LETTER

LRU-LC: Fast Estimating Cardinality of Flows over Sliding Windows**

Jingsong SHAN†*, *Nonmember*, **Jianxin LUO**†, *Member*, **Guiqiang NI**†a), **Yinjin FU**†, *and* **Zhaofeng WU**†, *Nonmembers*

SUMMARY Estimating the cardinality of flows over sliding windows on high-speed links is still a challenging work under time and space constrains. To solve this problem, we present a novel data structure maintaining a summary of data and propose a constant-time update algorithm for fast evicting expired information. Moreover, a further memory-reducing schema is given at a cost of very little loss of accuracy.

key words: *streaming data, flow, sketch, cardinality, sliding window*

1. Introduction

A *flow* in computer networks is defined as a stream of packets which share the same properties (e.g, source/destination addresses). Estimating the *cardinality of flows*, the number of distinct flows, is a fundamental problem in many network applications, such as network measurement, intrusion detection, and clickstream analysis. For example, sudden changes in the cardinality of flows in links may indicate DDoS attacks, worm outbreaks or port-scan activities [1], [2].

In most real-world applications, data are time-sensitive, i.e., very "old" items are considered less useful and relevant than the recent. A commonly used approach for covering the W most recent items is *sliding window model*. The paper aims to address the following problem: estimating the cardinality of flows over *sliding windows* on high-speed links (e.g., OC-48 (2.5Gbps), OC-192 (10Gbps)).

Packets on backbone links are generated continuously at a high rate and in a form of *data streams*. The line-rate stream is of unbounded length; its quantity is too huge to fit in main memory. Besides, data items are dynamic in sliding window model, where stale items are discarded over time. Given memory and time constraints, algorithms for counting the cardinality of flows has to (1) maintain a sketch of the data involved, instead of the actual data, (2) consume as little update time as possible for evicting stale information, and (3) process data in only one pass.

In this paper, combining a bitmap sketch with the least-recently used (LRU) replacement policy [3], we propose a novel sketch working with sliding window model, and introduce a simple but efficient update mechanism, which only needs very small constant time in one time slot.

Because the linear counting (LC) sketch is a building block of our approach, we review it briefly.

1.1 LC Sketch

LC sketch [4] only retains a bit vector of size m, a concise sketch of the actual data, consuming substantively smaller memory. The technique is based on hashing. For each element e, a uniformly distributed hash function $h(x)$ is used to map e to a bit position j. The corresponding bit at position j is set to 1. After N iterations, each bit of the vector has one of the two states: *empty* (or zero) or *non-empty* (or one). By *bins and balls theory*, the cardinality estimate $\hat{n}$ of the data set is:

$$\hat{n} = -m \ln \frac{z}{m} \tag{1}$$

where m is the sketch size, z is the number of empty cells. The estimation accuracy increases with the size of the bitmap. Further details can be found in [4].

The LC sketch only needs a single-pass over each data element, running very fast. As has been demonstrated by [5], for LC sketch to attain comparable accuracy, it needs less memory space than other counting sketches.

1.2 Difficulties of Applying LC in Sliding Window Model

In most real-world applications, newly arrived data items are more important and relevant than old ones. This results in *sliding window model* [6]: each data item expires after exactly W time units. In this model, items are arriving continuously; meanwhile some are being forgotten due to their expiry.

In order to adapt LC sketch to sliding window settings, we have to (1) extend LC sketch to support the aging mechanism, and (2) design a state update approach for evicting outdated information. To fit massive flows, the extension of LC sketch is only allowed to consume little extra memory. Besides, to continuously track the cardinality in real time, the algorithm for updating LC sketch should guarantee a small constant time in each time step. Due to hashing, the time correlation between cells is completely destroyed. A key problem we should address is how to timely update

Manuscript received December 28, 2015.
Manuscript revised June 7, 2016.
Manuscript publicized June 29, 2016.

†The authors are with PLA University of Science and Technology, Nanjing, P.R. China.
*Presently, with Huaiyin Institute of Technology.

