POLITECNICO DI TORINO
Repository ISTITUZIONALE

Staggered HLL: Near-continuous-time cardinality estimation with no overhead

Original
Staggered HLL: Near-continuous-time cardinality estimation with no overhead / Cornacchia, A., Bianchi, G., Bianco, A., Giaccone, P.. - In: COMPUTER COMMUNICATIONS. - ISSN 0140-3664. - STAMPA. - 193:(2022), pp. 168-175. [10.1016/j.comcom.2022.06.038]

Availability:
This version is available at: 11583/2970067 since: 2022-09-12T05:33:02Z

Publisher:
Elsevier

Published
DOI:10.1016/j.comcom.2022.06.038

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
Elsevier postprint/Author's Accepted Manuscript

© 2022. This manuscript version is made available under the CC-BY-NC-ND 4.0 license
http://creativecommons.org/licenses/by-nc-nd/4.0/.The final authenticated version is available online at:
http://dx.doi.org/10.1016/j.comcom.2022.06.038

(Article begins on next page)

08 October 2026

Staggered HLL: near-continuous-time cardinality estimation with no overhead

Alessandro Cornacchia[a], Giuseppe Bianchi[b], Andrea Bianco[a], Paolo Giaccone[a]

[a] *Politecnico di Torino, Corso Duca degli Abruzzi 24, Torino, 10129, Italy*
[b] *CNIT / Università degli Studi di Roma - Tor Vergata, Via del Politecnico 1, Roma, 00133, Italy*

Abstract

Most of existing cardinality estimation algorithms do not support natively interval queries under a sliding window model and are thereby insensitive to data recency. We present Staggered-HyperLogLog (ST-HLL), a probabilistic data structure that takes inspiration from HyperLogLog (HLL) and provides nearly continuous-time estimation of cardinality *rates*, rather than absolute *counts*. Our solution has zero-bit overhead with respect to vanilla HLL and negligible additional computational complexity. It is based on a periodic staggered reset of HLL registers and a register equalization operation at query times to compensate for staggered counting. We tested ST-HLL over both synthetic and real Internet traffic traces, showing its ability to track variations of the flow cardinality, quickly adapting to variations under non-stationary flow arrival processes. We show that for the same amount of memory footprint, our algorithm improves the accuracy up to a factor 2x with respect to the state-of-the-art solution, Sliding HLL.

1. Introduction

A classical problem in line-rate network monitoring consists on assessing the *spreading behavior* of a target flow. For instance, the sudden increase in the number of distinct destinations contacted by an IP subnet address space can bring about evidence of network anomalies such as horizontal network scanning activities or Distributed Denial of Service [1, 2]. Furthermore, the spreading nature of a flow can be taken into account for traffic engineering, e.g., by applying specific routing policies to superspreaders [3].

Efficient cardinality estimation of a target stream is a problem pioneered by Flajolet and Martin as early as 1985 [4], and then further addressed and improved in many subsequent works, including [5, 6]. Given a data stream which contains repeated items, the goal of cardinality estimation (a.k.a. distinct counting or count-unique) consists on finding how many items are distinct. Literature works very efficiently address the main obstacle behind the distinct counting problem, namely how to efficiently and scalably remember which items have already been seen in the past to avoid double counting.

However, in contrast to cumulative counting, many applications would rather like to benefit from a capability to "track", in continuous time, the spreading behavior of a target stream or traffic aggregate. Unfortunately, unlike other sketch-based data structures [7, 8, 9, 10], widely used count-unique data structures such as HyperLogLog (HLL) [6] do not provide any efficient mean to *forget* an item seen in the past but no longer seen more "recently". Refactoring such structures so as to permit them to operate using a sliding window or an exponential smoothing coefficient would provide this "short term" memory, but an analysis of the state of the art shows that, among hundreds of works focusing on (Hyper)LogLog-based count-unique data structures, only a couple specifically tackle the problem of devising a sliding-window-based cardinality counter [11, 12, 13].

The reader might argue that, *if the price to pay is an increase in complexity and resource consumption*, the classical and consolidated *binning* approach of taking measurements on independent and subsequent time batches may be an acceptable compromise, although it requires to strike a balance between latency (being results representative only at the end of the batch), and precision (a too small batch size would not be able to track a slow spreader). This leads to the question that motivates our paper: is there a way to *turn* an HLL-like data structure into a continuous-time one, *at no extra resource cost* in terms of increased number of internal counters, or increased per-counter memory, or extra hardware?

Our approach at a glance

Let us first recall that, if we could afford to deploy and run multiple HLL data structures in parallel, opposed to a single HLL reset every W seconds, we could use a *staggered* approach, as shown in Fig. 1. Such a combined structure would permit to employ a (possibly long) measurement window W, but would allow to track the traffic dynamics at a finer time scale $\tau = W/N$, where N is the number of parallel HLLs deployed. Each HLL would in fact "learn" about the incoming new items (equivalent to new traffic flows), but only one "active" HLL (the "oldest" one) would have accumulated arrivals for W time and would be in charge to report the current cardinality estimation (the bottom one in Fig. 1), hence achieving a staggered approximation of a sliding window operation.

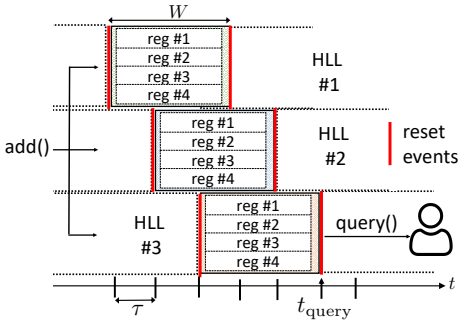


Figure 1: Staggering three HLLs, each of them with 4 registers. The query reads a single HLL and can happen just before resetting the register.

