# Understanding the HyperLogLog: a Near-Optimal Cardinality Estimation Algorithm

**Philippe Flajolet, Èric Fusy, Olivier Gandouet and Frédéric Meunier**

Camilo Morales and Pascal Welke

Universität Bonn, Germany

**Abstract**

*The HyperLogLog algorithm (HLL)* is a method to estimate the number of distinct elements in large datasets i.e. cardinality, in a single pass, and using a very small amount of memory. The HLL algorithm is an optimization of the method presented in 2003 by Durand and Flajolet in the paper *LogLog Counting of Large Cardinalities*. In this report we analyze the so called *cornerstone* of Big Data infrastructures, give a detailed description of the algorithm and explain the mathematical intuition behind it.

## 1 Introduction

The Hyperloglog algorithm [1] has become the main pillar, also called *cornerstone* [2], of Big Data Infrastructures. In this context, we refer to Big Data Infrastructures to the ones that regularly manage massive amounts of data arriving at a high rate. Due to the inherent difficulties of storing this kind of data in main memory, or even secondary storage, it must be processed on time as soon as it arrives.
The HLL method, based on probability and statistics, tackles the problem of calculating the cardinality when the number of distinct elements is so big that counting them exactly would require an undesirable amount of memory. With traditional methods (e.g. exact counting) we would need gigabytes of memory for counting a few billion elements, while with HLL we only need, for example, $log_2(log_2(2^{32})) = 5$ bits for counting up to 4 billion distinct elements.

### 1.1 Why Counting Large Cardinalities?

The following are two typical examples, also used by Flajolet in his publications[1; 3], to illustrate the usefulness of efficiently estimate large cardinalities:

- **Optimization of database systems:** Very often, database systems perform computationally complex operations such as joins of tables. Specifically in the case of the join, the complexity depends strongly on the number of distinct rows in the tables involved. Such systems can get great benefit by knowing in advance, how many distinct rows the involved tables have, and in this way, predict how complex the operation will be. This information could help the system select the most appropriate join strategy, and also to estimate the size of the outcome.
- **Detect of denial of service attacks:** The incoming traffic of web services like Facebook or Twitter, is usually measured by the number of distinct request the servers receive per time slice (distinct source IP's). The servers need to allocate certain amount of resources to process each request. If the number of distinct incoming requests (distinct IP's) increases in an abnormal way in a very short period of time, the resources could get

rapidly exhausted making the server crash. This "method" of flooding the servers with an immense amount of made-up random IP's is known as **Denial of Service Attack**. It has become famous due to make almighty infrastructures such as Facebook or LinkedIn crash in minutes. Using HLL, abnormal traffic behavior could be rapidly detected, by estimating the number of distinct incoming IP's, preventing the server to crash, and the service to go down.

In this last example, it can be seen that it is not necessary to calculate the exact number of incoming connections in order to detect an abnormal traffic behavior. If we are able to come up with a "good" estimate in short time, it is enough for the detection.
While the difference between the estimate and the exact number would be relatively small, the difference in the size of the memory needed to calculate it exactly compared with the memory needed to estimate it approximately, is, as we show later, rather very big.

## 1.2 Why Estimate Rather Than Calculate Exactly?

The problem of calculating **exact** cardinalities comes when the number of distinct elements is so big that the amount of memory needed for counting it grows until becoming unacceptable, or the memory available is even insufficient. As shown in the following two examples, large amounts of memory (at least Gigabytes) are needed to calculate exactly the number of distinct elements in "large" datasets. In this contexts "large" means at least billions of elements. This, added to the fact that the data is expected to keep growing, makes unfeasible in practice to allocate such amount of resources. Let's illustrate this with two examples:

- **Brute force approach**: If our input dataset has $n$ distinct elements, the Brute Force method saves in memory each one of the $n$ distinct elements as it receives them (streamwise). If each element needs $\log_2(element)$ bits to be stored, we need in total: $n \log_2(element)$ bits. Since we are considering large datasets, as soon as $n$ is counted in billions, we would need: $\log_2(element)$ Gigabytes. (since 1 billion bits = 1 Gb).
- **Naive Bitmap approach**: We use a function that maps each distinct element to a bit in a bit field without collisions. Now each distinct element need only 1 bit to be stored. Again, as soon as $n$ (number of distinct elements in our input dataset) is counted in billions we would need at least 1 Gigabyte.

How could we estimate the cardinality of large datasets using less than 1 bit to represent each distinct element, if it is valid to estimate rather than to calculate exactly?

