register set *independently* receives $n_0, n_1, \ldots, n_b$ distinct elements, respectively. For example, $\tilde{M}_j^{(1)}$ is the value of jth offset register, when the base register $\mathcal{B}$ is fixed to 1 and the register set receives n_1 unique elements that are totally different from the n_0 elements received when $\mathcal{B}$ is still zero.

After the register set receives all the $n = n_0 + n_1 + \ldots + n_b$ stream elements, the jth truncated register $\mathcal{B} + \tilde{M}_j$ becomes

$$\mathcal{B} + \tilde{M}_j = \max(\mathcal{B} + \tilde{M}_j^{(0)},\ \mathcal{B} + \tilde{M}_j^{(1)},\ \ldots,\ \mathcal{B} + \tilde{M}_j^{(b)}).$$

Because $\mathcal{B} + \tilde{M}_j^{(0)}, \mathcal{B} + \tilde{M}_j^{(1)}, \ldots, \mathcal{B} + \tilde{M}_j^{(b)}$ are independent, the cumulative probability for $\mathcal{B} + \tilde{M}_j$ (i.e., the probability for the jth truncated register to carry a value of at most k) is

$$\begin{aligned} &Pr\{\mathcal{B} + \tilde{M}_j \le k \mid n_0, n_1, \ldots, n_b\} \\ &\qquad = \textstyle\prod_{0 \le i \le b} Pr\{\mathcal{B} + \tilde{M}_j^{(i)} \le k \mid n_i\}. \end{aligned} \tag{15}$$

Here, it needs the cumulative distributions of the jth truncated register $Pr\{\mathcal{B} + \tilde{M}_j^{(i)} \le k \mid n_i\}$, when the base register is fixed to a value i ranging from 0 to b. If the base register is equal to b value, the cumulative distributions $Pr\{\mathcal{B} + \tilde{M}_j^{(b)} \le k \mid n_b\}$ is given in the following theorem. When the base register is equal to other values 0, 1, …, or $b - 1$, we can easily obtain their corresponding cumulative probability, if we replace the symbol b in (16) by 0, 1, …, or $b - 1$, respectively.

Property 1 (Cumulative Distribution of Truncated Register $\mathcal{B} + \tilde{M}_j^{(b)}$ With Fixed Base Register): When the base register $\mathcal{B}$ is fixed to a value b and the register set receives n_b distinct elements, the probability for the truncated register $\mathcal{B} + \tilde{M}_j^{(b)}$ to exhibit a value of no more than k is as follows.

$$\begin{aligned} &Pr\{\mathcal{B} + \tilde{M}_j^{(b)} \le k \mid n_b\} \\ &\qquad = \begin{cases} \left(1 - \dfrac{1}{m2^k}\right)^{n_b} & \textit{if } 0 \le k \le b + \mathcal{K} - 2 \\ 1 & \textit{if } k \ge b + \mathcal{K} - 1 \end{cases} \end{aligned} \tag{16}$$

Proof: Directly derived from Theorem 1. □

By applying (16) to (15), we can obtain the cumulative probability of the jth truncated register $Pr\{\mathcal{B} + \tilde{M}_j \le k \mid n_0, n_1, \ldots, n_b\}$. We refrain from expanding this formula, which otherwise will become too complicated. Then, with the cumulative probability in (15), we can derive the probability density function for the jth truncated register $\mathcal{B} + \tilde{M}_j$.

$$\begin{aligned} &Pr\{\mathcal{B} + \tilde{M}_j = k \mid n_0, n_1, \ldots, n_b\} \\ &\quad = \begin{cases} 0 & \text{if } k < b \\ Pr\{\mathcal{B} + \tilde{M}_j \le k \mid n_0, n_1, \ldots, n_b\} & \text{if } k = b \\ Pr\{\mathcal{B} + \tilde{M}_j \le k \mid n_0, n_1, \ldots, n_b\} - \\ Pr\{\mathcal{B} + \tilde{M}_j \le k{-}1 \mid n_0, n_1, \ldots, n_b\} & \text{if } k > b \end{cases} \end{aligned} \tag{17}$$

Here, the probability for the truncated register to take a value less than b is zero, because the base register $\mathcal{B}$ increases to b after receiving all the n elements, which makes it impossible for the truncated register $\mathcal{B} + \tilde{M}_j$ to be smaller than b.