Such a straightforward approach is practically ruled out by the unaffordable N-fold increase in required resources, as N fully fledged HLL should be deployed instead of a single one. However, this baseline approach naturally suggests the following apparently naïve idea: would it be possible to achieve the same result by *staggering the internal registers of a single HLL*? As shown in Fig. 2, this operation would come along with *zero* additional resource cost - it would suffice to deploy a staggered timer which periodically resets only *one single HLL register at a time* - anything else would be left unmodified.

The main contribution of this paper consists on turning such a naïve idea into an effective approach. To accomplish this goal, we need to address a major technical caveat: since such staggered registers now "count" items on different time windows, we cannot neither resort to the classical HLL stochastic averaging process to reduce the estimation error, nor their simple aggregation would work. It is trivial to check (see also discussion at the end of Sec. 3.1) that the estimation error would *increase* rather than *decrease*.

We address such hurdle by "equalizing" the registers by a proper scaling in such a way that all registers will estimate the rate on the "same" time window. This permits to exploit the well-known results for classical HLLs and compensate for the systematic errors with standard techniques.

In summary, our main contributions are the following:

- we propose a register-based staggered HLL, which enables near-continuous-time measurements at no extra cost with respect to a standard HLL;
- we evaluate its performance and show that it outperforms state-of-the-art solutions;
- we highlight many design and operation issues typical of continuous-time measurements which could be adapted to other probabilistic data structures.

The rest of the paper is organized as follows. In Sec. 2 we recall the HLL data structure and we discuss the related work. In Sec. 3 we present our proposed solution, whose accuracy is numerically evaluated in Sec. 4. Finally, we draw our conclusions in Sec. 5.

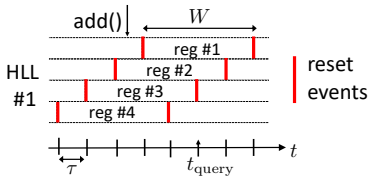


Figure 2: Staggering the 4 registers using a single HLL.

2. Background and related work

2.1. HyperLogLog data structure

HyperLogLog (HLL) is a probabilistic data structure for cardinality estimation proposed in [6] and derived from the probabilistic counting approach first proposed in [4]. We now describe the main idea of probabilistic counting applied to traffic flows, but the description can be rephrased in terms of counting generic items in data streams. We wish to evaluate the cardinality n of a set of flows X observed in a given time interval, i.e., with $|X| = n$. Now we can evaluate an hash function $h(x)$ on each flow $x \in X$ and compute the position[1] $p(h(x))$ of the left-most bit equal to 1 in the binary representation of $h(x)$. Let $R = p(h(x))$ be the *rank* of a flow x and observe that $\text{Prob}(R = r) = 2^{-r}$, i.e., R follows a standard geometric distribution, given the uniformity property of the hash function. This suggests the main idea behind the HLL estimator: by just knowing the maximum rank, i.e., $R_{\max} = \max_{x \in X} p(h(x))$ it is possible to roughly approximate the distinct number of flows as $n = 2^{R_{\max}}$. Note that, by construction, packets belonging to the same flow have the same value of the hash function and thus multiple packets of the same flow are counted just once. It is easy to see that this estimator is characterized by a large variance, due to possibility of outlier flows (i.e. flows with $R > \log_2 n$) that would blow up the estimation. HLL reduces such variance through *stochastic averaging*, introduced in [4]. This technique splits uniformly at random the traffic stream into m substreams, and for each substream stores the maximum rank $R^i_{\max}$, $i = \{1, .., m\}$, in a *register* (i.e., an integer value). Then, m independent estimations from the registers are aggregated to reduce noisy fluctuations, increasing the quality of the estimate. Flajolet et al. [6] demonstrated that taking the harmonic mean across individual register estimations reduces the relative error to $1.04/\sqrt{m}$ with respect to $1.30/\sqrt{m}$ of its predecessor, LogLog [5]. A graceful property of stochastic averaging is that it mimics the effect of multiple independent estimations using a single hash function: given a flow x, the first $\log_2(m)$ bits of $h(x)$ are used to select the register and the remaining bits to extract the rank.

[1] The position is counted starting from one.

Estimating the number of distinct items in a stream with high accuracy at low complexity is a long-standing problem for which several approximate algorithms have been proposed. Some well-known examples include *Linear Counting* [14], *PCSA* [4], *MinCount* [15, 16], *Multiresolution Bitmap* [17], *LogLog* and *SuperLogLog* [5] and the latest evolution *HyperLogLog++* [18]. A comprehensive overview and quantitative comparison can be found in the survey papers of Metwally et al. [19] and Harmouch et al. [20]. The approach we develop in our work takes as a reference the widespread HyperLogLog algorithm, which is part of several industrial products, like Google BigQuery [21], Microsoft Kusto Query Language [22] and Facebook distributed SQL engine [23]. Nevertheless, the same staggered approach could in principle be applied to other algorithms, provided they preserve the same functional architecture based on registers (e.g., MinCount and LogLog alternatives).

2.2. HyperLogLog over sliding windows

Vanilla HLL provides accurate estimation about the number of distinct items. However, it can only answer cardinality queries that refer to the stream history starting from the last register reset event. A strawman solution to adopt HLL under the sliding window model is to buffer, for every substream, all the ranks observed in a window duration. Then, since the m maxima among the stored ranks can be computed, the strawman solution would be capable of answering time-range queries using the HLL technique discussed in Sec. 2.1. However, this simple solution doesn't scale well as the amount of ranks to keep in memory grows fast for increasing window sizes. *Sliding HLL* [11] improves the strawman solution, thanks to the intuition that only the ranks eligible to become maxima in the future need to be maintained. Sliding HLL keeps m distinct lists called List of Possible Future Maxima (LPFM), containing pairs of the kind $\langle$*timestamp, rank*$\rangle$. When the rank of a new item is inserted into one of the LPFMs, all ranks smaller than the new one are evicted from the list, together with the ranks oldest than a past window. Therefore, differently from the strawman solution, the arrival of a large *possible future maxima* allows to free up LPFM entries, significantly reducing the memory consumption needed to maintain recent information. The asymptotic memory cost of Sliding HLL is upper bounded by $5m\ln(n/m)$, being n the maximum number of flows per window, which should be compared with $m\log_2\log_2(n/m)$ for vanilla HLL. Our algorithm, Staggered HLL, requires the same space of vanilla HLL and does not have to pay the extra complexity of searching for m maxima when executing a query.