## 1.3 Formal Definition of the Problem

**Def**.(Cardinality *estimation*):

- **Given:** A multiset $\mathcal{M}$ of unknown "large" cardinality $n$, where $n \in \mathbb{N}$.
- **Assumed:**
  - Limited storage capacity: Not possible to store all elements.
  - Limited time: Quick response time $\Rightarrow$ read once data.
  - It is valid to estimate $n$ rather than calculate exactly.
- **Find:** A plausible estimation of the number of distinct elements $n$ in $\mathcal{M}$.

# 2 The HyperLogLog Algorithm

As usual with Big Data algorithms, technically the HLL method is quite straightforward (no more than 7 lines of pseudo-code), while understanding it intuitively is rather complex.

For the sake of clarity, first we'll show the complete picture of the algorithm without any intuition behind, and then, after the whole picture was presented, we'll explain what is happening underlying each step.

The complete HLL algorithm is described in the following scheme:

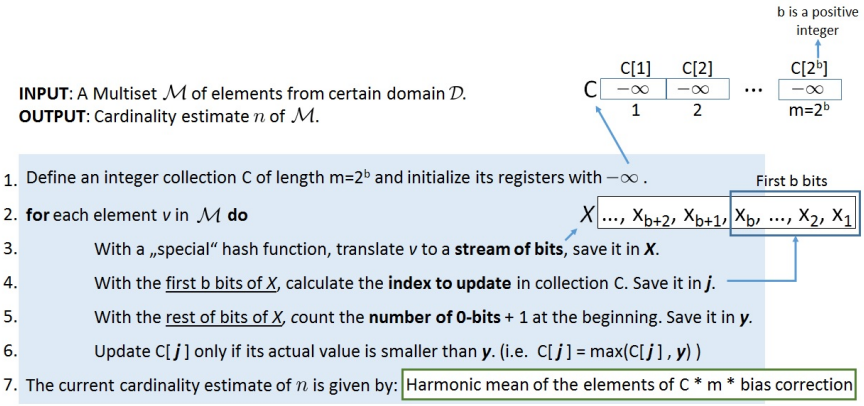


Figure 1: The HyperLogLog algorithm.

Now the underlying intuition:

- We want to use certain amount of buckets, lets call it $m$, to divide the $n$ distinct elements of the input multiset such that each bucket comprises approximately the same number of elements, namely $\approx \frac{n}{m}$. In figure 1, the buckets are represented by the registers of the integer collection C.
  - **Note:** At first sight it is not quite clear why we should divide and use buckets: The method is called "Stochastic averaging" and is used to reduce the variance of the estimator. Is explained in more detail in Section 2.1.
- To make each bucket comprise $\approx \frac{n}{m}$ elements, we need somehow to transform our input multiset $\mathcal{M}$ into an **ideal multiset**. This is why:
  - Ideal Multisets are those for which its elements are *evenly* distributed over a certain domain.
  - **Main idea**: if the elements are evenly distributed, and we divide the multiset in $m$ approximately equally-sized sections, each section should contain $\approx \frac{n}{m}$ elements.
- To transform the input multiset into an ideal multiset the HLL algorithm uses a special **hash function**. This function will transform the elements of the input set coming from certain data domain $D$, to a stream of 0's and 1's, $h : D \to \{0,1\}^\infty$, such that:
  - Each bit of the hashed values must be **independent from each other and must have the same probability** $\frac{1}{2}$ of being equal to 1. In other words, we want the

hash function to flip a fair coin several times and return the outcome as a stream of bits.

– The hashed value of two equal elements must be also equal: if $a = b \Rightarrow h(a) = h(b)$.

All the steps explained so far can be seen graphically in the figure 2.

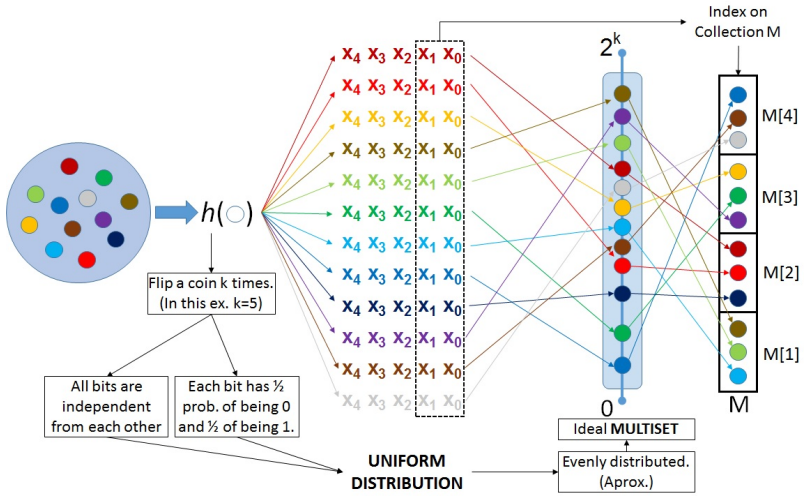


Figure 2: Randomization of elements and division in buckets.

