Tracking Audience Statistics with HyperLogLog

Evgeny Skvortsov, Jeff Wilhelm, Will Bradbury,
Josh Bao, Andreas Ulbrich, Lawrence Tsang

Google LLC

February 2021

Abstract

HyperLogLog is the state of the art nearly optimal algorithm for approximate cardinality estimation. We consider the application of HyperLogLog for building scalable systems for internet audience reach reporting. We present an extension of HyperLogLog that enables tracking additional information about the audience, such as demographic distribution, frequency histogram or fraction of spam. We also give an intuitive explanation of why HyperLogLog works, which we find useful, as intuition of the proof in the original HyperLogLog paper requires a lot of effort to understand. This extension and the intuition are itself generic and are not limited to internet reach reporting.

1 Introduction

HyperLogLog [1] is an algorithm for approximate cardinality estimation. A HyperLogLog sketch is a datastructure that describes a collection of objects and can be used to estimate approximate cardinality of the number of unique objects in the collection. The size of the sketch is bounded by a constant factor, which depends on the required accuracy of the cardinality estimation. The structure of the sketch allows computing unions of sketches. That is given a sketch S_1 representing collection C_1 and a sketch S_2 representing collection C_2 one can compute sketch S that would represent collection $C_1 \cup C_2$.

We are interested in the applications of the HyperLogLog to the Internet advertising reach reporting [2]. Reach of a campaign or a group of campaigns is the number of unique people that were exposed to this campaign or group of campaigns.

The ability to compute unions of sketches makes HyperLogLog valuable for efficient reach reporting. The system can store sketches of individual campaigns and allow interactive queries for arbitrary campaign groups. To compute the

reach of a campaign group the system extracts sketches for all of the campaigns of the group and computes union of those, then extracting the cardinality.

In addition to total number of people in the audience the reach reports may have additional information like demographic and frequency distribution in the audience. In this paper we propose an extension of HyperLogLog that stores a sample of characteristics of the audience, which is updated when a new event is inserted, as well as when sketches are merged. Characteristics of the sample stored along with the HyperLogLog sketch can then be used for estimating statistical distribution of the audience across various dimensions, such as demographics and frequencies of ad exposure.

The rest of the paper consists of two sections. In Section 2 we describe extensions of the HyperLogLog for tracking of user statistics. In Section 3 we discuss a theoretical algorithm that gives a simple intuition of why HyperLogLog cardinality estimate works.

2 Statistics tracking HyperLogLog

We extend the HyperLogLog algorithm by making it trace characteristics of a sample of users. Specifically, for each register of the sketch we store information about characteristics of one user. This user is identified in the sketch by a combination of the number of leading zeroes in their integer hash (the value that is generally tracked in HyperLogLog sketch registers) and an additional integer indicator hash of the small number of bits.

2.1 Storing demographics and frequency in HyperLogLog

Algorithm 1 shows the extension of HyperLogLog that tracks demographics and impression frequency of the audience.

input : Stream of events $\mathcal{E}$
output : Extended HyperLogLog sketch
parameters: The number of registers $m = 2^{b_m}$ and the size of the auxiliary indicator space $s = 2^{b_s}$, function h hashing user ids to `uint64`

1 initialize m registers $M[0], \ldots, M[m-1]$ to 0;
2 initialize m auxiliary indicators $I[0], \ldots, I[m-1]$ to 0;
3 initialize m frequency counts $F[0], \ldots, F[m-1]$ to 0;
4 initialize m demographic samples $D[0], \ldots, D[m-1]$ to *null*;
5 **for** $e \in \mathcal{E}$ **do**
6 | let $x = h(e.user\_id)$;
7 | let $j = \langle x_0, \ldots, x_{b_m-1} \rangle_2$;
8 | let $i = \langle x_{b_m}, \ldots, x_{b_m+b_s-1} \rangle_2$;
9 | let $w = (\min\{t \mid t >= 0, x_{b_m+b_s+t} = 1\}) + 1$;
10 | **if** $M[j] < w$ *or* $(M[j] = w$ *and* $I[j] < i)$ **then**
11 | | update $I[j] = i$;
12 | | update $F[j] = 1$;
13 | | update $D[j] = e.user\_demo$;
14 | **else if** $M[j] = w$ *and* $I[j] = i$ **then**
15 | | update $F[j] = F[j] + 1$;
16 | | **if** $D[j]! = e.user\_demo$ **then**
17 | | | choose $D[j]$ from $(D[j], e.user\_demo)$ following a pre-specified ranking randomly selected for each j;
18 | | **end**
19 | **end**
20 | update $M[j] = \max(M[j], w)$;
21 **end**
22 return (M, I, F, D);