C. Maximum Likelihood Estimator

As the probability for a truncated register $\mathcal{B} + \tilde{M}_j$ to carry an arbitrary k value is available in (17), the only problem that remains is how we use this parameterized probabilistic model with b unknown variables $n_0, n_1, \ldots, n_b$, to generate an unbiased estimation of the total stream cardinality n.

We address the problem by estimating the b unknown parameters one by one. When the base register $\mathcal{B}$ is about to increase from zero to one, we estimate n_0, the number of distinct elements received. To accomplish this task, since the base $\mathcal{B}$ is still zero, we can use directly the maximum likelihood estimator in Section IV. Note that when the estimation of n_0 is smaller than m, we will use instead the estimated result by LinearCounting [19] for better accuracy, as inspired by the work [16] that argues LinearCounting is more accurate than HyperLogLog if given enough memory space.

Then, following the principle of mathematical induction, we assume that the stream cardinalities $n_0, n_1, \ldots, n_{b-1}$ all have been estimated as $\hat{n_0}, \hat{n_1}, \ldots, \hat{n_{b-1}}$, at the time that the base register $\mathcal{B}$ is about to update to $1, 2, \ldots, b$, respectively. Based on them, we will further estimate the next unknown variable n_b. The likelihood function of n_b is as follows.

$$\begin{aligned} &\mathcal{L}(n_b \mid N_0, N_1, \ldots, N_{b+\mathcal{K}-1}) \\ &\quad = \tfrac{m!}{N_0! N_1! \ldots N_{b+\mathcal{K}-1}!} \\ &\qquad \cdot \textstyle\prod_{k=b}^{b+\mathcal{K}-1} Pr\{\mathcal{B} + \tilde{M}_j = k \mid \hat{n_0}, \hat{n_1}, \ldots, \hat{n_{b-1}}, n_b\}^{N_k} \end{aligned} \tag{18}$$

TABLE II
APPLY HLL-TC+ TO DATA STREAMS WITH 16×10^9 DISTINCT ELEMENTS

Register Number (m)	**Avg Bias**	**Std Deviation**	**Error Eqn**
1024	-0.02%	3.13%	$1.00/\sqrt{m}$
4096	-0.06%	1.56%	$1.00/\sqrt{m}$
8192	-0.01%	1.11%	$1.00/\sqrt{m}$

The probability density function $Pr\{\mathcal{B}+\tilde{M}_j = k \mid n_0, n_1, \ldots, n_b\}$ of truncated register $\mathcal{B}+\tilde{M}_j$ is in (17). We replace the true values of $n_0, n_1, \ldots, n_{b-1}$ by their estimated values in (18).

The likelihood function in (18) has a complicated mathematical expression after expansion. It is difficult to find a closed-form solution that can maximize this likelihood in (18). Hence, we numerically search for the optimal model parameter $\hat{n_b}$ using the derivative of the log-likelihood function as a heuristic. Let $\hat{n_b}$ be an optimized estimation that can maximize the log-likelihood function. Then, we have

$$\hat{n_b} = \arg\max_{n_b} \log \mathcal{L}(n_b \mid N_0, N_1, \ldots, N_{b+\mathcal{K}-1}). \qquad (19)$$

We solve this maximum log-likelihood problem by a steepest ascent method. Its implementation details is in Appendix VIII.

Because the stream cardinalities $n_0, n_1, \ldots, n_b$ all have been estimated, we can obtain an estimation of the total stream cardinality as $\hat{n} = \hat{n_0} + \hat{n_1} + \ldots + \hat{n_b}$. We call this algorithm HLL-TailCut+ (abbreviated as HLL-TC+). Unlike HLL-TC, this algorithm has no bias problem as illustrated in Fig. 6(a).