- To assign the elements to the $m$ buckets, the HLL algorithm uses the first $b$ bits of the hashed values (recall that the number of buckets was given by $m = 2^b$), and take the decimal representation of this first $b$ bits as the index of the register (bucket) in the C collection. Therefore, the number of buckets is given by the number of bits we want to use at the beginning of the hashed values.

  **Note**: As it will be shown in the proof later on, the standard error (relative accuracy) of the HLL algorithm is given by $\frac{\beta_m}{\sqrt{m}}$ (where $\beta_m$ is a constant). The denominator $\sqrt{m}$ tells us that we have a trade-off between accuracy and number of buckets (memory): the more buckets, the better the accuracy, but also the more memory we would need, as we would need more buckets.

- An explanation of why we can expect each bucket to comprise $\approx \frac{n}{m}$ elements is the following:

  – Consider that we use only the first bit of the hashed values to assign the elements to the buckets. This means we will have only 2 buckets (i.e. $2^1$). Since the hash function is flipping a fair coin everytime, and each toss is independent from each other and have the same $\frac{1}{2}$ probability of being 0 or 1, we can expect that, approximately, half of the first bits of all hashed values (column $x_0$ in figure 2) to be 0 and half of them to be 1. Thus, half of them would go to the first bucket (index 0) and half to the second (index 1). The same principle applies if we use more bits: 4 buckets for 2 bits, 8 for 3 bits, etc.

So far we have explained how to randomize and divide the elements in buckets, now we explain how to estimate the number of distinct elements in each bucket:

- Clearly, we don't want to store all the elements belonging to each bucket (it would be just like a randomized brute force approach), but still we want to know how many distinct elements there are. To do this the HLL algorithm saves in each bucket only the longest run of starting 0-bits+1 seen out of all the hashed values of the elements that belong to that bucket. With this number, it uses the following estimator to approximate the number of distinct elements in the bucket:

$$\#\text{ of distinct elements in bucket} \approx 2^{\text{length of longest run of starting 0-bits + 1}}$$

  This estimator relies on the probability of the bit-pattern observable (in this case the number of consecutive starting 0-bits), and on its expectation to calculate the number of distinct elements. This is called *Probabilistic Counting* and how it works is explained in more detail in Section 2.2.

- Since we only have to store the length of the longest run of 0-bits+1 seen to estimate approximately the number of distinct elements in each bucket, we just need to allocate $log_2$(longest run of 0-bits+1) bits, which is at max $log_2$(max possible run of 0-bits+1) bits. From here comes the ***LogLog*** in the name of the paper. Specifically, if cardinality $n$ needs to be estimated:

  - We need at max $\log_2(n)$ bits (specifically 0-bits) to estimate $n$ since, according to our estimator $n \approx 2^{\text{longest run of starting 0-bits+1}}$.
  - But all we need to store to estimate $n$ is the length of the longest run of starting 0-bits, which is at max: $\log_2(\log_2(n))$:

$$\log_2 \log_2 n \to \log_2 \log_2 2^{\text{max. pos. run of 0-bits}} \to \log_2 \text{max. pos. run of 0-bits}$$

    In practice, this is the number of bits necessary to store numbers like 8, 16 or 32 which is $log_2(32) = 5$ bits. (5 bits to estimate $2^{32}$ distinct elements, i.e. more than 4 billion distinct elements!)

- Although being able to estimate up to $2^{32}$ distinct elements with 5 bits is fascinating, the estimator has an important problem: It has a big variance, and therefore, huge outliers can appear in the estimations.

  - Why big variance → What is the maximum number of tosses until we get the first Heads? It is not bounded, is theoretically infinite! (Although $\mathbb{P}\text{rob.} \approx 0$)
  - This phenomenon is called *slow-decaying right tails* for the shape of the probability distributions of the geometric-distributed random variables. (i.e. number of tries until the first success)

  Naively, the solution seems quite simple: just repeat the experiment several times and use some averaging technique to get the most repeated result. The problem is that in our set up we consider single-pass data, therefore, we can only perform the random experiment once.

