Fast Updates for Line-Rate HyperLogLog-Based Cardinality Estimation

Pedro Reviriego, *Senior Member, IEEE*, Valerio Bruschi, Salvatore Pontarelli, Daniel Ting, and Giuseppe Bianchi

***Abstract*— In a network it is interesting to know the different number of flows that traverse a switch or link or the number of connections coming from a specific sub-network. This is generally known as cardinality estimation or count distinct. The HyperLogLog (HLL) algorithm is widely used to estimate cardinality with a small memory footprint and simple per packet operations. However, with current line rates approaching a Terabit per second and switches handling many Terabits per second, even implementing HLL is challenging. This is mostly due to a bottleneck in accessing the memory as a random position has to be accessed for each packet. In this letter, we present and evaluate Fast Update HLL (FU-HLL), a scheme that eliminates the need to access the memory for most packets. Results show that FU-HLL can indeed significantly reduce the number of memory accesses when the cardinality is much larger than the number of registers used in HLL as it is commonly the case in practical settings.**

***Index Terms*— Network monitoring, high speed networks, cardinality, hyperloglog.**

I. Introduction

COUNT distinct or cardinality estimation is an important task in several networking scenarios. Count distinct algorithms are in fact used to gather information about the number of active flows on a high speed link [1]. Another compelling application scenario is that of network security monitoring, where, for instance, cardinality estimation techniques are used to infer the presence of massive network scanners [2], [3]. Detecting network nodes that have a large number of connections is also of interest and again done with cardinality estimation [4], [5].

Since the exact count of distinct elements in a large multi-set is cumbersome and resource consuming, over the years many different *approximate* cardinality estimation algorithms have been proposed [6]. Among them, the HyperLogLog (HLL) algorithm [7] has established itself as *state of the art*, since it provides accurate estimates over a large range of cardinalities using a small memory footprint [8].

While several works have attempted to further reduce HLL's memory footprint, to the best of our knowledge no previous work has addressed a potential HLL bottleneck in terms of memory *access frequency*. Indeed, to implement HyperLogLog in a link or network device, each packet must be parsed and then used to update the data stored in the HyperLogLog. For that, one register in a large array, typically stored in a memory, is accessed to see if its value needs to be changed. This poses a challenge for high-speed links or network nodes. For instance a 400Gb/s link fed by minimum size 512 bits packets would roughly require one memory access every nanosecond thus ruling away the possibility to employ cheap DRAMs [9], [10]. Two further orders of magnitude would be mandated to manage an HLL implementation over a today' state of the art switching fabric (64 ports at 400 Gb/s each) therefore also challenging SRAM technologies.

Such a *memory access bottleneck* may be addressed via two opposite strategies. The first one, which we do not take in this letter, consists in carefully organizing memory into parallel blocks, and manage the relevant access procedures. Of course the switch must be equipped with high performance SRAMs, and it is worth to notice that, while the network bandwidth is increasing at a significant pace, the increment of on-chip SRAM throughput is much less. Rather, our work is motivated by the following observation. It is true that HLL requires in principle one memory access per packet, but in practice the overwhelming majority of such accesses end up in discovering that the just read register does *not* require *any* update!

The approach presented in this letter, called Fast Update HLL (FU-HLL), reduces the number of memory accesses needed to implement HyperLogLog. The baseline idea is very simple: access the register bank only when an actual update (i.e. a write operation) is mandated. This can be accomplished by storing the minimum value for all the registers in the HyperLogLog. In practice, a few non trivial caveats need to be addressed, as detailed later on in the presentation of our specific construction. The theoretical analysis and simulation results show that a significant fraction of the memory accesses can be avoided, and that the proposed scheme is very effective when the cardinality is much larger than the number of registers used in the HyperLogLog (as is commonly the case in practical settings).

The rest of the letter is organized as follows. The HyperLogLog algorithm is briefly presented in Section II. Section III presents the proposed Fast Update HLL scheme and analyzes it theoretically. The proposed method is evaluated by simulation in Section IV to illustrate its potential benefits. Finally the letter ends in Section V with the Conclusions.

