Sliding HyperLogLog: Estimating cardinality in a data stream

Noam Musba

With the advisor Prof. Eran Tavor

Introduction:

In this article I will present the problem of counting the cardinality of a big number of unique elements, and how it occurs in real life. I will present a few naïve solutions and then dive into the best solution, namely the HyperLogLog algorithm, and its improvement Sliding HyperLogLog algorithm which allows us to count this cardinality in the last window of time the user wants, allowing the user to save only the near past data and thus helping even further against attacks such as DoS.

Then I will present an implementation of this algorithm, from top to bottom, and finally show an analysis of the code’s performance, and how each parameter affects the outcome or how it affects other parameters.

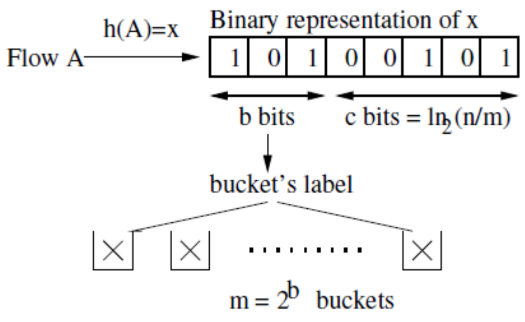
<u>Background:</u>

i. **<u>QUIC</u>**:
QUIC is a general-purpose transport layer network protocol initially designed by Jim Roskind at Google, implemented and deployed in 2012 and announced publicly in 2013 as experimentation broadened. QUIC is used by more than half of all connections from the Chrome web browser to Google's servers, as well as Microsoft Edge, Firefox and Safari web browsers.
QUIC aims to be nearly equivalent to a TCP connection but with much-reduced latency, thanks to the changes below, and thus **replacing the TCP connection** in the future.
<u>The first change</u> is to greatly reduce overhead during connection setup, by including more information - setup keys and supported protocols - in the initial handshake process.
<u>The second change</u> is to use UDP as its basis, which on the one hand does not include loss recovery, but on the other hand, QUIC implements it at its level instead of the UDP level. This means that if an error occurs in one stream, the protocol stack can continue servicing other streams independently, thus improving performance.
<u>Another change</u> is improving performance during network-switch events, like when a user of a mobile device moves from a local Wi-Fi hotspot to a mobile network. To avoid each connection timing out one-by-one and re-establishing them on demand, QUIC includes a connection identifier ("cid") which uniquely identifies the connection to the server regardless of source. This allows the connection to be re-established simply by sending a packet, which always contains this ID, as the original connection ID will still be valid even if the user's IP address changes.
<u>More changes</u> include encrypting the packets individually, the ability to implement QUIC in the user space and in the future even including forward error correction (FEC) to further improve performance when errors are expected.

ii. **<u>Sliding HyperLogLog algorithm</u>**:
The objective of the HyperLogLog algorithm is to estimate the cardinality of a given multiset's (a set where an element can be repeated) distinct elements. This can be used to estimate the number of distinct elements in a massive data stream, like in QUIC.
The motivation behind such an algorithm can be counting the number of distinct users in a website, early detection of DoS attacks, and more.
If we will try to solve this problem naively, namely a big array where each entry saves an element, while checking for repetitions before adding a new element- the complexity of time and space would be polynomial to input's size, which can be very costly, as we are talking about 40gigabyte a second.

A better but still not good enough answer would be to save it in a hash function, thus the complexity of the time for adding an element would be order of 1, but still the space complexity would be linear to input's size.
This algorithm is built on 3 improvements- the loglog, the hyper and the sliding improvements.
LogLog- First, it is important to understand that it is a probabilistic algorithm, meaning we will not get the accurate answer but a 'close enough' answer.

- The first phase of the LogLog algorithm is to send each element to a hash function, which will return a binary number x. The function has some conditions so the algorithm will work as expected:
  First, it will uniformly distribute a random number, meaning that the chance of getting a 0 or a 1 is equivalent.
  Second, if the same element is sent to this function again- the same output x will be distributed, thus helping the algorithm to count each distinct element once.
  Finally, the function will work independently on the elements.
- Next, there are m buckets, initialized to 0. Each bucket saves the largest number of the leftmost 1-bit among all the hashed elements sent to it. To decide what bucket the hashed element belongs to, the first b bits of the hashed element determine the id of the bucket, thus b and m upholds: $b = \log_2 m$.



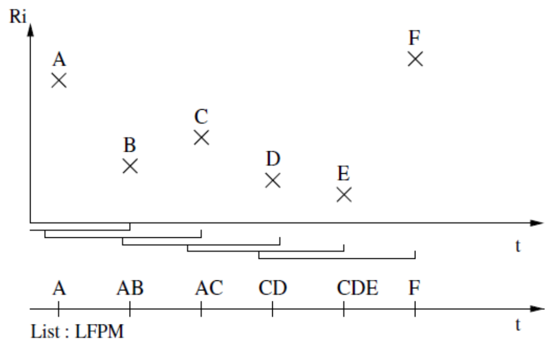
- After determining to which bucket the hashed element belongs to, the algorithm counts the position of the leftmost 1-bit of the remaining number: $c = (x - b)$, and saves it in his bucket iff it is larger than the existing number saved in this bucket.
- Finally, a mean is taken. The LogLog algorithm takes the arithmetic mean of all the buckets, multiplied by a constant $\alpha(m)$ and by m^2 which is the number of buckets to the 2nd power. This mean is not accurate enough and that is where the Hyper improvement comes in:

HyperLogLog- Instead of taking the arithmetic mean of all the buckets, a harmonic mean is taken, thus improving the algorithm significantly. A harmonic mean tends strongly toward the least values of leftmost 1-bit. The rest of the algorithm stays the same.