  The solution presented in the paper is called *Stochastic averaging*. (See section 2.1)

- Finally, we get to the HLL estimator considering that: The cardinality of each bucket, $2^{\text{longest run of 0-bits + 1}}$, should be $\approx \frac{n}{m}$. Then, the Harmonic mean of the $m$ buckets:

$\frac{m}{\sum_{j=1}^{m} \frac{1}{2^{C[j]}}}$, should also be $\approx \frac{n}{m}$. If we call $Z$ this term $\frac{1}{\sum_{j=1}^{m} \frac{1}{2^{C[j]}}}$, then the Harmonic mean is now: $mZ$. Since, $mZ \approx \frac{n}{m}$, then $m(mZ)$ should be $\approx n$. But the estimator $m^2Z$ has an additive bias of 1.33, which is corrected with the constant $\alpha_m$. Including $\alpha_m$ we get to the final cardinality estimator: $E = \alpha_m m^2 Z$, which is the estimator presented in the paper.

## 2.1 Stochastic Averaging

Stochastic averaging technique *emulates* multiple "smaller" random experiments out of just one "big" random experiment. Specifically in the HLL algorithm, it first divide the elements evenly in $m$ buckets, and then estimate the number of distinct elements **in each bucket**. The random experiment is then performed $m$ times instead of only 1. As we know from ideal multisets, each bucket should comprise $\approx \frac{n}{m}$ elements, but now we have to take into account that some buckets could contain huge outliers (very far from the expected $\frac{n}{m}$). We have to find then an appropriate averaging technique such that it decrease, as much as possible, the effect of the outliers.
The selection of this averaging technique is the core difference between the original *LogLog* algorithm, which uses the arithmetic mean, the *SuperLogLog* which uses a "manual" version of the geometric mean (they get rid manually of the highest 30% of buckets), and the HLL which uses the **Harmonic mean**.

The Harmonic mean, the least known of the means, is the most appropriate for the HLL method. It could be seen as an "inverse-weighted mean" where the weight of each operand is the reciprocal of itself. This means, the bigger the outlier, the smaller the effect it has on the averaging, exactly what we needed it to do.

## 2.2 Probabilistic Counting

Intuitively Probabilistic Counting works in following way:

- Consider the random experiment of tossing a fair coin 2 times. What are all the possible outcomes and their probabilities? (See Table 1)

| | | |
|---|---|---|
| T | T | $\mathbb{P} = \frac{1}{2} * \frac{1}{2} = \frac{1}{2^2} \longrightarrow 25\%$ |
| T | H | $\mathbb{P} = 25\%$ |
| H | T | $\mathbb{P} = 25\%$ |
| H | H | $\mathbb{P} = 25\%$ |

Table 1: Possible outcomes of 2 coin tosses.

As all possible patterns have the same probability to occur, this tells us that we could expect to see, on average, each pattern once every 4 times we perform the random experiment, e.g. we can expect to see the pattern HH once every 4 repetitions of 2 coin tosses.

Since all patterns have the same probability, this doesn't help us to calculate the number of distinct elements. Here is when the Bit-Pattern observables come into play.

The Bit-pattern observables focus on certain patterns of bits occurring at the **beginning** of the binary stream. So now our random experiment is not flip the coin 2 times, but flip

the coin as many times as needed until we see the first tails. With this, all the possible outcomes have now different probabilities. The longer the pattern, the less probable it is to occur. See table 2.

| | | | | | |
|---|---|---|---|---|---|
| ... | ... | ... | ... | H | $\mathbb{P} = \frac{1}{2} \rightarrow$ Appr. $\frac{1}{2}$ starts with H. |
| ... | ... | ... | H | H | $\mathbb{P} = \frac{1}{2^2} \rightarrow$ Appr. $\frac{1}{4}$ starts with HH. |
| ... | ... | H | H | H | $\mathbb{P} = \frac{1}{2^3} \rightarrow$ Appr. $\frac{1}{8}$ starts with HHH. |
| ... | H | H | H | H | $\mathbb{P} = \frac{1}{2^4} \rightarrow$ Appr. $\frac{1}{16}$ starts with HHHH. |
| H | H | H | H | H | $\mathbb{P} = \frac{1}{2^5} \rightarrow$ Appr. $\frac{1}{32}$ starts with HHHHH. |

Table 2: Possible Heads patterns and their probabilities.