**This work was supported by the 863 Program (No. 2012AA01A510 and No. 2012AA01A509), the NSFC (No. 61402518) and the JPSFY (No. BK20150722).

a) E-mail: networkmanage707@gmail.com

DOI: 10.1587/transinf.2015EDL8263

the aging degree of each cell in such a temporally unordered structure without scanning all entries. In this paper, we aim to address problems mentioned above.

2. Our Solution: LRU-LC Sketch

In this paper, we propose an LRU-LC sketch which adapts LC to sliding window scenarios. The idea of LRU-LC comes partly from the well-known memory page replacement algorithm, LRU. LRU always evicts the least-recently-used page from an LRU queue which organizes the buffer pages ordered by time of their last reference. The salient merit of LRU is constant time complexity. Inspired by LRU, LRU-LC also arranges entries in time order by LRU queue so that evicting stale information only needs constant time.

We use the following notation in the rest of the paper. Each item $e \in S$, consisting of the five-tuple (dest/src ip, dest/src port, protocol) extracted from a packet, is mapped to an entry by a uniform hash function $h(e)$. Items in S emerge in a streaming manner: $\{e_{t_1}, e_{t_2}, \ldots e_{t_n} \ldots\}$, where the subscript t_i denotes the arrival time. The time axis is partitioned into small-equal-length time slots which are assumed to be so small that no two items arrive in one slot. Let Δt denote the inter-arrival time between two adjacent items. The sliding window size W is defined as the number of time slots. The problem of estimating the cardinality of flows over a window is to count the cardinality of S over the W.

2.1 Structure of LRU-LC Sketch

As shown in Fig. 1, the data structure for LRU-LC sketch is an array of length m, where each entry consists of three components: backward pointer (bp), time difference between two adjacent items (Δt) (will be further discussed later) and forward pointer (fp); each entry has two states: *active state* and *inactive state*. Their definitions are given as follows:

Definition 1 (Active State): Suppose the entry i is hit at time t_i, after that the entry i remains active state within a window of time. During this time, if no items are mapped to the entry, it will become inactive.

Two pointers, *pre* and *suc*, are introduced to facilitate the arrangement of active entries in LRU queue. Note that the index of each entry keeps invariant, unconditionally; LRU queue is built only by pointers. All entries are grouped into two clusters by their states: active or inactive. Entries of LRU queue are sorted in order of the last time they are hit.

In order to represent the aging degree, we first introduce the concept of time difference. The primary purpose of using time difference is to represent the aging degree of each active entry. A schematic illustration of time difference is shown in Fig. 2; its formal definition is as follows:

Definition 2 (Time Difference): Given two adjacent entries $V[i]$ and its direct predecessor $V[i].pre$ in the LRU queue, and t_i (t_{i-1}) is the last hit time of $V[i]$ ($V[i].suc$), the time difference Δt of $V[i]$ is defined as $t_i - t_{i-1}$.

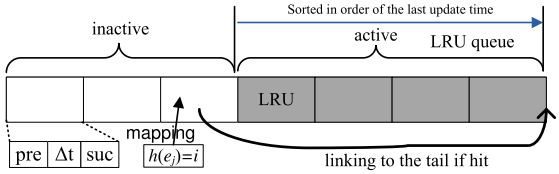

Fig. 1 Structure of LRU-LC

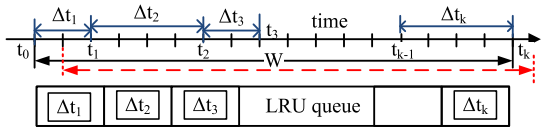

Fig. 2 Illustration of time difference schema

Given an entry $V[i]$ in LRU queue, its aging degree, negatively proportional to its distance from the start of window, is equal to the substraction of the sum of all its predecessors' Δts from W, formally denoted as:

$$D_{aging} = W - \sum_{V[j] \in S_p^i} \Delta t_j \tag{2}$$

where S_p^i denotes the set of all predecessors of $V[i]$. As illustrated in Fig. 2, each active entry records its time difference Δt; obviously, the left-most entry, the oldest entry, is the prodecessor of all entries of LRU queque. In order to evict inactive entries, aging degree of each entry must decay gradually over time (or as window moves forward) after being hit. By Formula (2), Δt_1 minus 1 means D_{aging}s of all entries in LRU queue plus 1, and indicates that the window moves one step forward (demonstrated by the red dotted line). By means of time difference schema, only few operations are needed to be perfomed on the left-most entry for discarding inactive entries from LRU queue.