II. HyperLogLog

As discussed in the introduction, HyperLogLog (HLL) is a widely used algorithm for cardinality estimation [7]. Before

Manuscript received July 16, 2020; revised August 15, 2020; accepted August 15, 2020. Date of publication August 20, 2020; date of current version December 10, 2020. This work was supported by the ACHILLES Project (PID2019-104207RB-I00) and the Go2Edge Network (RED2018-102585-T) funded by the Spanish Ministry of Science and Innovation and by the Madrid Community Research Project TAPIR-CM (P2018/TCS-4496). The associate editor coordinating the review of this letter and approving it for publication was B. Dezfouli. *(Corresponding author: Pedro Reviriego.)*

Pedro Reviriego is with the Departamento de Ingeniería Telemática, Universidad Carlos III de Madrid, 28911 Madrid, Spain (e-mail: revirieg@it.uc3m.es).

Valerio Bruschi, Salvatore Pontarelli, and Giuseppe Bianchi are with the Department of Electronic Engineering, University of Rome Tor Vergata, 00133 Rome, Italy (e-mail: valerio.bruschi@uniroma2.it; salvatore.pontarelli@uniroma2.it; giuseppe.bianchi@uniroma2.it).

Daniel Ting is with Tableau Software, Seattle, WA 98103 USA (e-mail: dting@tableau.com).

Digital Object Identifier 10.1109/LCOMM.2020.3018336

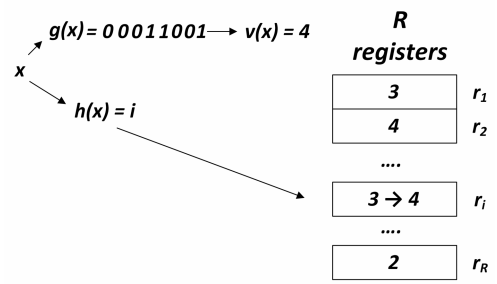

Fig. 1. Example of a counter update in HLL.

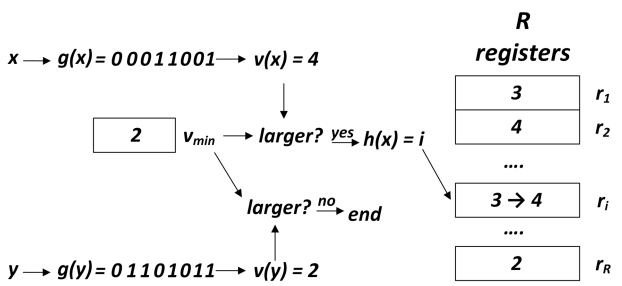

Fig. 2. Example of counter updates in FU-HLL.

presenting HLL in more detail, let us define the notation that we will use in the rest of the letter:

- R: number of registers
- r_i: value of the i^{th} register
- $h(x)$: hash function to select a register for element x
- $g(x)$: hash function to generate the value for element x
- $v(x)$: value generated from the leading zero count of $g(x)$
- C: true cardinality
- C_{HLL}: HLL cardinality estimate
- v_{min}: minimum value of the HLL registers

An HLL sketch is formed by an array of R registers each having only a few bits. To update the sketch with an element x, two hash functions $h(x)$ and $g(x)$ are computed. The first chooses the register $r_{h(x)}$ to update. The second generates a potential update value $v(x)$ equal to one plus the number of leading zeros in the bit representation of $g(x)$. The new value of the register is the maximum of the old value and $v(x)$. An example of an element update is shown in Figure 1 in which element x is inserted by first selecting the register with $h(x)$. Then $v(x)$ is computed based on $g(x)$ and a value of 4 is obtained that is larger than the value stored in the register. Therefore the value of the register is updated to the new value of 4.

After updating the HLL sketch with the procedure described, the cardinality can be estimated using the contents of the registers as follows. First the harmonic mean of the register contents is computed as follows:

$$Z = \frac{1}{\sum_{i=1}^{R} 2^{-r_i}}, \tag{1}$$

where r_i is the value of the i^{th} register.

Then, the cardinality estimate is obtained by multiplying the result by a constant factor α_R that depends on the number of registers R:

$$C_{HLL} = \alpha_R \cdot R^2 \cdot Z \tag{2}$$

The error of the HLL estimate for large cardinalities is approximately $\frac{1.04 \cdot C}{\sqrt{R}}$ where C is the true cardinality. To obtain a given accuracy, the number of registers R can be selected based on that equation. For example, when $R = 1024$, the expected error is approximately 3%. Small cardinalities require a modified estimator, where the best known estimators are given in [11], [12].