Algorithm 1: HyperLogLog with demographics and frequency tracking

Note that each register is designed to keep information about one sampled user. Condition on line 15 is satisfied if the demographic of the *user_id* is inconsistent from event to event, or if we had a collision of both $M[j]$ and $I[j]$ for two different users that impacted the sketch. In this case on line 16 we make a deterministic choice between the demo in the register and the demo of the event. Furthermore this choice must be done based on a full ordering of the demo values by priority (which could vary from register to register) so that we are guaranteed that the sketch does not depend on the order of element insertion, which is a useful property for software debugging and maintenance.

Algorithm 2 shows how to merge sketches created by Algorithm 1.

input : Two extended sketches to merge (M_1, I_1, F_1, D_1), (M_2, I_2, F_2, D_2)
output : Extended HyperLogLog sketch
parameters: The number of registers $m = 2^{b_m}$

initialize m registers $M[0], \ldots, M[m-1]$ to 0;
initialize m auxiliary indicators $I[0], \ldots, I[m-1]$ to 0;
initialize m frequency counts $F[0], \ldots, F[m-1]$ to 0;
initialize m demographic samples $D[0], \ldots, D[m-1]$ to *null*;
for $j \in [0, \ldots, m-1]$ **do**
    if $M_1[j] < M_2[j]$ *or* $(M_1[j] = M_2[j]$ *and* $I_1[j] < I_2[j])$ **then**
        update $M[j] = M_2[j]$;
        update $I[j] = I_2[j]$;
        update $F[j] = F_2[j]$;
        update $D[j] = D_2[j]$;
    else if $M_1[j] = M_2[j]$ *and* $I_1[j] = I_2[j]$ **then**
        update $M[j] = M_1[j]$;
        update $I[j] = I_1[j]$;
        update $F[j] = F_1[j] + F_2[j]$;
        choose $D[j]$ from $(D_1[j], D_2[j])$ following a pre-specified ranking randomly selected for each i;
    else
        update $M[j] = M_1[j]$;
        update $I[j] = I_1[j]$;
        update $F[j] = F_1[j]$;
        update $D[j] = D_1[j]$;
    end
end
return (M, I, F, D);

Algorithm 2: Merging HyperLogLog sketches that track demographics and frequency

Lemma 1. *For any $k > 0$, among all non-empty registers the expected fraction of registers j such that there are k values of user_id in the sketch that has $M[j]$ leading zeros is equal to 2^{-k}.*

Proof. Consider a register j. Consider a Markov chain, where the state k is the number of objects that are stored in the sketch and have the number of leading zeros equal to $M[j]$. Consider an event that an object z was added to the sketch and was hashed to the register and that has greater than or equal to $M[j]$ leading zeros.

With probability 0.5 the hash of this new object will have exactly $M[j]$ leading zeros and with probability 0.5 it will have more than $M[j]$ leading zeros. Indeed it will have more than $M[j]$ leading zeros if and only if $(M[j])$-th bit in binary representation of $h(z)$ is zero.

If z has more than $M[j]$ leading zeros then the chain transitions to state 1 regardless of which state it was in. If z has exactly $M[j]$ zeros then the chain transitions from state k to state $k + 1$.

Let $p(k)$ be the probability of the k-th state. At each step of the Markov chain we have

$$p(1) = \sum_{k \geq 1} 0.5 \cdot p(k) = 0.5 \cdot \sum_{k \geq 1} p(k) = 0.5.$$

While for $k > 1$ we have $p(k) = p(k - 1) \cdot 0.5$. From which the desired equality $p(k) = 2^{-k}$ follows. □

The probability of full collisions, i.e. collision of the number of leading zeros and auxiliary hash together, is low and the impact on the histogram is not significant. Indeed Lemma 1 shows that half of the registers have a unique element that has the maximal number of leading zeros, and from there the number of registers with k colliding elements decreases exponentially. In a practical setting using an indicator of the size of 1 byte, i.e. $b_s = 8, s = 256$, makes the number of registers occupied by colliding registers insignificant. We leave getting exact estimates for the probability of collision outside of the scope of this paper.

Demographic and frequency distributions can be recovered from the sketch via extrapolating from the sampled demographic values. Algorithm 3 shows how this extrapolation is done for the demographic distribution.

2.2 Generalized statistics tracking with HyperLogLog

We can generalize Algorithm 1 to aggregate arbitrary statistics, as long as an update to the information about an individual user can be done with a commutative associative operation. Commutativity and associativity are required to ensure that the sketch does not depend on the order of elements being added to it. This generalization is Algorithm 4.