2.2 Algorithm for Maintaining LRU-LC Sketch

The algorithm for maintaining LRU-LC sketch is presented in Algorithm 1. It has two objectives: building LRU queue and evicting expired entries from the queue.

Building process. Similar to LC, LRU-LC is also based on hashing. In each time slot i, if the item e_i arrives, it is mapped to the index $pos = h(e_i)$. If the entry $V[pos]$ is inactive, it is appended to the tail of the queue (line 5-8); otherwise it is moved to the tail of the queue for keeping the queue chronologically ordered (line 10-15). Note that once entry *pos i* is shifted to tail, its direct successor's Δt_{i+1} is assigned to $\Delta t_{i+1} + \Delta t_i$ for keeping aging degree unchanged. In the end, all active entries are organized into LRU queue and are sorted in order of the last time they are hit.

Evicting process. The process of evicting only involves the left-most entry because the aging degrees of entries are organized in strictly decreasing order from the left to right. The Δt of the left-most entry is decremented by 1 in each time step (i.e., the window makes one step forward). If the Δt is equal to zero, the entry has become inactive, and

Algorithm 1: Algorithm for building LRU-LC

```
Input: Streaming items S
Output: LRU-LC: V[m], Number of inactive entries: z
initialize V, z = m, Window Size W;
for each time slot i do
    if item e emerges in the slot i then
        pos = h(e);// map e to index pos
        δt = t_i − t_{i−1};// compute the time difference
        if V[pos] is inactive then      // not in LRU queue
            V[pos].Δt = δt;
            append V[pos] to LRU queue;
            decrement z by 1;
        else // V[pos] has existed in the LRU queue
            if V[pos] is the last entry then
                V[pos].Δt=V[pos].Δt + δt
            else
                alter the Δt of V[pos]'s successor;
                shift V[pos] to the tail of LRU queue;
            end
        end
    end
    decrement Δt of the left-most entry by 1;
    if the Δt==0 then
        evict the inactive entry;
        increment z by 1;
    end
end
```

then is removed from the LRU queue.

As stated in Sect. 1.1, estimating cardinality only involves two variables: z and m. The variable z is incrementally maintained by Algorithm 1 and m is user-specified parameter, thus it is easy to get cardinality estimates by Formula (1).

2.3 Algorithm Analysis

Time complexity. Our algorithm supports constant update time ($O(1)$ time) in each time step. Reasons are as follows. The building procedure only needs one hash mapping and few pointer operations. Besides, evicting procedure removes at most one entry from the list. Lastly, the procedure of estimating only needs to compute a simple formula (Formula (1)) in constant time.

Space complexity. LRU-LC needs an array of length m, each of which includes three parts: two pointers and one time difference Δt. Each pointer needs $log_2 m$ bits; Δt uses at most $log_2 W$ bits because its maximum is W. Therefore, LRU-LC uses at most $m(2log_2 m + log_2 W)$ memory space. Identically to the size of LC, the LRU-LC size m is determined by the preset accuracy we want to attain. The window size W is configured according to the needs of specific applications, and often far greater than m.

2.4 Reducing Memory Space

As mentioned in the last section, the memory requirement of LRU-LC is dominated by two quantities: sketch size m and the maximum of time difference Δt_{max}. The former can be considered as a fixed quantity. The latter is at most equal to W, far greater than m in most cases. If we can decrease the dominating quantity Δt_{max}, the memory requirement of LRU-LC would be reduced, significantly. We will introduce a simple but effective method to address the problem at a cost of very little accuracy loss. The key idea is to set a proper threshold Th (far less than W) for Δt_{max}, such that the range $[0, Th]$ covers the vast majority of Δts.