III. Fast Updates in HyperLogLog

When HLL is used to estimate cardinality in high speed networks, one HLL update operation, i.e., one memory access to the HLL array of registers, is needed per packet. However, in some HLL implementations, the minimum value of the HLL registers v_{min} is kept so that registers can store only the increment over that minimum and thus registers can use fewer bits [13]. Interestingly, this minimum value can also be used to reduce the number of memory accesses needed for updates by storing v_{min} on a dedicated register in a hardware implementation. In particular, when an element x arrives and we compute $g(x)$ and $v(x)$, we only need to access the register when $v(x) > v_{min}$. This on average would occur for $\frac{1}{2^{v_{min}}}$ of the elements, therefore when the minimum value is larger than zero, a significant number of memory accesses can be saved. A side benefit of this scheme is that the hash computation for $h(x)$ can also be saved. This is the main idea behind the proposed Fast Updates (FUs) scheme that is illustrated in Figure 2 which shows the update operations for two elements x and y. In the case of x, $v(x) = 4$ that is larger than v_{min} so we need to access the counter given by $h(x)$ and perform the update operation as in the traditional HLL. Instead, for y, $v(y) = 2$ and since it is not larger than v_{min}, the operation ends after comparing with v_{min} and no memory access is needed.

It must be taken into account that computing v_{min} also requires some memory accesses. To minimize this number, we can have a register c_{next} that stores the number of HLL registers that have a value of $v_{min}+1$ or larger. Then, when we increment a register to a value $v_{min}+1$ or larger, we increase that register and when it reaches R, we increase v_{min} and reset the counter c_{next} and read all the R registers to initialize it for the new value of v_{min}. This is simple and would require at most $v_{min} \cdot R$ memory accesses. That should be negligible in most cases compared to the savings. It is important to note that the reading of all the registers to initialize c_{next} can be done in parallel with updates for new packets using spare access cycles or using a small fraction of the available memory bandwidth. This would in the worst case only slightly delay the next update for v_{min} while allowing packet processing to continue during the initialization of c_{next}.[1]

The detailed flowchart for the proposed Fast Update operation for an element x is shown in Figure 3. The steps that require a memory access are shown with a grey background and the operations that are not always required and can

[1] A careful reader can point out that during this update window an erroneous double increment of c_{next} can occur. Suppose that the update procedure already read the registers between 0 and i, and a new packet updates a register j. With $j > i$ we must avoid to increment c_{next} since it will be incremented by the procedure. If $j \leq i$ we will increment c_{next} as usual.

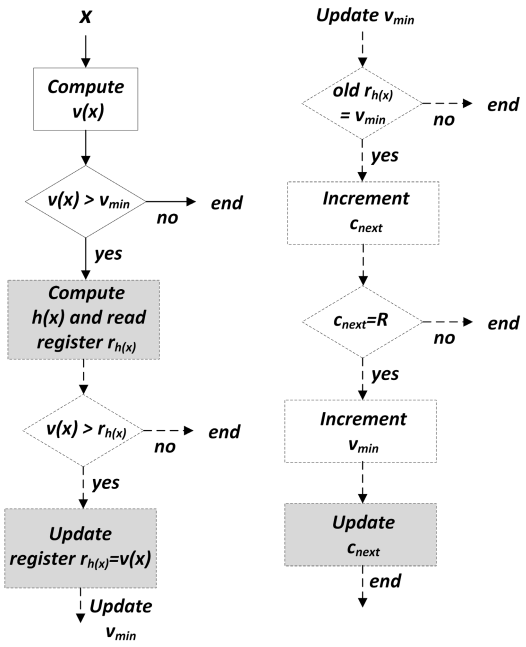

Fig. 3. Flow chart for the proposed Fast Update (FU) operation.