As to the convergence of the steepest ascent method, we can prove the log-likelihood function in (19) has only one local maxima, by proving the second-order derivative of the log-likelihood function is always negative. By (18), we have

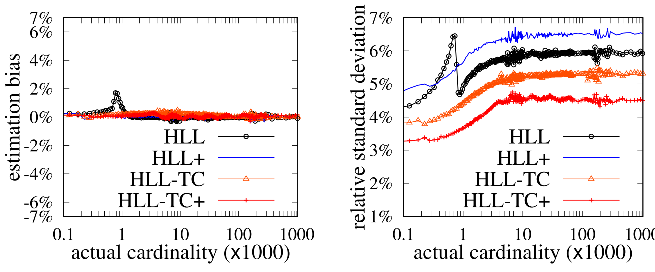
$$\frac{\partial^2}{\partial {n_b}^2} \log \mathcal{L}(n_b \mid N_0, N_1, \ldots, N_{b+\mathcal{K}-1}) = \sum_{k=b}^{b+\mathcal{K}-1} N_k \frac{\left(\frac{\partial^2}{\partial {n_b}^2} Pr\{\mathcal{B}+\tilde{M}_j=k\}\right) Pr\{\mathcal{B}+\tilde{M}_j=k\} - \left(\frac{\partial}{\partial n_b} Pr\{\mathcal{B}+\tilde{M}_j=k\}\right)^2}{Pr\{\mathcal{B}+\tilde{M}_j=k\}^2},$$

where $Pr\{\mathcal{B}+\tilde{M}_j = k\}$ is an abbreviation for $Pr\{\mathcal{B}+\tilde{M}_j = k \mid \hat{n_0}, \hat{n_1}, \ldots, \hat{n_{b-1}}, n_b\}$, which is defined in (17). It is not difficult to verify that the nominator of each term is negative. The proof is similar to the convergence proof in Section IV-B. The key is to treat each $Pr\{\mathcal{B}+\tilde{M}_j^{(i)} \le k \mid n_i\}, i < b$, in (15) as a fixed value. Due to page limit, we omit the detailed steps.

D. Analysis of Memory Cost

The memory cost of HLL-TC+ is the number of offset registers m multiplied by three bits, and its relative standard error is roughly $\frac{1.0}{\sqrt{m}}$. We obtain this relative error $\frac{1.0}{\sqrt{m}}$ by applying HLL-TC+ to a fixed cardinality (e.g., ten million) for ten thousand times, and then calculating the relative standard deviation of estimated results. Later in Table II, we will use more extensive experiments to verify this relative error equation also applies for other m values and other n values.

Since the standard error of HLL-TC+ is $\frac{1.0}{\sqrt{m}}$ and that of HLL is $\frac{1.04}{\sqrt{m}}$, we can show HLL-TC+ only needs 55% memory of HLL to attain the same accuracy. Let $m_{\text{HLL-TC+}}$ (or m_{HLL}) be the number of registers used by HLL-TC+ (or HLL). Then, we have $\frac{1.0}{\sqrt{m_{\text{HLL-TC+}}}} = \frac{1.04}{\sqrt{m_{\text{HLL}}}}$, to attain the same accuracy. Since the register size of HLL is five bits and that of HLL-TC+ is only three bits, the memory cost of HLL-TC+ divided by that of HLL is $\frac{\text{Memory}_{\text{HLL-TC+}}}{\text{Memory}_{\text{HLL}}} = \frac{3 \text{ bits} \cdot m_{\text{HLL-TC+}}}{5 \text{ bits} \cdot m_{\text{HLL}}} \approx \frac{3}{5} \cdot \left(\frac{1.0}{1.04}\right)^2 \approx 55\%$.



(a) Estimation bias (b) Standard deviation

Fig. 7. Compare cardinality estimators with the same $1.54\ k$ bits memory.

VII. EXPERIMENTS

In this section, we evaluate the performance of our proposed HLL-TC and HLL-TC+ algorithms, and compare them with state-of-the-art algorithms, including HyperLogLog (HLL) [8] and HyperLogLog+ (HLL+) [10]. Note that we have shared online the source code of all these four algorithms [15].