<u>Sliding HyperLogLog-</u> The above algorithm answers the question- “how many distinct elements were counted up until now?”. The Sliding algorithm wants to change it to the question- “how many distinct elements have been seen over the last w unit of time?”. This improvement of the above algorithm aims to maintain a short list of packets. A packet consists of a pair $< t_i, R_i >$, where the t_i is the arrival time of the packet, and R_i is the position of the leftmost 1-bit in the binary representation of the hashed element associated to this packet. The algorithm will store only packets that are useful to the computation of the cardinality, meaning a packet will stay stored in this list if it is a possible maximum over a future window of time, thus the name of this list is List of Future Possible Maxima (LFPM). It is updated in the following way:

```
For each received packet $< t_k, R_k >$ do
- Delete old packets (packets with $t_i < t_k - W$) from the
  list LPFM
- Delete packets with $R_i \leq R_k$ from the LPFM
- Add $< t_k, R_k >$ to the LPFM
```

An example of a LFPM can be:



Each bucket has one list of LFPM, which is updated independently.
To estimate, at a given time t, the cardinality of the last w units of time, we first update all of the LFPM lists to include only packets that their arrival time is less than $t - w$, then compute the largest R_i, and with these the cardinality is computed like in the HyperLogLog algorithm.

The error on the estimation is about $\frac{1.04}{\sqrt{m}}$, meaning the error rate is around 1-3%.

Denoting n as the real number of flows, the algorithm requires space of only $m * \log_2 \log_2 \frac{n}{m}$ bits.

Workflow of my algorithm:

ICD:

To run the script, you should have BitVector already installed.
You can either run the script without changing the parameters, if so, just type in Window's cmd <python sliding_hll.py> or in Linux's terminal <python3 sliding_hll.py>
If you want to change the parameters too, then:
python slide.py <size_of_hash_outcome> <number_of_buckets> <size_of_window> <testing_mode>
or on linux:
python3 slide.py <size_of_hash_outcome> <number_of_buckets> <size_of_window> <testing_mode>

The parameters as seen above are, as in the Sliding HyperLogLog algorithm:

- The number of bits the hash function return, from it we derive the bucket size. This value must be a power of 2.
- The number of buckets- to know how many buckets and what is the size of the variable 'b' as explained above. This value also must be a power of 2.
- The size of the time window in seconds.
- The testing mode means if we want in parallel to the algorithm, to count the real number of active flows. This value can either be empty, which means that we are not counting the real cardinality, or 'y' which means we do count it.

When we run the program, all the LFPM lists of all the buckets are initialized to a tuple of <start_time, 0>.

From there, the script starts to listen to QUIC messages, counting each unique dcid value as in the Sliding HyperLogLog algorithm. This keeps on going up until an exception occurs- ctr+c - and then the counting phase starts and finally prints out the estimation for the cardinality.

Top-Down description:

After finding a QUIC packet, the "add" part of the algorithm starts. We call the hll_add_element function, which does the following:

- Hashing the cid in get_element_hashed function, which uses the Python's built in Sha1 hash function, which meets the conditions discussed above. The function eventually cuts and returns only the number of bits which are decided in the first parameter.
- Parsing the hashed element returned previously and finding the variables 'b' and 'c' discussed above, so we can find the bucket relevant to this element, and to find the rest of the hashed element for the next step.
- Finding the leftmost 1-bit by calling get_rho function, which finds it by subtracting the BIN_HASH_SIZE with the actual length of the c bits.

- Updating the LFPM relevant to the bucket we found in step 2, by calling update_lfpm function. This function deletes packets which are not relevant to the current window of time, or that their R_i is smaller than the R of the packet we are processing now. Finally, it adds the current packet.

When the user wants to count the cardinality of the current window of time, he should use control+c so the adding phase will be finished and the counting phase will start with hll_count_cardinality function:

- Firstly, the constant $\alpha(m)$ is computed with a well-known values for each number of buckets.
- Then, we update all the LFPM lists of all the buckets, with the same function update_lfpm discussed above, by “trying” to add a packet with $R = 0$ and the current time- this will not affect the cardinality because only the largest R’s will be chosen, and each LFPM is initialized already with a value of 0.
- While updating each LFPM we also build a new list which holds the largest R of each LFPM, for the next step.
- Finally, the harmonic mean is calculated, taking into consideration the range corrections. Finally, the function returns the cardinality.