be saved are marked with dashed lines. The first step is to compute $g(x)$ and subsequently obtain $v(x)$. Then $v(x)$ is compared to v_{min} and only when is larger the update continues, otherwise it ends with no memory access and without computing $h(x)$. Instead, when $v(x) > v_{min}$, $h(x)$ is computed and the register $r_{h(x)}$ is read. Then, only when $v(x) > r_{h(x)}$ the process continues by updating the value of the register to $r_{h(x)} = v(x)$. The last steps of the algorithm are needed to keep a consistent value of v_{min}. To do so, if the initial (old) value of $r_{h(x)}$ was equal to v_{min}, then c_{next} is incremented and compared to R. When they are equal, it means that all registers now are larger than v_{min} and thus v_{min} can be incremented. Therefore, v_{min} is updated to $v_{min} + 1$ and finally, c_{next} has to be updated to reflect the number of registers that already store values equal to or larger than the new v_{min}. This would require R memory accesses but this step would be rarely executed and can be done while processing packets as discussed before. It is important to note that although the update procedure in FU-HLL is more complex than in HLL, in many cases it ends after the comparison of $v(x)$ with v_{min} saving not only the memory access but also the computation of $h(x)$ and thus requiring less computational effort.

The savings obtained by using this fast update procedure depend on the value of v_{min} that in turn depends on the cardinality of the set C that has already updated the HLL. A relationship between the two can be obtained by using results from the classical coupon's collector problem [14]. Each register is associated with a coupon. For any fixed value v, we say register r_i's coupon is collected if it has value at least v. Since an item x has value $v(x)$ equal to the number of leading zeros in a uniform hash $g(x)$ plus one, the probability it has at least a given value V is 2^{-V+1}. The coupon collector problem states that the cardinality T_V at which v_{min} is incremented to V has expectation:

$$\mathbb{E}T_V = 2^{V-1} \cdot R \cdot H_R = 2^{V-1} \cdot R \cdot (\log(R) + O(1)) \quad (3)$$

where $H_n = \sum_{i=1}^{n} 1/i$ is the n^{th} harmonic number.

An estimate of the fraction of fast updates F that access the memory when the HLL is built from scratch and filled with C different elements can be obtained by:

$$F = \frac{1}{C} \cdot \left(\sum_{i=1}^{i_{max}} \frac{(\mathbb{E}T_i - \mathbb{E}T_{i-1})}{2^{i-1}} + \frac{(C - \mathbb{E}T_{i_{max}})}{2^{i_{max}}} \right) \quad (4)$$

where $T_0 = 0$ and i_{max} is the largest index i such that $\mathbb{E}T_i \leq C$ and is thus a deterministic approximation of v_{min}.

Finally, it is interesting to note that it may be possible to further increase the savings in the number of memory accesses by keeping the minimum value for different parts of the HLL registers. For example, keep v_{min_1} for the first half of the registers and v_{min_2} for the second. This would tend to produce larger v_{min} values and thus larger savings at the cost of additional variables and a slightly more complex update procedure. This optimization is not considered in the rest of the letter and left for future work.

IV. Evaluation

This section presents the evaluation results obtained by simulation and compares them with the theoretical estimates. First the test setup and procedure are described and then the experimental results are presented and discussed. The section ends with a discussion of the potential benefits of FU-HLL when used in a high speed switch.

A. Test Setup

We implement the proposed approach in a software simulator written in python that processes packets from a trace file. The simulator processes packet by packet in order as they are found in the file. It extracts the 5-tuple from the packet and implements the algorithm described above. The HLL operations are implemented by using the xxHash [15]. This is an extremely fast hash algorithm, that can run at RAM speeds and it is typically used in non-cryptographic software implementations.

In order to evaluate the benefits of the proposed solution, we prepared two different workloads. To provide conservative results, the first workload is a synthetic workload in which each stream contains no duplicated elements. The 5-tuples are generated by selecting randomly the source and destination ports while the source and destination IPs are incremented sequentially to ensure that there are no duplicates. To test the independence of the algorithm to the order of items in the stream, we also generate five copies of each stream and permute the order of items in each.