Our proposed method comes from a classic theory that the inter-arrival time between two neighboring packets e_{t_i}, $e_{t_{i+1}}$ follows a exponential distribution. The time difference Δt, is actually the inter-arrival time, has the same property. Therefore, we obtain the following formula:

$$Pr\{\Delta t > t\} = \exp(-\lambda t) \tag{3}$$

where λ is mean arrival rate per time unit, $t = n * T$ ($n = 0, 1, 2 \ldots$; T = a slot of time). If we plot a histogram of the Δt, it would be an exponentially decreasing function, which means the vast majority of Δts are concentrated in the range from 0 to a threshold Th ($Th \ll W$). Therefore we can find an appropriate threshold Th for Δt_{max}, such that the probability $Pr\{\Delta t \geq Th\}$, failure rate, is very small.

Applying Markov's Inequality to Formula (3), we obtain the upper bound of failure rate that $\Delta t > Th$:

$$Pr\{\Delta t \geq Th\} \leq \frac{E(\Delta t)}{Th} = \frac{\lambda}{Th}. \tag{4}$$

It follows that $Th = \frac{1}{\lambda} * Pr\{\Delta t \geq Th\}$. For a given failure rate (e.g., 0.01) we can obtain a threshold Th far less than W. In such a way, the memory consumption of field Δt will decreases remarkably.

3. Experiments

The evaluation of LRU-LC is performed on two trace data sets obtained from MAWI traffic repository of WIDE project [7] (DS1 for short, which consists of nearly 4×10^7 IP packet traces) and MOAT project of NLANR [8] (DS2, including over 1.7×10^7 IP packet traces), respectively. In experiments, we aim to demonstrate our analysis of LRU-LC in last section: LRU-LC uses $O(1)$ time in each step; setting a threshold Th for time difference Δt has very little effect on accuracy. In order to obtain the average values, 10 trials of experiments are carried out for each case. All experiments were performed on a PC with a 2.8 GHz Intel Core2 Duo E7400 CPU and 4 GB RAM.

Update time. We have argued, by analysis in Sect. 2.3, that LRU-LC supports constant update time in each time slot. Here we verify this result by experiments. Two quantities m and W may affect the time complexity in one time slot. We count the running time of the LRU-LC with different sizes m and with different window sizes W on two data sets respectively. Results, listed in Table 1, show that the running time for different sizes m (from 10^4 to 10^6 keeping $W = 10^7$) and for different window size W (from $2E + 6$ to $128E + 6$ keeping $m = 10^4$) keeps almost unchanged on

Table 1 Running time for different LRU-LC sizes m, and for different window sizes W on DS1, DS2

Time for different m (Unit: ms)			Time for different W (Unit: ms)		
m (entry)	on DS1	on DS2	W (slot)	on DS1	on DS2
10000	51707	178012	2E+6	51520	194140
50000	52118	179220	4E+6	50418	192818
100000	52443	183183	8E+6	50358	191614
200000	53146	186156	16E+6	50560	193472
400000	53895	189465	32E+6	49919	191510
800000	55405	191726	64E+6	48811	189584
1000000	55160	193449	128E+6	46717	185345

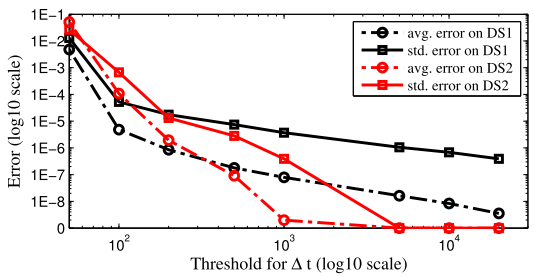

Fig. 3 Estimate errors for varying time difference threshold Th

each data set. This means that the running time of LRU-LC is independent of the two parameters. Therefore, these results confirm that our proposed algorithm uses constant time in each time step. Besides, by the data set used and running time consumed, we can approximately figure out the throughout of LRU-LC, about 20Gbps. It can work in backbone lines with bandwidth up to OC-192, even higher.