2.3. Related solutions

Other streaming algorithms designed for a sliding window model have been recently proposed. Similarly to Sliding HLL, *SWAMP* [9], *WCSS* [8], *Memento* [24] and *Sequential zeroing* [25] are all based on the removal of outdated information from their data structures, so that only the most recent items contribute to the estimation. However, these works only support event-based windows, usually defined in terms of number of packets, and not time-interval queries. SWAMP can answer set-membership, frequency, cardinality and entropy queries with a single data structure, which is a circular buffer to track fingerprints of recent items, plus an auxiliary counting hash table to store their frequencies. Being designed with generality in mind, SWAMP is memory demanding. Moreover, the use of a *TinyTable* [26] makes complex its implementation on programmable switches. Ivkin et al. [13] devised an elegant sketch-based framework that allows to specify the time frame of interest as a query parameter. Similar to [12], it offers an integrated solution for various measurement types in a single structure, however it needs at least 56MB of memory to accurately detect a DDoS attack. Our approach limits its scope only to a single task (cardinality estimation), but with much smaller memory footprint. A different class of algorithms [27, 28] down-weights the relative importance of aged items with respect to recent ones. *AdaSketches* [27] is a time-aware sketch that emphasizes newly inserted items with a function that monotonically increases with the timestamp of arrival, so providing higher query accuracy to recent events. A customization of AdaSketches tailored to commodity switches can be found in [29], but, aiming at frequency estimation, it is orthogonal to our work.

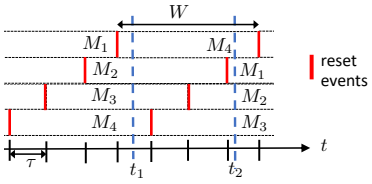


Figure 3: Register-based staggered HLL with 4 registers sampled at time t_1 (when the sorted sequence of registers is $[M_1, M_2, M_3, M_4]$) and at time t_2 (when the sorted sequence of registers is $[M_2, M_3, M_4, M_1]$).

3. Register-based staggered HLL

We start to describe the proposed solution to adapt a single HLL to time-continuous measurements. This highlights the issue of managing heterogenous registers, which must be properly scaled and equalized as discussed in Sec. 3.2. Our solution, denoted as ST-HLL, is finally presented in details in Sec. 3.3.

3.1. Overview, notation and assumptions

Similarly to a standard HLL data structure, let us consider m registers. We introduce two different time scales: i) a *smoothing time scale* W, and ii) an *updating time scale* τ. W is a window size which defines the target memory depth of the data structure. $\tau \ll W$ is the time scale at

which the HLL is updated. For reasons that will become clear later on, it is convenient to set $\tau = 2W/m$. Considering that m is usually set to a relatively large value, say 512 or 1024, a relatively long window W, e.g., 4 minutes, would be updated at a rate of about 1 or 2 times per second, thus producing a near-continuous-time effect for monitoring applications which aim to follow traffic dynamics and updates at a time scale in the order of seconds.

Our staggered HLL approach builds upon the idea of resetting only *one* among the m registers at each time slot τ. More specifically, we reset counters in a circular fashion, as shown in Fig. 3. This implies that each register in the HLL will track a *different* time period: At the time in which a register is reset, the previously reset one will have tracked its fraction of traffic arrived in the latter slot τ, the second previous one will have tracked a time interval $2 \cdot \tau$, and so on. It is convenient to rank all the registers based on the reset time, and denote with M_i the value of the i-th most recent register reset, with $1 \leq i \leq m$. Owing to the above convenient notation, at an arbitrary time instant t, register M_i will have counted arrivals in the interval $(\lfloor t/\tau \rfloor \tau - (i-1)\tau, t]$.

Let us now assume throughout the remainder of this paper that *the number of new items recorded by each register is proportional to the size of its measurement period* - see also Fig. 2 (we'll discuss this assumption in more depth in Sec. 4.2). If we "read" the status of all registers at a time t exactly in the middle of the updating time slot τ, and we assume that the overall rate of new arrivals is λ items per second, then register i, which tracks $1/m$-th of the traffic, will have recorded a fraction λ/m of new arrivals for a time period $(i-1/2)\cdot\tau = (i-1/2)\cdot\frac{2W}{m}$. Hence, by *summing* the content of all the m counters, we would trivially obtain an estimate[2] of the number of new arrived items in a time period W, as shown in the following:

$$\sum_{i=1}^{m} \frac{\lambda}{m}(i-1/2)\cdot\frac{2W}{m} = \lambda W$$

Unfortunately, summing the content of all counters is not a viable approach, as the variance of the so-obtained estimator would dramatically *increase* rather than decrease. The estimation error would reduce by taking a stochastic average, but this is not anymore straightforward in our case, as the registers record estimates taken on different time periods, hence they ultimately estimate *different* quantities.

3.2. Dealing with heterogenous registers

As anticipated in the previous section, the above described construction brings about a crucial difference with respect to the classical HLL data structure. In standard HLL, the uniform split of the traffic across the m deployed registers makes such registers *statistically homogeneous*, i.e., each register yields an estimate of the same quantity. In our case, each register instead used a *different* measurement window, and therefore accounts for a different number of items.