We also use traffic traces obtained from CAIDA.[2] In this second workload elements can appear many times as a flow can have many packets (i.e. the Chicago trace has in average 35 copies of the same item, while the NYC trace about 14 copies). The traces are 60 seconds capture and contain about 24M-37M packets, while the stream is composed of about 700K-2.6M distinct elements. Therefore, we expand the real workload set by splitting the CAIDA traces according to four different time windows (i.e. $\{5, 15, 30, 60\}$ seconds). In this way, we can test the proposed solution on different time

[2]We take one trace from 2015 and another from 2019 [16]: equinix-chicago.dirB.20150917-135800.UTC.anon.pcap and equinix-nyc.dirA.20190117-132400.UTC.anon.pcap.

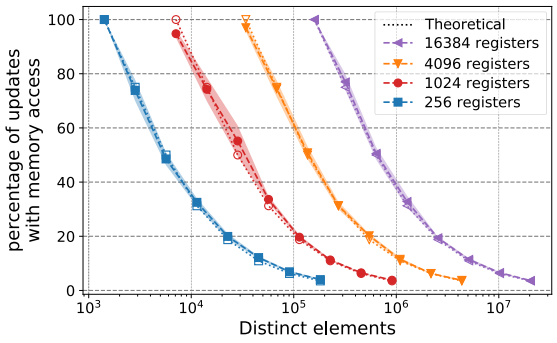

Fig. 4. Percentage of Fast Updates that need to access the memory both simulated and estimated by Eq. 4.

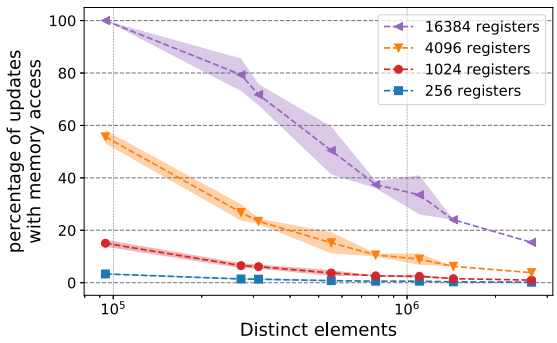

Fig. 5. Percentage of Fast Updates that need to access the memory for the CAIDA traces.

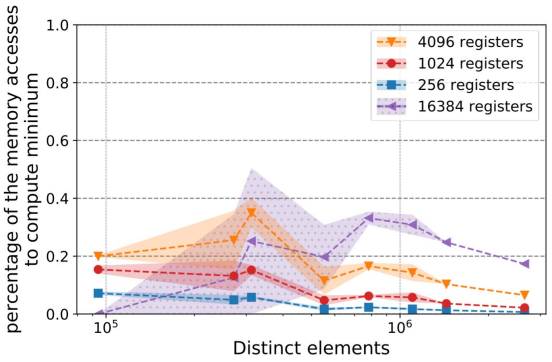

Fig. 6. Ratio (in percentage) of the memory accesses to compute v_{min} to the number of updates done for the CAIDA traces.

windows that contain different number of distinct elements (ranging from 90K to 2.6M). We expect an improvement in the results as the proposed scheme can benefit from these repetitions as once a large number of elements have been added to the HLL, subsequent packets will rarely need to access the memory.

Finally, tests were conducted on a 12 physical core server (running Intel Xeon Silver @2.10 GHz without the Intel TurboBoost feature) with 128 GB RAM memory.

B. Evaluation

We conduct several experiments in order to evaluate the benefits of the proposed FU-HLL scheme. In particular, the percentage of update operations that need to check an HLL register and thus access the memory. The experiments are done for several values of the number of HLL registers R and cardinalities C to illustrate for each value of R the cardinality values for which FU-HLL is effective. As discussed before, the benefits of FU-HLL are expected to be relevant when C is significantly larger than R.

Figure 4, shows the percentage of fast updates that need to access the HLL registers for synthetic sets of different cardinalities. The colored area around the main line shows the standard deviation of the different trials done for each configuration. It can be seen that the variability is low. The theoretical estimates obtained with Eq. 4 are also shown in the Figure and match reasonably well those obtained by simulation. As expected, the benefits of the proposed solution are significant when there are many more unique elements C in the stream than registers R in the HLL as it is normally the case in practical settings. For example, an HLL with $R = 1024$ estimates cardinalities with a error of approximately 3% and is thus sufficient for many applications. The results show that when $R = 1024$, the percentage of updates that require a memory access goes below 50% for a cardinality of only 30,000 showing the potential benefits of the proposed scheme.