A. Experiment Setup

For each cardinality estimator, we will evaluate two performance metrics: the average estimation bias and the average estimation error when given a same amount of memory. We will evaluate the performance of the cardinality estimators under three different scenarios. First, we assume very limited memory budget, no more than a few hundreds bytes per stream, to support cardinality measurements with coarse accuracy ranging from 4% to 10%. Second, we assume the available memory is several kilobytes per stream, which enables highly accurate estimations with the expected errors lower than 2% or even 1%. Third, we would like to verify whether our HLL-TailCut+ estimator can support the measurement of extra large streams whose cardinalities exceed 4×10^9. This bound is important since a five-bit HLL register can only count cardinalities up to $2^{2^5} \approx 4 \times 10^9$.

B. Coarse-Accuracy Estimation

We consider the coarse accuracy $\sigma \approx 4.4\%$. Then, the number of registers m should be $(1.0/0.044)^2 \approx 512$ for HLL-TC+, occupying $512 \times 3 = 1.54\ k$ bits memory. We give the same amount of memory to the other three algorithms, and depict their performance in Fig. 7.

Fig. 7(a) shows that all four algorithms are approximately unbiased. Fig. 7(b) shows that the estimation error of HLL is slightly smaller than the error of HLL+. This is because HLL defines the register size to be five bits to support the counting of data on Giga scale, while HLL+ enlarges the register size to six bits, to extend the operating range to Tera or Peta scale. Hence, when given the same memory budget, HLL+ can allocate a smaller number of registers than HLL. Fig. 7(b)

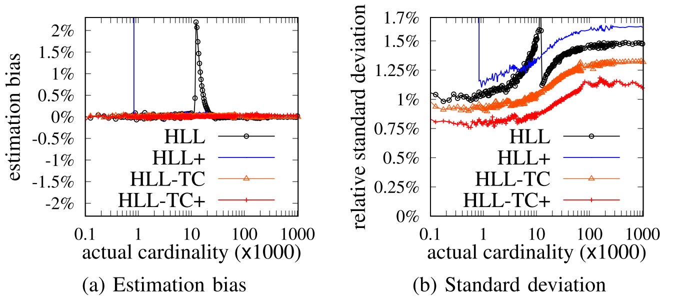


Fig. 8. Compare cardinality estimators with the same $24.58k$ bits memory.

also shows that our HLL-TC and HLL-TC+ algorithms can provide smaller estimation error. This is because HLL-TC and HLL-TC+ have compressed the register size to four bits and three bits, respectively. Given the same amount of memory, they can allocate more registers to achieve higher accuracy.

C. Fine-Accuracy Estimation

We consider the estimation error $\sigma \approx 1.1\%$. To achieve such fine accuracy, HLL-TC+ needs about $(1.0/0.011)^2 \approx 8192$ registers, which occupies $24.58k$ bits memory. We give the same amount of memory to the other three algorithms, and evaluate their performance. Fig. 8(b) shows that HLL-TC+ provide the best accuracy among the four algorithms, and its expected error is $1.0/\sqrt{m} \approx 1.1\%$.

Fig. 8(a) shows that HLL has a high spike that is strongly biased. This is because, in the small region around $2.5m = 2.5 \cdot 8192 \cdot \frac{3 \text{ bits}}{5 \text{ bits}} \approx 12288$, HLL makes a switch between LinearCounting and its raw estimation equation in (2). This bias problem has also been elaborated by previous work [10]. As shown in Fig. 8(a), this bias problem has been solved by HLL+, HLL-TC and HLL-TC+, however using different methods. HLL+ corrects the bias in a brute-force way [10]: It empirically calculates the bias of 200 hundred reference values in the small region from $2m$ to $5m$, and then interpolates between the 200 reference points to determine the correction to apply for any given raw estimation value by (2). In contrast, our HLL-TC addresses this problem elegantly, by substituting the Eq. (2) with a MLE estimator in (5), within the small region from $2m$ to $5m$. Our HLL-TC+ also does not have the bias problem, because it uses the MLE estimator in (19).

D. Extra Large Measurement Range

The previous experiments only show the evaluation results for cardinalities up to one million. In following, we will verify that our HLL-TC+ can measure extra large streams that have over four billions distinct elements. Unlike HyperLogLog+ which increases the register size to six bits to support such large streams, we only need an array of three-bits offset registers (whose number is m) plus a single base register which is at least six bits long.