To establish a quantitative insight on how each register' statistics depend on the number of accounted items, it is instructive to note that this relation would become trivial if each register R, instead of being an integer random variable, were approximated by a *continuous random variable*. Indeed, for such "continuous" register, the probability that a new arriving item "hits" a given (real-valued) register position would now follow an exponential law instead of the geometric one introduced in Sec. 2.1, i.e., $\text{Prob}(R > x) = 2^{-x}$.

Let us now assume that n items are accounted by the register. It readily follows that the random variable $R_{\max}$ representing the current register's state, i.e., the maximum value among the n values drawn from the above exponential distribution, has cumulative probability distribution given by the product of the n exponential distributions; in formulae:

$$\text{Prob}(R_{\max} \leq x) = \left(1 - 2^{-x}\right)^n$$

Such continuous distribution is very convenient, as it yields very simple closed-form expressions for the statistical moments[3]. Routine computation indeed yields the register's expected value:

$$\mathbb{E}[R_{\max}] = \frac{H_n}{\ln(2)} \quad \overset{n\to\infty}{\approx} \quad \frac{\gamma}{\ln(2)} + \log_2(n) \qquad (1)$$

where $H_n = \sum_{i=1}^{n} 1/i$ are the well known Harmonic numbers H_n, and the approximation follows from the definition of the Euler constant $\gamma = \lim_{n\to\infty}(H_n - \ln n) = 0.5772$.

For our purposes, it is important to further note that the variance of the register does not diverge for large n, but rather converges to a constant quantity:

$$\text{Var}[R_{\max}] = \frac{H_n^{(2)}}{\ln(2)^2} \quad \overset{n\to\infty}{\approx} \quad \frac{\pi^2}{6\ln(2)^2} = 3.424$$

where $H_n^{(r)} = \sum_{i=1}^{n} 1/i^r$ is the Harmonic number of order r, and $\lim_{n\to\infty} H_n^{(2)} = \pi^2/6$. In Fig. 4 we show the coefficient of variation of $R_{\max}$, denoted as:

$$\text{Cv}(R_{\max}) = \frac{\sqrt{\text{Var}(R_{\max})}}{\mathbb{E}(R_{\max})}$$

[2] This explains why we have specifically selected $\tau = 2W/m$. With such setting, the "shorter" time of the most recently reset counters is compensated by the "older" counters which can account for up to a time interval of $2W$ before being reset.

[3] Being a positive random variable, the moments can be directly computed from the complementary cumulative probability distribution as

$$E\left[X^r\right] = r\int_{x=0}^{\infty} x^{r-1}P(X > x)dx = r\int_{0}^{\infty} x^{r-1}\left(1 - \left(1-2^{-x}\right)^n\right)dx$$

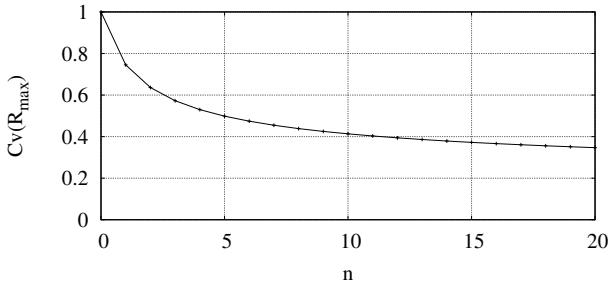


Figure 4: Evaluation of relative error $\mathrm{Cv}(R_{\max})$ in a HLL register given a number n of recorded flows.

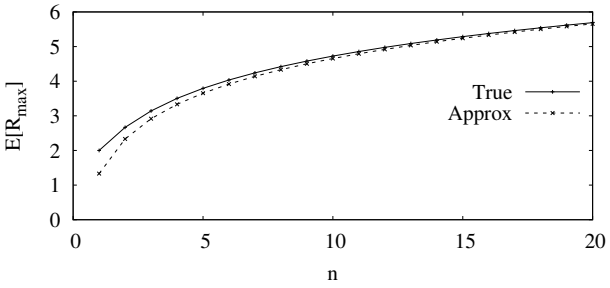


Figure 5: Evaluation of the error in $\mathbb{E}[R_{\max}]$ when considering the approximation (2).

which can be seen as the relative error of a HLL for a given number of flows counted in a HLL register. From the plot, it is clear that the most inaccurate registers are the ones with few arrived flows. This is a crucial observation, since in ST-HLL, by constructions, the error depends on the register, differently from a standard HLL in which all the registers are fed by homogenous arrivals, leading to errors identical on average. The main problem arises when combining together the estimations provided by the registers, since the average should take into account the different levels of accuracy characterizing each register.

So far, for simplicity, we have assumed "continuous" - exponentially distributed - register values, whereas an HLL register of course can only assume integer values. However, this is trivially accommodated by adding a constant $1/2$ that accounts for such quantization. More formally, this extra constant yields from eq. (2.8) in [30], which provides an explicit approximation for the expectation of the maximum of i.i.d. geometric distributions. It follows that equality (1) is readily adapted to the discrete case as

$$\mathbb{E}[R^g_{\max}] \approx \frac{1}{2} + \mathbb{E}[R_{\max}] = \gamma' + \log_2(n) \tag{2}$$

where the superscript g refers to the geometric distribution of the actual HLL registers, and $\gamma' = 1/2 + \gamma/\ln(2) = 1.332$. Observe that (2) provides the most accurate formula we will need in the following for actual value stored in a HLL counter, given n recorded flows. Fig. 5 shows the true value for $\mathbb{E}[R^g_{\max}]$ and its approximation according to (2). The relative error is very small (e.g., $< 4\%$ for $n = 5$) and decreases with n. Thus the approximation (2) is accurate also for small values of n.