Figure 5, shows the percentage of fast updates that need to access the HLL registers for the CAIDA traces considered. The colored area around the main line shows the standard deviation of the different trials that correspond to the same time window. In this case, there is more variability due to the presence of duplicates. But, it can be observed that the results are inline with the previous experiment with synthetic workload. From a practical point of view, the benefits for high speed traffic links are clearly shown as the number of updates that require a memory access is below 20% for all the traces and time windows when $R = 1024$ or smaller.

Finally, it is worth checking that the additional memory accesses needed to compute v_{min} are negligible compared to the savings obtained in the updates by using the proposed scheme. Figure 6 shows the ratio of the memory accesses to compute v_{min} to the number of updates done for the CAIDA traces. It can be seen that the ratio is below 0.5% in all cases and thus negligible. In fact, the value is smaller for lower values of R so that for practical settings, the overhead is even smaller. Therefore, the savings in memory accesses for updates are much larger than the accesses needed to maintain v_{min} and thus overall, FU-HLL significantly reduces the number of memory accesses when the cardinality is significantly larger than the number of counters R.

C. Discussion

To illustrate the potential benefits of FU-HLL, let us consider a state of the art switch ASIC with 64 ports at 400 Gb/s for which internal SRAMs have an access time in the order of 1 ns. To estimate for example the number of flows that traverse the switch we would need in the worst case (512 bit packets) to perform 50 memory accesses per nanosecond. Even considering an average packet length of 5000 bits, 5 memory accesses per nanosecond would be needed. This can only be done by using several memories in parallel even for the average case. Instead, with FU-HLL, if the fraction of the updates that needs a memory access is below 20%, the average packet length can be supported with a single memory. This is

the case when 1024 HLL registers are used for all the CAIDA trace experiments as shown in Figure 5 that correspond to a single link. Therefore, it can be expected than in many cases, FU-HLL can reduce the need for parallel memories in switches to support the update rates.

V. Conclusion and Future Work

This letter has presented Fast Updates HLL (FU-HLL) a technique to reduce the number of memory accesses needed for HyperLogLog based cardinality estimation. The proposed scheme keeps the minimum value of the HyperLogLog registers and uses it to determine when an update operation needs to check the corresponding HyperLogLog register. The theoretical analysis and the simulation results show that FU-HLL significantly reduces the number of memory accesses when the cardinality is much larger than the number of HyperLogLog registers as it is commonly the case in practical settings. Therefore, it can be used to reduce the memory bandwidth needed to implement HyperLogLog that can be a bottleneck when used on high-speed links or switches.

The use of different minimum values for blocks of the HLL registers seems an interesting idea to further reduce the memory accesses that can be explored as future work. The evaluation of the benefits of the proposed scheme for other applications not related to networking can also be of interest. Another area for future work is the combination of the proposed FU-HLL scheme with techniques that store the minimum value of the counters to reduce the number of counter bits and thus the memory footprint as they can complement each other. Finally, the implementation of FU-HLL in switch ASICs would also be interesting.

Appendix

A. Expanded Proof for Cardinality at Update Time for v_{min}

This provides a more detailed proof for equation 3. Suppose there are R coupons and a stream of coupons are randomly selected with replacement. The classical coupon collector problem gives the expected total number of coupons collected before all distinct coupons are found. Denote the total number of coupons needed to obtain k distinct coupons as D_k. When k distinct coupons have been found, the probability of finding a new distinct coupon is $(R-k)/R$. Thus, the expected number of additional coupons is $\mathbb{E}(D_{k+1}-D_k) = R/(R-k)$. The total number of coupons D_m needed to find all distinct coupons is thus

$$\mathbb{E}D_R = \sum_{k=0}^{R-1} \mathbb{E}(D_{k+1} - D_k) = \sum_{k=0}^{R-1} R/(R-k) = \sum_{k=0}^{R-1} R/(k+1) = R(\log R + O(1)) \quad (5)$$