We list in Table II the average estimation bias and error of our HLL-TC+ algorithm, when it is given different numbers of registers m, such as 2^{10}, 2^{12} and 2^{13}. We only show the experimental results of a single stream cardinality value 16×10^9, since it takes days to process such a large data stream for ten thousands times. In this table, the second column lists the average estimation bias of HLL-TC+, which is negligibly small as compared with its standard deviation shown in the third column. This implies that our algorithm can unbiasedly estimate extra large streams beyond the bound of four billions.

The last column of Table II rewrites the standard deviation of estimated results (shown in the third column) into the form of a constant divided by $\sqrt{m}$. It shows that the standard deviation of our algorithm can be accurately approximated by $1.0/\sqrt{m}$. According to our previous analysis in Section VI-D, if the expected relative error of HLL-TC+ is $1.0/\sqrt{m}$, then it can save 45% memory cost than traditional HyperLogLog.

VIII. Conclusion

This paper studies a fundamental problem called cardinality estimation, in the domain of one-pass processing of streaming big data. We present a new solution named HLL-TailCut, which is able to reduce memory consumption by $20\%{\sim}45\%$ than the state-of-the-art HyperLogLog. This remarkable improvement originates from a technique we proposed that truncates the right-side long tail of the register distribution of HyperLogLog. This technique brings two key benefits — improve estimation accuracy by rejecting outliers in the long tails, and compress the register size by recording only sixteen (or eight) highest bars in the histogram of HLL. Therefore, our algorithm can provide the standard error $\frac{1.0}{\sqrt{m}}$ using only four-bit (or three-bit) memory per register. Our HLL-TailCut has also addressed the strong bias problem of HyperLogLog in the small region where it switches to LinearCounting for handling small cardinalities. This is achieved by adopting maximum likelihood estimation technique in this small region.

Appendix A
Probability of Register Value

In this section, we analyze the probability for a HLL register to carry an arbitrary value k. Let V_j be the number of elements that are received by the jth register M_j. Because all the n stream elements are distributed uniformly and pseudorandomly among the m registers in a register set, V_j approximately follows a binomial distribution $Binom(n, \frac{1}{m})$. Hence, the probability for V_j to be equal to a certain value v is

$$Pr\{V_j = v\} = C(n, v)\left(\frac{1}{m}\right)^v \left(1 - \frac{1}{m}\right)^{n-v}, \quad (20)$$

where $C(n, v)$ is the number of combinations when drawing v items from a pool of n items. For this binomial distribution, we denote its mean value by μ and its variance by σ^2.

$$\mu = \frac{n}{m} \qquad \sigma^2 = \frac{n}{m}\left(1 - \frac{1}{m}\right) \quad (21)$$

For the case that V_j equals zero and the register M_j receives no elements, M_j will maintain its initial value of zero. Thus, the probability for the jth register to be zero is

$$Pr\{M_j = 0\} = Pr\{V_j = 0\} = \left(1 - \frac{1}{m}\right)^n, \quad (22)$$

by assigning the symbol v to zero in Eq. (20).

For the case that V_j equals a nonzero value v, we examine what will happen to the jth register if it receives v elements. The random variable M_j, we recall in (1), is the maximum of v random variables that are independently and geometrically distributed according to $Pr\{Y > k\} = \frac{1}{2^k}$ (as depicted in Fig. 1). Hence, the accumulative probability function of M_j is

$$Pr\{M_j \le k \mid V_j = v \wedge v > 0\} = (1 - \frac{1}{2^k})^v. \quad (23)$$

Then, we have

$$Pr\{M_j = k \mid V_j = v \wedge v > 0\} = \big(1 - \frac{1}{2^k}\big)^v - \big(1 - \frac{1}{2^{k-1}}\big)^v.$$

Combining the above equation with (20), we obtain the probability for a register to carry an arbitrary positive value k.