3.3. Scaling and equalization in Staggered HLL

We now have all the tools necessary to specify our proposed *staggered HLL* scheme, described in details in the pseudocode of Fig. 2. Assume, non restrictively, that we are interested in reading the overall HLL count at a time instant t corresponding to the end of a time slot τ, i.e. right before the "next" register is reset, and define W_i as the time window span of the i-th register M_i, $i \in \{1, m\}$. Owing to our notation,

$$W_i = i \times \tau = i\frac{2W}{m}.$$

Let now t be the actual HLL reading time. Since each register receives a fraction $1/m$ of the items, the expected number of items n_i accounted by each register M_i in its time window W_i is:

$$n_i = \int_{t-W_i}^{t} \frac{\lambda(t)}{m} dt$$

If we now assume a constant arrival rate within the past W_i interval of time, i.e., $\lambda(t) = \lambda$, then:

$$n_i = \frac{\lambda}{m}\frac{2W}{m}i \tag{3}$$

which provides an explicit relation between λ and the expected number of arrivals in a specific register. By inverting (3) it, we can derive a local estimation $\hat{\lambda}_i$ of the arrival rate at register M_i:

$$\hat{\lambda}_i = \frac{n_i m^2}{2Wi} \tag{4}$$

Thanks to (2), we can estimate the expected value of a generic register M_i based on its corresponding window W_i as

$$\mathbb{E}[M_i] = \gamma' + \log_2(n_i)$$

and then, by a first order approximation, we can claim

$$n_i = 2^{M_i - \gamma'}$$

which allows to rewrite (4) as follows:

$$\hat{\lambda}_i = 2^{M_i - \gamma'}\frac{m^2}{2Wi} \tag{5}$$

It follows that the above equation (5) permits to turn the *heterogeneous* registry values M_i into estimators $\hat{\lambda}_i$ of a *same* quantity λ, namely the overall arrival rate of new items to the HLL data structure. We can hence now proceed exactly as in the case of a standard HLL, i.e., compute the harmonic mean of all $\hat{\lambda}_i$ by scaling by a proper factor

1: **procedure** QUERY()
2: **for** $i \leftarrow 1$ to m **do** ▷ For each register
3: $\hat{\lambda}_i = 2^{(M_i-1)} \frac{m^2}{Wi}$ ▷ Estimate λ locally
4: $\hat{\lambda} = \alpha_m$ HARMONICMEAN$([\hat{\lambda}_i]_{i=1}^m)$ ▷ Estimate λ globally
5: **if** $\hat{\lambda} \leq \frac{5}{2}m$ **then** ▷ Small range corrections - linear counting
6: $v = |\{M_i, i \in \{1, \ldots, m\} | M_i = 0\}|$ ▷ Num. zero registers
7: **if** $v \neq 0$ **then**
8: **return** $m \ln(m/v)$
9: **if** $\hat{\lambda} > \frac{1}{30} 2^{32}$ **then** ▷ Large range corrections
10: **return** $-2^{32} \ln(1 - \hat{\lambda}/2^{32})$
11: **return** $\hat{\lambda}$ ▷ Medium range - no corrections

Figure 6: The query algorithm for ST-HLL.

α_m, which has been computed in [6] and also accounts for γ' in (3). Approximately, $\alpha_m \approx 0.7$.

The pseudocode of the query function is reported in Fig. 6. At a first step, a local estimation of the rate at each register is evaluated using (5) (ln. 2-3). As second step, the local estimations are combined together though an harmonic average, according to the standard methodology for HLL (ln. 4). Finally some well-known correction factors to the rate estimator, as derived in [6] for standard HLL, are applied to consider different level of "occupancy" of each register (ln. 5-11).

In terms of implementation complexity, the proposed solution is identical to a standard HLL, without the need of additional memory as in alternative solutions, as discussed in Sec. 2. Thus, the total memory is $m \log_2 \log_2(n/m)$ where n is the maximum number of flows during the observation window. An internal timer must be available to trigger the reset of a single register every τ time.

3.4. Low-counter variance compensation

As discussed in Sec. 3.2, Fig. 4 shows that the accuracy of the estimated number of flows in a register depends greatly from the number of recorded flows, and thus the estimators referring to the lowest ranked registers $(M_1, M_2, \ldots)$ are the most inaccurate. In order to reduce the variance in the final evaluation of the rate estimator, we propose the following heuristic approach: when computing the harmonic mean (ln. 3 in the pseudocode of Fig. 6), we consider only the registers with index i larger than a given threshold $i_{\min}$. Even if $i_{\min}$ requires some tuning, we could observe a good tradeoff between accuracy and temporal granularity by setting $i_{\min} \approx m/8$, i.e., we neglect the 12.5% of the (supposed) lowest registers. This variant of ST-HLL with the variance compensation technique will be later denoted as ST-HLL+. Notice that in ST-HLL+ all the counters are updated as usual when a new packet arrives, whereas the compensation is applied only at query time during the final average computation.

4. Performance evaluation

We have assessed the performance of ST-HLL using both i) synthetic traffic traces suitably crafted so as to test how our scheme responds to significant traffic fluctuations (see Sec. 4.1), and ii) real world traffic traces, which we used to quantify the effectiveness of our approach also compared with the sliding-window HLL solution proposed in [11] (see Sec. 4.2).

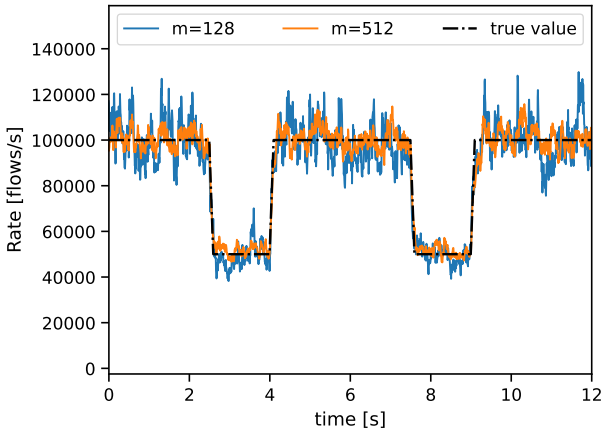


Figure 7: SQU traffic scenario, $W = 0.1$ s.