Function $\mathcal{A}$ can be an arbitrary function for initializing and accumulation of characteristics of a user. Algorithm 5 shows an example of such function that

input : demographic samples $D[0], \ldots, D[m-1]$ from the extended HyperLogLog sketch
output : demographic distribution $\mathcal{D}$ mapping demographic category to the estimated fraction in the audience

initialize $\mathcal{D}[d] = 0$ for all demographic buckets d;
initialize $total = 0$;
for $j \in \{0, \ldots, m-1\}$ **do**
  if $D[j]$ *is not null* **then**
    update $\mathcal{D}[D[j]] = \mathcal{D}[D[j]] + 1$;
    update $total = total + 1$
  end
end
for $d \in \mathcal{D}$ **do**
  update $\mathcal{D}[d] = \mathcal{D}[d]/total$
end
return $\mathcal{D}$

Algorithm 3: Extracting demographic distribution from the extended HyperLogLog sketch.

input : Stream of events $\mathcal{E}$
output : Extended HyperLogLog sketch
parameters: The number of registers $m = 2^{b_m}$ and the size of the auxiliary indicator space $s = 2^{b_s}$, function h hashing user ids to `uint64`, commutative associative function $\mathcal{A}$ for statistics initialization and accumulation.

initialize m registers $M[0], \ldots, M[m-1]$ to 0;
initialize m auxiliary indicators $I[0], \ldots, I[m-1]$ to 0;
initialize m statistic objects $S[0], \ldots, S[m-1]$ to *null*;
for $e \in \mathcal{E}$ **do**
  let $x = h(e.user\_id)$;
  let $j = \langle x_0, \ldots, x_{b_m-1} \rangle_2$;
  let $i = \langle x_{b_m}, \ldots, x_{b_m+b_s-1} \rangle_2$;
  let $w = (\min\{t \mid t >= 0, x_{b_m+b_s+t} = 1\}) + 1$;
  if $M[j] < w$ *or* $(M[j] = w$ *and* $I[j] < i)$ **then**
    update $S[j] = \mathcal{A}(null, e)$; `// Initializing` $S[j]$
  else if $M[j] = w$ *and* $I[j] = i$ **then**
    update $S[j] = \mathcal{A}(S[j], e)$; `// Updating` $S[j]$
  end
  update $M[j] = \max(M[j], w)$;
end

Algorithm 4: HyperLogLog extended for arbitrary statistics tracking

```
   input      : Current state s which is null or of the form
                (frequency, watch_time, user_demo, spam_frequency)
                and an event e
   output     : Updated state
 1 if s.is_spam then
 2 |  let spam_count = 1;
 3 else
 4 |  let spam_count = 0;
 5 end
 6 if s is null then
 7 |  return (1, e.watch_time, e.user_demo, spam_frequency);
 8 end
 9 update s.frequency = s.frequency + 1;
10 update s.watch_time = watch_time + e.watch;
11 chose s.user_demo from (s.user_demo, e.user_demo) following a
    pre-specified ranking;
12 return s;
```

Algorithm 5: Example of a statistic accumulation function tracking total video watch time, frequency, demographics and spam fraction.

can be called from Algorithm 4 to accumulate statistics about demographics, frequency, watch-time distribution and fraction of spam traffic.

Algorithm 5 is written to handle some events marked as spam afterwards, say based on some batch pipeleine or manual analysis. These events should be inserted into the sketch with is_spam set to $true$. The fraction of users that were added to the sketch purely due to spam can then be estimated as the fraction of registers for which $frequency = spam\_frequency$.

Note that if $watch\_time$ is measured with high granularity, e.g. up to microsecond, then there will be no two users with the same watch time and therefore aggregating the watch time into a distribution is not helpful, as each value would have overly high uncertainty. Yet the sketch stores an almost uniform sample of the watch times and can be used as such. For instance median watch time can be estimated as the median watch time value stored in the registers. Intuitively such estimate would be reasonable, but calculating its exact variance is a non trivial statistical exercise which is outside of the scope of this paper.