$$\begin{aligned} &Pr\{M_j = k\} \\ &= \textstyle\sum_{v=1}^{n} Pr\{V_j = v\} \cdot Pr\{M_j = k \mid V_j = v \wedge v > 0\} \\ &= \sum_{v=1}^{n} C(n,v)\big(\tfrac{1}{m}\big)^v \big(1 - \tfrac{1}{m}\big)^{n-v} \big[\big(1 - \tfrac{1}{2^k}\big)^v - \big(1 - \tfrac{1}{2^{k-1}}\big)^v\big] \end{aligned} \quad (24)$$

However, it is infeasible to use the Eq. (24) to evaluate the probability for a register to carry an arbitrary value k, which needs to apply summation over the index v ranging from 1 to n. Note that the stream cardinality n may be as large as 10^9 in many applications. Hence, we need to simplify the probability in (24), by eliminating the summation mark $\sum$ and the symbol v. According to the Binomial theorem,

$$\begin{aligned} Pr\{M_j = k\} &= \sum_{v=1}^{n} C(n,v)\big(1 - \tfrac{1}{m}\big)^{n-v} \big[\big(\tfrac{1}{m} - \tfrac{1}{m2^k}\big)^v - \big(\tfrac{1}{m} \\ &\quad - \tfrac{1}{m2^{k-1}}\big)^v\big] = (1 - \tfrac{1}{m2^k})^n - (1 - \tfrac{1}{m2^{k-1}})^n. \end{aligned}$$

APPENDIX B
ITERATIVE NUMERICAL SOLUTION OF HLL-TAILCUT+

In this section, we discuss the HLL-TailCut+ algorithm, which compresses the size of offset register to only three bits and can save memory cost of HLL by 45%. We present how to solve its maximum likelihood problem in (19). We will use the following iterative optimization method to find the optimized estimation for n_b, the number of distinct elements received by the register set when the base register $\mathcal{B}$ equals b:

$$\begin{aligned} &\hat{n}_b^{(i+1)} \\ &= \hat{n_b}^{(i)} + \eta \frac{\partial}{\partial n_b^{(i)}} \log \mathcal{L}(n_b^{(i)} \mid N_0, N_1, \ldots, N_{b+\mathcal{K}-1}) \end{aligned} \quad (25)$$

where $\hat{n_b}^{(i)}$ is the current estimation for n_b, $\hat{n_b}^{(i+1)}$ is the next round estimation, η is the optimization step size configured to $\alpha_m 2^b m$. The derivative of log-likelihood function over n_b, according to the definition of likelihood function in (18), is

$$\begin{aligned} &l \frac{\partial}{\partial n_b} \log \mathcal{L}(n_b \mid N_0, N_1, \ldots, N_{b+\mathcal{K}-1}) \\ &= \sum_{k=0}^{b+\mathcal{K}-1} N_k \frac{\frac{\partial}{\partial n_b} Pr\{\mathcal{B} + \tilde{M}_j = k \mid \hat{n_0}, \hat{n_1}, \ldots, \hat{n_{b-1}}, n_b\}}{Pr\{\mathcal{B} + \tilde{M}_j = k \mid \hat{n_0}, \hat{n_1}, \ldots, \hat{n_{b-1}}, n_b\}}. \end{aligned}$$

The derivative of $Pr\{\mathcal{B} + \tilde{M}_j = k \mid n_0, n_1, \ldots, n_b\}$ is obtained from the derivative of its accumulative distribution function.

$$\begin{aligned} &\frac{\partial}{\partial n_b} Pr\{\mathcal{B} + \tilde{M}_j = k \mid n_0, n_1, \ldots, n_b\} \\ &\quad = \tfrac{\partial}{\partial n_b} Pr\{\mathcal{B} + \tilde{M}_j \le k \mid n_0, n_1, \ldots, n_b\} \\ &\quad\quad - \frac{\partial}{\partial n_b} Pr\{\mathcal{B} + \tilde{M}_j \le k - 1 \mid n_0, n_1, \ldots, n_b\} \end{aligned}$$