Then the script will print out the cardinality and will stop.

performance analysis:

Adjusting the parameters:

There are 4 main parameters, we will discuss shortly each one of them:

- Enlarging **the number of unique numbers and the size of the superset**, does not change the error rate which depends on the number of buckets. Reducing this number does not change the error rate too thanks to the range corrections discussed in the original paper on the HyperLogLog algorithm.
- Changing the **number of buckets** changes according to the error rate, denoted $\frac{1.04}{\sqrt{m}}$, meaning the more buckets we have, the smaller the error rate is. For example, for 16 buckets the error rate we get is around 0.045, which is better than $\frac{1.04}{\sqrt{16}} = 0.26$. For 8192 bucket the error rate we get is 0.007 while the calculations suggest $\frac{1.04}{\sqrt{8192}} = 0.011$, again, an even better result.
  We should pay attention that the number of buckets is also influencing the size of the hash outcome- if we have less buckets, we need less bits to represent each bucket, meaning the 'b' bits are smaller, thus 'c' is larger- thus a bigger leftmost 1-bit can be achieved, and vice versa.
- Changing the **size of the hash outcome** as said previously can be too small if we have too many buckets- the 'b' bits representing the bucket will leave a too small 'c', resulting a smaller cardinality.
  Taking a 64 bit size of hash outcome will improve the error rate even further.
- Changing the **size of the window** of time changes the cardinality as expected- the smaller the window, the smaller the cardinality (because less packets are still relevant), and if we take a window big enough for the running time of the script, then the Sliding HyperLogLog algorithm is just a HyperLogLog, as expected.

Memory usage:

Each R value is represented with the BitVector class. It allows us to save only the number of bits needed to represent the position of the leftmost 1-bit. Each R value is part of a tuple with arrival time of the packet which is an integer. All the tuples are saved in a list (LFPM), representing the list of the bucket. All the buckets are part of a list too. Hence, the memory usage is of a list of buckets, each consists of a list of tuples of integers and BitVectors. Hence, the bigger the bucket size and the more buckets we use, the bigger the list of buckets gets. Moreover, the bigger the window of time we look at, the potentially bigger LFPM lists and thus the list of buckets get bigger too.

All the rest of the parameters are integers or constants, thus having a low effect on the memory usage.

We do not discuss about the sha1 hash function as it is a temporary memory usage, because in the end, we only save from this number the position of the leftmost bit, thus it is not affecting the memory usage. The same goes for the network packets caught while listening to the network.

In conclusion, denoting n as the real number of flows, and assuming the size of the LFPM list and the size of the bucket are small enough to be considered as a constant, the algorithm requires space of only $m * \log_2 \log_2 \frac{n}{m}$ bits.

CPU usage:

The time complexity depends mainly on the size of the output of the hash function, but as this size is fixed, we can consider the running time to O(1).

We start the algorithm by initializing the buckets to 0, which means initializing a list of lists, each list has a tuple of the starting time (calculated beforehand) and a BitVector initialized to 0, so that takes an order of O(m), or constant as this number is fixed beforehand.

When adding a new packet to the LFPM, it takes the time of hashing, parsing the hashed value and finally updating the LFPM list, all considered to be O(1) actions.

When counting the cardinality we depend on the number of buckets m, finding the alpha constant and calculating the product, thus again the cost is O(m), but this number is fixed in most implementations, thus a cost of O(1) can be considered.

In conclusion, the time complexity is either O(m) or O(1), depending on if the number of buckets is considered a constant or not.

Summary and Conclusions:

We presented the problem of counting the cardinality currently running on a network, specifically for QUIC, as a complicated problem especially in terms of memory usage, and found that an algorithm called HyperLogLog solve this problem in a space complexity of $O\left(mloglog\left(\frac{n}{m}\right)\right)$, which is an outstanding result, while having an error rate of $\frac{1.04}{\sqrt{m}}$, which is approximately an error rate of 1-3%. We also discovered that the actual results could get even better than that in real life.

In addition to this algorithm, an improvement named Sliding HyperLogLog was presented, using the HyperLogLog algorithm but only for the last window of time, which can be dependent on the user. This algorithm does not drop the effectiveness of the original algorithm, meaning that the error rate stays the same, but with an additional useful feature.

In the end, I presented an implementation of the Sliding HyperLogLog, which showed that the theoretical results were accurate.

In conclusion, this algorithm is highly recommended if you face this problem.

Recommendations:

Every company which wants to count a cardinality of high number of unique elements should either use this algorithm or use it as a tool which you can get from other companies, like Amazon, etc.

A future improvement could be to use this algorithm to evaluate the cardinality in the near future- should it go up or down, or maybe stay the same, or even drop down sharply. This analysis can help companies predict what will happen soon, or what should they do in the near future.

References:

1. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm:

   http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf

2. yousra Chabchoub, Georges Hébrail. Sliding HyperLogLog: Estimating cardinality in a data stream. 2010. ffhal-00465313
   https://hal.archives-ouvertes.fr/hal-00465313/document
3. QUIC information:

   https://datatracker.ietf.org/wg/quic/about/

   https://en.wikipedia.org/wiki/QUIC