3 HyperReal: building intuition for HyperLogLog

In this section we discuss the HyperReal algorithm, which we find helps build intuition for how HyperLogLog and extended HyperLogLog works. HyperReal differs from HyperLogLog in that its registers store float valued hashes rather than counts of leading zeros. Real values take more space than counts of zeros and thus HyperReal is not practical, but they make the mechanics of the algorithm simpler and thus helps build intuition about how HyperLogLog works.

input : Stream of events $\mathcal{E}$
output : Extended HyperLogLog sketch
parameters: Function r hashing user ids unformly at random to integer range $[0, m-1]$ and function f hashing user ids to `float` uniformly at random in the range $[0, 1]$

```
1 initialize m float valued registers F[0], ..., F[m − 1] to 1;
2 for e ∈ E do
3 |   let w = f(e.user_id);
4 |   let j = r(e.user_id);
5 |   update F[j] = min(F[j], w);
6 end
7 return F;
```

Algorithm 6: HyperReal algorithm

Algorithm 6 shows the creation of the HyperReal sketch. Each element is hashed to a random register and for each register minimal value of a float-valued hash is computed.

Algorithm 7 shows how cardinality is estimated from HyperReal sketch. It is easy to see that it implements formula

$$n \approx \frac{m^2}{\sum_j F[j]}. \tag{1}$$

input : HyperReal sketch F
output : Estimation of cardinality of the set of objects inserted into F

```
1 let v = ∑_j F[j]/m;
2 return m/v;
```

Algorithm 7: Cardinality estimation for HyperReal sketch

Lemma 2. *For large m and large cardinality $n, n \gg m$ of objects inserted into HyperReal sketch, Algoritm 7 gives an unbiased estimate of the cardinality n.*

Proof. For large n and m each register has approximately n/m elements hashed to it. Thus each register is a minimum of about n/m random variables uniformly distributed over $[0, 1]$ and therefore each register approximately follows $M[j] \sim Beta(1, n/m)$. Since $n \gg m$ we have Beta distribution well approximated by an Exponential distribution and thus $M[j] \sim m/n \cdot Exp(1)$.

Therefore value v computed on line 1 of Algorithm 7 is approximately equal to the expectation of a random variable distributed according to $m/n \cdot Exp(1)$ and is thus equal to m/n. Thus the returned value is approximately equal to n, which is the desired quantity to estimate. □

HyperLogLog achieves extra efficiency compared to HyperReal by storing an integer value

$$M[j] = -\lfloor \log_2(F[j]) \rfloor, \tag{2}$$

which can fit in one byte, compared to 8 bytes usually required for high precision float number storage.

If we invert equation 2 we get an approximate equality

$$F[j] \approx 2^{-M[j]} \tag{3}$$

Note that HLL's cardinality estimate formula

$$n \approx \alpha_m \cdot \frac{m^2}{\sum_j 2^{-M[j]}} \tag{4}$$

can be obtained by substituting approximate Equaltion 3 into the formulas of Algorithm 7. Normalization constant α_m happens to account for the truncation of the float value to the floor.

Algorithm 8 extends HyperReal for tracking user statistics. The probability of float valued hash collision is zero and thus $F[j]$ for each register is defined by a unique element. For this element we store and update user characteristics that we need to track such as demographics and frequency. Algorithm 4 is more complicated as it has to deal with hash collisions to save the space storing in each register two bytes $M[j], I[j]$ instead of a float $F[j]$.

input : Stream of events $\mathcal{E}$
output : Extended HyperLogLog sketch
parameters: Function r hashing user ids unformly at random to integer range $[0, m-1]$ and function f hashing user ids to `float` uniformly at random in the range $[0, 1]$, commutative associative function $\mathcal{A}$ for statistics accumlation.

initialize m registers $F[0], \ldots, F[m-1]$ to 1;
initialize m statistic objects $S[0], \ldots, S[m-1]$ to $null$;
for $e \in \mathcal{E}$ **do**
    let $x = f(e.user\_id)$;
    let $j = r(e.user\_id)$;
    if $x < F[j]$ **then**
        update $S[j] = \mathcal{A}(null, e)$; `// Initializing` $S[j]$
    else if $x = F[j]$ **then**
        update $S[j] = \mathcal{A}(S[j], e)$; `// Updating` $S[j]$
    end
    update $F[j] = \min(F[j], x)$;
end

Algorithm 8: HyperReal extended for arbitrary statistics tracking

Acknowledgment: We are grateful to Jim Koehler for multiple discussions of the algorithms presented here and valuable advice on preparation of this report for publication.

References

[1] Frédéric Meunier, Olivier Gandouet, Éric Fusy, and Philippe Flajolet, *Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm*, Discrete Mathematics & Theoretical Computer Science (2007).

[2] Evgeny Skvortsov and Jim Koehler, *Virtual people: Actionable reach modeling*, (2019), available at `https://research.google/pubs/pub48387/`.