By the definition of accumulative distribution function $Pr\{\mathcal{B} + \tilde{M}_j \le k \mid n_0, n_1, \ldots, n_b\}$ in (15), we have

$$\begin{aligned} &\tfrac{\partial}{\partial n_b} Pr\{\mathcal{B} + \tilde{M}_j \le k \mid n_0, n_1, \ldots, n_b\} \\ &\quad = \tfrac{\partial}{\partial n_b} Pr\{\mathcal{B} + \tilde{M}_j^{(b)} \le k \mid n_b\} \\ &\quad \cdot \prod_{i=0}^{b-1} Pr\{\mathcal{B} + \tilde{M}_j^{(i)} \le k \mid n_i\}, \end{aligned} \quad (26)$$

where the derivative $\frac{\partial}{\partial n_b} Pr\{\mathcal{B} + \tilde{M}_j^{(b)} \le k \mid n_b\}$ can be easily obtained from $Pr\{\mathcal{B} + \tilde{M}_j^{(b)} \le k \mid n_b\}$ in (16).

We still need an initial guess of n_b, which is denoted by $\hat{n_b}^{(1)}$ and will be iteratively optimized by (25). If the initial guess is of high quality, then the convergence speed of iterative optimization can be improved. We firstly generate an estimation $\hat{n}$ of the total stream cardinality by HLL-TC in (12), which of course is negatively biased. From the experimental results in Fig. 6(a), we know that HLL-TC often has -5.2% estimation bias. So we generate an initial guess of n_b as

$$\hat{n_b}^{(1)} = \tfrac{1}{1+\mathrm{hlltcBias}}\, \hat{n}_{\mathrm{hlltc}} - \hat{n_0} - \hat{n_1} \ldots - \hat{n_{b-1}}, \quad (27)$$

where $\hat{n}_{\mathrm{hlltc}}$ is the estimation of the total stream cardinality n by HLL-TC, and $\mathrm{hlltcBias}$ is the expected bias of HLL-TC.

REFERENCES

[1] AggregateKnowledge. (2012). *Choose a Good Hash Function*. [Online]. Available: http://research.neustar.biz/2012/02/02/choosing-a-good-hash-function-part-3

[2] K. Aouiche and D. Lemire, "A comparison of five probabilistic view-size estimation techniques in OLAP," in *Proc. 10th Int. Workshop Data Warehousing (DOLAP)*, 2007, pp. 17–24.

[3] Z. Bar-yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan, "Counting distinct elements in a data stream," in *Proc. RANDOM, Int. Workshop Randomization Approximation*, 2002, pp. 1–10.

[4] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla, "On synopses for distinct-value estimation under multiset operations," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2007, pp. 199–210.

[5] J. Cao, Y. Jin, A. Chen, T. Bu, and Z. Zhang, "Identifying high cardinality Internet hosts," in *Proc. IEEE INFOCOM*, Apr. 2009, pp. 810–818.

[6] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *Proc. Eur. Symp. Algorithms (ESA)*, 2003, pp. 605–617.

[7] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high-speed links," *IEEE/ACM Trans. Netw.*, vol. 14, no. 5, pp. 925–937, Oct. 2006.

[8] P. Flajolet, É. Fusy, and O. Gandouet, "HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm," in *Proc. AOFA*, 2007, pp. 137–156.

[9] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for database applications," *J. Comput. Syst. Sci.*, vol. 31, no. 2, pp. 182–209, Oct. 1985.

[10] S. Heule, M. Nunkesser, and A. Hall, "HyperLogLog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm," in *Proc. 16th Int. Conf. Extending Database Technol. (EDBT)*, 2013, pp. 683–692.

[11] A. Kumar, M. Sung, J. Xu, and J. Wang, "Data streaming algorithms for efficient and accurate estimation of flow size distribution," *SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 1, p. 177, Jun. 2004.

[12] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang, "Data streaming algorithms for estimating entropy of network traffic," *SIGMETRICS Perform. Eval. Rev.*, vol. 34, no. 1, p. 145, Jun. 2006.

[13] P. Lieven and B. Scheuermann, "High-speed per-flow traffic measurement with probabilistic multiplicity counting," in *Proc. INFOCOM*, 2010, pp. 1–9.

[14] Y. Lu and B. Prabhakar, "Robust counting via counter braids: An error-resilient network measurement architecture," in *Proc. INFOCOM*, 2009, pp. 522–530.