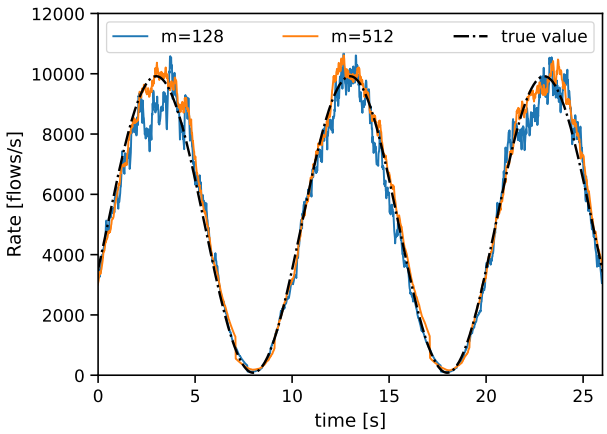


Figure 8: SIN traffic scenario, $W = 1$ s.

4.1. Synthetic traffic streams

In order to asses the tracking effectiveness of ST-HLL in answering window-based cardinality queries and track variations in the flow arrival rate, we developed an ad-hoc numerical simulator in python, which permits to test different input traffic scenarios. The simulator generates synthetic traffic workloads, where each flow consists of a single packet. Therefore, the traffic stream is composed only of packets belonging to distinct flows, and thus each packet is accounted as a new item by the HLL counter. Such packets are generated according to a non-stationary Poisson process, whose instantaneous rate is modulated by (in principle) an arbitrary function $\lambda(t)$. We specifically report results for the following two scenarios:

- Squared wave traffic (SQU, Fig. 7), in which $\lambda(t)$ varies between 50,000 and 100,000 flows/s with a period of 5 seconds and a duty cycle equal to 70%.

Here, we have used a relatively short window $W = 0.1$ s to specifically assess the ability of ST-HLL to promptly follow abrupt traffic fluctuations.

- Sine wave traffic (SIN, Fig. 8), in which $\lambda(t)$ varies according to a sinusoidal function between 0 and 10,000 flows/s, with a period equal to 10 s - we here used a longer window $W = 1$ s.

The plots shown in the figure are obtained by periodically reading the content of the ST-HLL counter (hence by performing the query procedure which computes the estimated rate sample), with a sampling time multiple of the register's reset time slot τ. We removed the transient phase from our numerical results.

In both figures, we compare the results obtained by two ST-HLL settings ($m = 128$ and $m = 512$) with the *true value* obtained by filtering the nominal arrival rate with a moving average window of duration W - in essence, by comparing with an ideal sliding window counter of depth W. Note that this comparison is somewhat unfair for us, as our ST-HLL filter does not implement a "rectangular" sliding window of depth W, but mimics a "triangular" window[4], as it combines low-index registers, which use window depths lower than W, with high-index registers which instead measure items on a time period which may span up to twice the size of the nominal window W.

From both figures, we remark that ST-HLL remains very close to the true cardinality count. Its gap with the true count remains most of the time within $1.04/\sqrt{m}$, which is in line with the theoretical error bounds derived for HLL. For $m = 512$ this value is 4.59. The estimate remains close to the true value even when the rate $\lambda(t)$ gets close to zero. Notice that in this region most of the registers of ST-HLL are empty, so as in [6] we resort to linear counting, for which the accuracy guarantees might not be the same of HLL.

4.2. Real Internet traces

To provide more realistic results, we analyze the algorithm's performance over real Internet traces [31, 32]. We considered two traces, CAIDA-2018 and CAIDA-2019, collected in a backbone router link at Equinix-New York and whose main features are reported in Table 1.

A major motivation behind such experiments consists in assessing whether our proposed estimation is robust also in the case of real traces. Indeed, unlike the synthetic traffic used in the previous section, real world traces may not anymore closely follow our baseline modeling assumption stated in Sec. 3.1, i.e., that *the number of new items recorded by each register is proportional to the size of its measurement period*. In practice, in real world traffic, each flow may in fact appear more than once inside the sliding window, thus leading to the presence of duplicate items, and the frequency and burstiness of such duplicate items largely varies across different flows.

It is intuitive to see that duplicate elements have a much greater impact on short measurement windows than on long ones. For an extreme example, a single persistently recurring flow would be always accounted as "+1" on any measurement window size. Hence, its impact on the rate estimation would be significantly greater for a short window rather than for a long one [5].

Indeed, the above intuition was confirmed by our experiments on the CAIDA traces. Fig. 9 in fact shows a 6-7% bias in the rate estimation obtained by a 1024 register ST-HLL. However, the above discussion also suggests that the very simple heuristic introduced in Sec. 3.4 for a different purpose, namely reduce the impact of the more noisy low-index registers, can also effectively mitigate the impact of duplicate flows. We recall that such heuristic trivially consists in discarding a relatively small fraction (1/8 in our experiments) of low-indexed counters when computing the rate estimation. And since low-index registers are those which more severely affect the rate estimation, we expect a significant improvement in the estimation itself. This is experimentally confirmed in Fig. 9 by the dramatic increase in the accuracy of the ST-HLL+ plot with respect to the baseline ST-HLL counter.