To apply this to HLL, each coupon corresponds to a register in the sketch. An item in the stream corresponds to coupon k if it is hashed to bin k and its value is greater than v_{min}. An item's value is greater than v_{min} with probability $2^{-v_{min}}$. A stream of items can be converted into coupons as follows. Take the substream of distinct items. Downsample the stream by randomly choosing items with probability $2^{-v_{min}}$. Thus, the number of items per coupon is $2^{v_{min}}$ in expectation, and the numbers needed for each coupon are independent. In this downsampled substream, the coupon collector problem gives that $R(\log R + O(1))$ coupons are needed before all bins have value $> v_{min}$ and v_{min} should be incremented. Each coupon corresponds to $2^{v_{min}}$ items in expectation. Thus, the total number of distinct items encountered before v_{min} is incremented is in expectation

$$2^{v_{min}} R(\log R + O(1)) \quad (6)$$

B. Expanded Derivation for Number of Memory Accesses

This provides a more detailed exposition for equation 4. This equation gives the expected proportion of distinct items which require a memory access beyond a comparison to v_{min}. An item requires a memory access when it is hashed to a value $> v_{min}$. For a new distinct item, this happens with probability $1/2^{i-1}$ when $v_{min} = i-1$. For the period when $v_{min} = i-1$, there are $\min\{C, T_i\} - T_{i-1}$ new, distinct items, so the expected number of times that a new item generates a memory access in that period is $\mathbb{E}(T_i - T_{i-1})/2^{i-1}$ if $T_i \leq C$ and $(C - \mathbb{E}T_{i-1})$ otherwise. Summing over the values for $i-1$ gives the desired approximation.

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, Oct. 2006.

[2] W. Chen, Y. Liu, and Y. Guan, "Cardinality change-based early detection of large-scale cyber-attacks," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 1788–1796.

[3] Y. Chabchoub, R. Chiky, and B. Dogan, "How can sliding HyperLogLog and EWMA detect port scan attacks in IP traffic?" *EURASIP J. Inf. Secur.*, vol. 2014, no. 1, pp. 1–11, Dec. 2014.

[4] L. Zheng, D. Liu, W. Liu, Z. Liu, Z. Li, and T. Wu, "A data streaming algorithm for detection of superpoints with small memory consumption," *IEEE Commun. Lett.*, vol. 21, no. 5, pp. 1067–1070, May 2017.

[5] Y. Liu, W. Chen, and Y. Guan, "Identifying high-cardinality hosts from network-wide traffic measurements," *IEEE Trans. Dependable Secure Comput.*, vol. 13, no. 5, pp. 547–558, Sep. 2016.

[6] D. M. Kane, J. Nelson, and D. P. Woodruff, "An optimal algorithm for the distinct elements problem," in *Proc. 29th ACM SIGMOD-SIGACT-SIGART Symp. Princ. Database Syst. Data (PODS)*, 2010, pp. 41–52.

[7] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm," in *Proc. Int. Conf. Anal. Algorithms (AOFA)*, 2007, pp. 1–21.

[8] 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.

[9] Y. Kanizo, D. Hay, and I. Keslassy, "Maximizing the throughput of hash tables in network devices with combined SRAM/DRAM memory," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 3, pp. 796–809, Mar. 2015.

[10] J. C. Mogul and P. Congdon, "Hey, you darned counters!: Get off my ASIC!" in *Proc. 1st Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2012, pp. 25–30.

[11] O. Ertl, "New cardinality estimation algorithms for HyperLogLog sketches," 2017, *arXiv:1702.01284*. [Online]. Available: http://arxiv.org/abs/1702.01284

[12] D. Ting, "Approximate distinct counts for billions of datasets," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2019, pp. 69–86.

[13] Q. Xiao, Y. Zhou, and S. Chen, "Better with fewer bits: Improving the performance of cardinality estimation of large data streams," in *Proc. IEEE INFOCOM-IEEE Conf. Comput. Commun.*, May 2017, pp. 1–9.

[14] M. Ferrante and M. Saltalamacchia, "The coupon collector's problem," *Mater. Math.*, vol. 2014, no. 2, pp. 1–35, 2014.

[15] Y. Collet. (2016). *xxHash: Extremely Fast Hash Algorithm*. [Online]. Available: https://github.com/Cyan4973/xxHash

[16] *The CAIDA UCSD Anonymized Internet Traces*. Accessed: Feb. 19, 2020. [Online]. Available: https://www.caida.org