[15] (2016). *Source Code and Tester Code of HLL-TailCut+ Algorithm*. [Online]. Available: https://www.dropbox.com/s/l0eaexhzvi34x9u/HLLPlus.zip

[16] A. Metwally, D. Agrawal, and A. E. Abbadi, "Why go logarithmic if we can go linear?: Towards effective distinct counting of search traffic," in *Proc. 11th Int. Conf. Extending Database Technol. Adv. Database Technol. (EDBT)*, 2008, pp. 618–629.

[17] S. Nath, P. B. Gibbons, S. Seshan, and Z. Anderson, "Synopsis diffusion for robust aggregation in sensor networks," *ACM Trans. Sensor Netw.*, vol. 4, no. 2, pp. 7:1–7:40, Apr. 2008.

[18] N. Ntarmos, P. Triantafillou, and G. Weikum, "Counting at large: Efficient cardinality estimation in Internet-scale data networks," in *Proc. 22nd Int. Conf. Data Eng. (ICDE)*, Apr. 2006, p. 40.

[19] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications," *ACM Trans. Database Syst.*, vol. 15, no. 2, pp. 208–229, Jun. 1990.

[20] M. Yoon, T. Li, S. Chen, and J.-K. Peir, "Fit a spread estimator in small memory," in *Proc. 28th Conf. Comput. Commun.*, Apr. 2009, pp. 504–512.

[21] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with OpenSketch," in *Proc. USENIX NSDI*, 2013, pp. 29–42.

Qingjun Xiao (Member, IEEE) received the B.Sc. degree from the Computer Science Department, Nanjing University of Posts and Telecommunications, China, in 2003, the M.Sc. degree from the Computer Science Department, Shanghai Jiao-Tong University, China, in 2007, and the Ph.D. degree from the Computer Science Department, The Hong Kong Polytechnic University, in 2011. After graduation, he joined the Georgia State University and the University of Florida, and worked for three years combined as a Post-Doctoral Researcher. He is currently an Associate Professor with Southeast University, China. His research interests include protocol and algorithm design in wireless sensor networks, RFID systems, and network traffic measurement. He is a member of the ACM.

Shigang Chen (Fellow, IEEE) received the B.S. degree in computer science from the University of Science and Technology of China in 1993, and the M.S. and Ph.D. degrees in computer science from the University of Illinois at Urbana-Champaign in 1996 and 1999, respectively.

After graduation, he had worked with Cisco Systems for three years before joining the University of Florida in 2002. His research interests include computer networks, the Internet of Things, big data, cybersecurity, data privacy, edge-cloud computing, intelligent cyber-transportation systems, and wireless systems. He has published over 200 peer-reviewed journal/conference papers. He holds 13 U.S. patents, and many of them were used in software products. He is currently a Professor with the Department of Computer and Information Science and Engineering, University of Florida.

Dr. Chen received the NSF CAREER Award. He has served in various chair positions or as a committee member for numerous conferences. He holds the University of Florida Research Foundation Professorship and the University of Florida Term Professorship. He has served as an Associate Editor for the IEEE TRANSACTIONS ON MOBILE COMPUTING, the IEEE/ACM TRANSACTIONS ON NETWORKING, and a number of other journals. He is also an ACM Distinguished Scientist.

You Zhou received the B.S. degree in electronic information engineering from the University of Science and Technology of China, Hefei, China, in 2013. He is currently pursuing the Ph.D. degree in computer and information science and engineering with the University of Florida, Gainesville, FL, USA. His advisor is Prof. S. Chen. His research interests include network security and privacy, big network data, and the Internet of Things.

Junzhou Luo (Member, IEEE) received the B.S. degree in applied mathematics and the M.S. and Ph.D. degrees in computer network from Southeast University, Nanjing, China, in 1982, 1992, and 2000, respectively. He is currently a Full Professor with the School of Computer Science and Engineering, Southeast University. His research interests are next generation network architecture, network security, cloud computing, and wireless LAN. He is a member of the IEEE Computer Society and the Co-Chair of the IEEE SMC Technical Committee on Computer Supported Cooperative Work in Design. He is a member of the ACM and the Chair of the ACM SIGCOMM China.