Accuracy analysis. The approximation errors of LRU-LC have two sources. One is the estimate error of LC algorithm; another is the threshold Th for Δt. Details of the analysis of the former can be found in [4]. This section focuses only on the latter. Two commonly used metrics, average error and standard error, are exploited to measure the deviation caused by the time difference threshold Th. Keeping $W = 10^7$ and $m = 10^4$, we vary the threshold Th from 50 to 10^5. Figure 3 plots the average error and standard error for different Th (50, 100, 200, 500, 1000, 5000, 10000) on two data sets. It can be seen that LRU-LC has an average error of 0.48% (5%) with standard error 0.013 (0.029) on DS1 (DS2) for $Th = 50$. The two types of errors decrease exponentially with the increase of Th. When $Th = 100$ LRU-LC only yields the maximum average error of less than 10^{-3}. In the cases of $Th > 200$, the inaccuracy caused by Th is very small. In terms of memory space, the field Δt needs only 8-bit memory for $Th = 200$. Without such approximation of Δt, LRU-LC has to provide 28-bit memory for Δt.

Comparison. There exit two methods adapting LC to sliding window model: timestamp vector (TSV) [9] and cut-down vector (CDV) [10]. TSV replaces the bitmap of LC with a 64-bit-timestamp vector. For a query, all entries of TSV must be probed and checked whether their timestamps have exceed the window boundary, a time-comsuming process, although it has no loss of accuracy compaired with LC. CDV uses a vector of small counters where the max value is C, and updates k entries sequentially in each time step. However, the parameter k is determined only by experience. For a small k, it will incur much loss of accuracy. For a big k, it will cost more update time and more memory space. Comparison of LRU-LC with the two aforementioned methods in terms of time and space are shown in Table 2. Shomura *et al.* [11] design a counting method for detecting anomalies. Shomura's study and LRU-LC differ in two aspects. Firstly, unlike Shomura's approach working in the whole data set, LRU-LC is a sliding-window-based technique counting in dynamic settings. Secondly, LRU-LC is based on a probabilistic data structure (bitmap sketch), but Shomura's method operates on a hash table.

Table 2 Performence comparison for three mehods

Method	Update time	Space
LRU-LC	$O(1)$	$m(2log_2m + log_2Th)$
TSV	$O(m)$	$64 * m$
CDV	$O(k)$	$O(m * log_2C)$

4. Conclusions

In this paper, we propose an LRU-LC sketch for fast estimating cardinality of flows over sliding windows on high-speed links. A fast algorithm is given for maintaining the sketch, which needs only constant time in each time slot, independent of its size and window size. Additionally, a time difference schema is provided to reduce the memory requirement of LRU-LC. The schema proposed in the paper can also be applied to other bitmap sketches [1].

References

[1] 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, 2006.
[2] W. Chen, Y. Liu, and Y. Guan, "Cardinality change-based early detection of large-scale cyber-attacks," Proc. of the IEEE INFOCOM, pp.1788–1796, 2013.
[3] E.J. O'Neil, P.E. O'Neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," Proc. ACM SIGMOD, pp.297–306, 1993.
[4] 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, 1990.
[5] A. Metwally, D. Agrawal, and A.E. Abbadi, "Why go logarithmic if we can go linear?: Towards effective distinct counting of search traffic," Proc. of 11th EDBT, pp.618–629, 2008.
[6] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," SIAM J. Comput., vol.31, no.6, pp.1794–1813, 2002.
[7] "MAWI Traffic Archive." available: http://mawi.wide.ad.jp/mawi/
[8] "Waikato project." available: http://wand.net.nz/wits/index.php
[9] H.-A. Kim and D.R. O'Hallaron, "Counting network flows in real time," Proc. of GLOBECOM, pp.3888–3893, 2003.
[10] J. Sanjuàs-Cuxart, P. Barlet-Ros, and J. Solé-Pareta, "Counting flows over sliding windows in high speed networks," Proc. of NETWORKING, pp.79–91, 2009.
[11] Y. Shomura, Y. Watanabe, and K. Yoshida, "Analyzing the number of varieties in frequently found flows," IEICE Trans. Commun., vol.E91-B, no.6, pp.1896–1905, 2008.