To gather further quantitative insights on the performance of ST-HLL and of the ST-HLL+ heuristic, we ran an extensive set of results for both CAIDA traces and for a variety of different ST-HLL parameter settings and sliding windows durations. Results are shown in Figs. 10-11. As a further term of comparison, we compared our proposed staggered construction with the Sliding HLL (denoted as "W-HLL" in the following), which is the state-of-the-art solution proposed in [11] and discussed in Sec. 2. Note that, unlike our proposed approach, W-HLL uses extra memory. Hence, for a fair comparison, ST-HLL and W-HLL have been compared given the same overall memory. Now computing the ratio between the memory of the W-HLL and the memory of the ST-HLL, we get: $5\ln(n/m) \times (\log_2 \log_2(n/m))^{-1}$, which is above 7 for $n/m = 18$. Since in our traces the number of flows per

[4] Indeed, being $\lambda(t)$ the instantaneous arrival rate, a classical sliding window would produce a smoothed measured rate $r(t) = \int_{t-W}^{t} \lambda(x)/W dx$. Instead, for a large number of HLL registers, our smoothed measured rate would converge to:

$$r(t) = \int_{t-2W}^{t} \lambda(x) \left(1 - \frac{t-x}{2W}\right) dx.$$

i.e., the convolution of $\lambda(t)$ with a triangle gate function. Hence, even assuming an ideal operation, our results are in principle expected to (slightly, if W is relatively small with respect to the traffic dynamics) differ from those obtained by a pure sliding window.

[5] This is a direct consequence of our definition for the local rate estimator (2): if we add such extra persistent flow to the remaining n_i measured by the i-th register, i.e., we alter the original estimation $\hat{\lambda}_i$ as:

$$\hat{\lambda}'_i = \frac{(n_i+1)m^2}{2Wi} = \hat{\lambda}_i + \frac{m^2}{2Wi},$$

then its relative impact would be significantly greater for small values of i.

Scenario	Trace	Ave. Bitrate	Link rate	Num. Packets	Num. Flows
CAIDA-2018	equinix-nyc-2018	4.26 Gbps	10 Gbps	37.8M	1.8M
CAIDA-2019	equinix-nyc-2019	4.49 Gbps	10 Gbps	36.7M	1.2M

Table 1: Main features of the considered CAIDA traffic traces

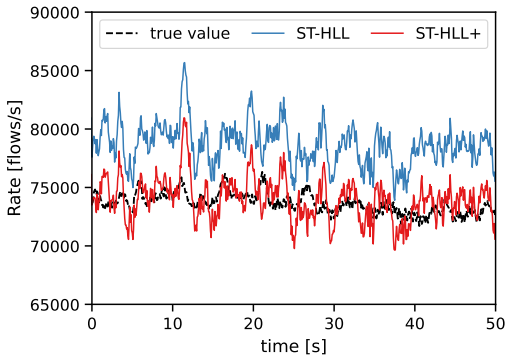
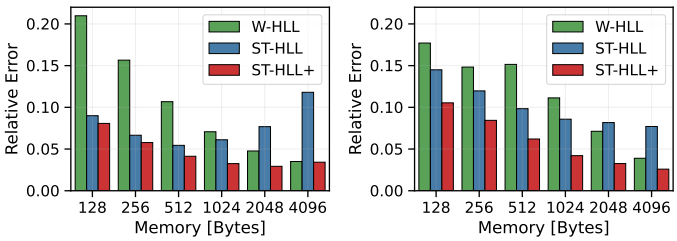


Figure 9: Cancelling the over-counting effect of ST-HLL when duplicated items are present. CAIDA 2019, $W = 1$ s, $m = 2048$.



(a) $W = 0.1$ s (b) $W = 1$ s

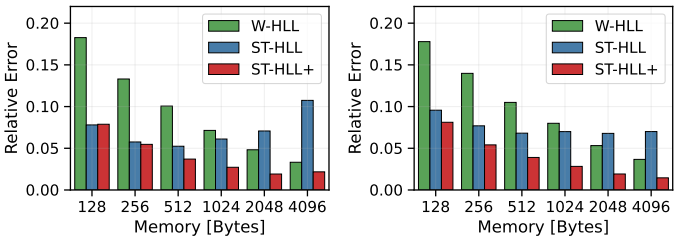
Figure 10: CAIDA-2018 traffic scenario.

observation window is around 50-80k, n/m is well above 18, we consider that in practise the memory of W-HLL is 8 times larger than ST-HLL for a fixed number of registers. This consideration will allow us to compare ST-HLL and W-HLL given the same amount of memory, scaling the number of registers by the same factor.

As Figs. 10-11 show, with less than 1KB of memory available, both ST-HLL and ST-HLL+ give more accurate estimates than W-HLL. In all considered traffic scenarios, W-HLL needs at least 4× more memory compared to ST-HLL+ to guarantee a relative estimation error smaller than 5%.

5. Conclusion

We have addressed the problem of estimating, in near-continuous time, flow arrival rates for real-time traffic streams. We have proposed Staggered-HyperLogLog (ST-HLL), a register-based probabilistic data structure which does not introduce any memory overhead with respect to vanilla HLL. Thanks to a proper periodic resets of the registers and an equalization of the rate estimators of each register, we showed that ST-HLL estimates the rate for an



(a) $W = 0.1$ s (b) $W = 1$ s

Figure 11: CAIDA 2019 traffic scenario.

arbitrary observation window, while capturing rate variations in near-continuous time. We validated the proposed approach throughout extensive simulations, using both synthetic and real traffic traces, and showed that ST-HLL achieves a much better accuracy with respect to other state-of-the-art solutions, given the same memory footprint.

Extending a standard HLL, designed to count, to estimate rates has been proved to require facing many design issues. The proposed solution could be exploited in other probabilistic data structures, paving the way to novel evolution of data structures specifically tailored for real-time traffic monitoring and stream processing.

ACKNOWLEDGMENTS

Computational resources were provided by HPC@POLITO, which is a project of Academic Computing within the Department of Control and Computer Engineering at the Politecnico di Torino (http://hpc.polito.it).

References

[1] Y. Chabchoub, R. Chiky, B. Dogan, How can sliding HyperLogLog and EWMA detect port scan attacks in IP traffic?, EURASIP Journal on Information Security 2014 (1) (2014).

[2] V. Bruschi, S. Pontarelli, J. Tollet, D. Barach, G. Bianchi, Flowfight: High performance–low memory top-k spreader detection, Computer Networks 196 (2021) 108239.

[3] Y. Liu, W. Chen, Y. Guan, Identifying high-cardinality hosts from network-wide traffic measurements, IEEE Transactions on Dependable and Secure Computing 13 (5) (2015) 547–558.

[4] P. Flajolet, G. N. Martin, Probabilistic counting algorithms for data base applications, Journal of computer and system sciences 31 (2) (1985) 182–209.

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

[6] P. Flajolet, É. Fusy, O. Gandouet, F. Meunier, HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm, in: Conference on Analysis of Algorithms (AofA), 2007.

[7] G. Bianchi, N. d'Heureuse, S. Niccolini, On-demand time-decaying bloom filters for telemarketer detection, SIGCOMM Computer Communication Review 41 (5) (2011) 5–12.
[8] R. Ben-Basat, G. Einziger, R. Friedman, Y. Kassner, Heavy hitters in streams and sliding windows, in: INFOCOM, IEEE, 2016, pp. 1–9.
[9] R. B. Basat, G. Einziger, R. Friedman, Give me some slack: Efficient network measurements, Theoretical Computer Science 791 (2019) 87–108.
[10] X. Gou, Y. Zhang, Z. Hu, L. He, K. Wang, X. Liu, T. Yang, Y. Wang, B. Cui, A sketch framework for approximate data stream processing in sliding windows, IEEE Transactions on Knowledge and Data Engineering (2022).
[11] Y. Chabchoub, G. Heébrail, Sliding HyperLogLog: Estimating cardinality in a data stream over a sliding window, in: International Conference on Data Mining Workshops, IEEE, 2010.
[12] E. Assaf, R. B. Basat, G. Einziger, R. Friedman, Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free, in: INFOCOM, IEEE, 2018, pp. 2204–2212.
[13] N. Ivkin, R. B. Basat, Z. Liu, G. Einziger, R. Friedman, V. Braverman, I know what you did last summer: Network monitoring using interval queries, Measurement and Analysis of Computing Systems 3 (3) (2019) 1–28.
[14] K.-Y. Whang, B. T. Vander-Zanden, H. M. Taylor, A linear-time probabilistic counting algorithm for database applications, ACM Transactions on Database Systems (TODS) 15 (2) (1990) 208–229.
[15] F. Giroire, Order statistics and estimating cardinalities of massive data sets, Discrete Applied Mathematics 157 (2) (2009) 406–427.
[16] F. Giroire, Réseaux, algorithmique et analyse combinatoire de grands ensembles, Ph.D. thesis, Paris 6 (2006).
[17] C. Estan, G. Varghese, M. Fisk, Bitmap algorithms for counting active flows on high speed links, in: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement, 2003, pp. 153–166.
[18] S. Heule, M. Nunkesser, A. Hall, Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm, in: Proceedings of the 16th International Conference on Extending Database Technology, 2013, pp. 683–692.
[19] A. Metwally, D. Agrawal, A. E. Abbadi, Why go logarithmic if we can go linear? towards effective distinct counting of search traffic, in: International conference on Advances in database technology (EDBT), 2008, pp. 618–629.
[20] H. Harmouch, F. Naumann, Cardinality estimation: An experimental survey, Proc. VLDB Endow. 11 (4) (2017) 499–512.
[21] E. Bisong, Google BigQuery, Apress, Berkeley, CA, 2019, pp. 485–517.
[22] Kusto query language (kql) overview, `https://docs.microsoft.com/en-us/azure/data-explorer/kusto/query/`, accessed: 05-2022.
[23] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, et al., Presto: Sql on everything, in: 2019 IEEE 35th International Conference on Data Engineering (ICDE), IEEE, 2019, pp. 1802–1813.
[24] R. B. Basat, G. Einziger, I. Keslassy, A. Orda, S. Vargaftik, E. Waisbard, Memento: Making sliding windows efficient for heavy hitters, in: CoNEXT, 2018.
[25] B. Turkovic, J. Oostenbrink, F. Kuipers, I. Keslassy, A. Orda, Sequential zeroing: Online heavy-hitter detection on programmable hardware, in: IFIP, IEEE, 2020.
[26] G. Einziger, R. Friedman, Counting with tinytable: Every bit counts!, in: Proceedings of the 17th International Conference on Distributed Computing and Networking, 2016, pp. 1–10.
[27] A. Shrivastava, A. C. Konig, M. Bilenko, Time adaptive sketches (ada-sketches) for summarizing data streams, in: SIGMOD, ACM, 2016.
[28] G. Cormode, V. Shkapenyuk, D. Srivastava, B. Xu, Forward decay: A practical time decay model for streaming systems, in: International conference on data engineering, IEEE, 2009.
[29] Y. Qiu, K.-F. Hsu, J. Xing, A. Chen, A feasibility study on time-aware monitoring with commodity switches, in: Proceedings of the Workshop on Secure Programmable Network Infrastructure, 2020, pp. 22–27.
[30] W. Szpankowski, V. Rego, Yet another application of a binomial recurrence order statistics, Computing 43 (4) (1990) 401–410.
[31] CAIDA 2018, `https://www.caida.org/catalog/datasets/trace_stats/nyc-a/2018/equinix-nyc.dira.20180517-130000.utc.df.txt`, accessed: 05-2022.
[32] CAIDA 2019, `https://www.caida.org/catalog/datasets/trace_stats/nyc-a/2019/equinix-nyc.dira.20190117-130000.utc.df.txt`, accessed: 05-2022.