And then, if we recall the expectation of each pattern (as shown in table 1 for two coin tosses), we can estimate the number of times the random experiment was executed, for example:

- If the longest run of heads we saw was 2, We know that we could expect to see the pattern HH, on average, once every 4 repetitions. Then, we can estimate that the experiment was performed $2^2 = 4$ times. This number is exactly the reciprocal of the probability of seeing HH, as: $\mathbb{P}(HH) = \frac{1}{2^2} = \frac{1}{4}$, and its reciprocal: $\mathbb{P}(HH)^{-1} = (\frac{1}{4})^{-1} = 4$.
  The same principle applies for all other possible patterns HHH, HHHH, etc. In general, the estimator is the reciprocal of the probability:

$$\begin{aligned}\text{Estimated number of repetitions} &\approx \mathbb{P}(\text{longest run of starting Heads is k})^{-1} \\ &= (\frac{1}{2^k})^{-1} = 2^k\end{aligned}$$

# 3 Practical Considerations

When it comes to the implementation of the HyperLogLog algorithm, there are three critical considerations that need to be taken care of:

## 3.1 Initialization of Registers

The original HLL algorithm states that the registers (i.e. buckets) should be initialized in $-\infty$. This is very useful for the proof of the asymptotically behavior of the estimator, but in practice, it comes with a big problem: If any of the registers remains untouched, even just 1, the harmonic mean would be 0:

$$\frac{m}{\frac{1}{2^{-\infty}} + \frac{1}{...} + ...} = \frac{m}{2^{\infty} + \frac{1}{...} + ...} \approx 0$$

and then, the cardinality estimation will also be 0, as:

$$E = \text{Harmonic mean} * m * \text{bias correction}$$

which means that we would be clearly **underestimating**.

- *When should we expect to have empty buckets?* The Coupon's Collector problem tells us that the expected number of distinct elements that we should have so that we could expect all buckets to be touched at least once is: $m \log_2 m$, where $m$ is the number of buckets. Therefore, if $n << m \log m$ we can expect to have at least one empty bucket, and if they are initialized with $-\infty$, the estimate will be 0.

The solution proposed in the paper is to **initialize the registers with 0 instead of** $-\infty$. By changing this, the algorithm will still generate usable estimates, as long as $n$ remain at least as a small multiple of $m$ (greater than one), i.e. we could expect to have very few empty buckets, if we have empty buckets at all.
But what happened if $n$ is not even a small multiple of $m$, for example $n = m$?

## 3.2 Small Range Corrections

If $n$ is not even a small multiple of $m$, we can expect to have several empty buckets. The authors studied how this fact affected the estimates, finding empirically that:

- The algorithm still generates plausible estimates as long as: $n \geq \frac{5}{2}m$; when $m \geq 16$.
- Contrarily, if $n < \frac{5}{2}m$, distortions start appearing in the estimates.

The solution proposed in the paper is to completely disregard the estimator $E$ of HLL, and switch to the estimator used in the Hit Counting algorithm (Linear counting [4]):

$$E^* = m \log \frac{m}{\#\text{ empty registers}} = -m \log \frac{\#\text{ empty registers}}{m}$$

This estimator performs better than the original HLL estimator when several empty buckets are expected. $E^*$ comes from the following analysis:

$$\begin{aligned}
\mathbb{P}(\text{1 register to be touched}) &= \frac{1}{m} \\
\mathbb{P}(\text{1 register to not be touched}) &= 1 - \frac{1}{m} \\
\mathbb{P}(\text{1 register remain untouched after n elements}) &= (1 - \frac{1}{m})^n \\
&= ((1 - \frac{1}{m})^m)^{\frac{n}{m}} \\
&\approx e^{-\frac{n}{m}}; \text{ since: } (1 - \frac{1}{x})^x \approx e^{-1}
\end{aligned}$$

Since we have a total of $m$ buckets, we can expect after $n$ elements, to have approximately:

$$V = me^{-\frac{n}{m}} \text{ , empty registers.}$$

Finally, solving for $n$, we get the estimate:

$$E^* = -m \log \frac{V}{m}$$

which is precisely the Linear Counting estimator.

### 3.3 Large Range Corrections

There is also a problem on the other side of the spectrum, namely: What happen if $n$ is "very large" ($n \approx 2^{\text{Maximum possible 0-bit run + 1}}$)?

The larger $n$ gets with respect to $2^{\text{Maximum possible 0-bit run + 1}}$ the more probably it is to have hash collisions. A collision mean that at least two distinct elements were hashed to the same value, and therefore, they are considered as equal for the estimation, leading the HLL algorithm to **undercount**.

Why we should expect more hash collisions as $n$ approaches $2^L$, where L is the maximum possible 0-bit run + 1, is given in the following analysis:

$$\mathbb{P}(\text{no collision after 1 element}) = \frac{2^L - 1}{2^L}\text{; only 1 bucket touched.}$$

$$\mathbb{P}(\text{no collision after 2 elements}) = \frac{2^L - 1}{2^L} * \frac{2^L - 2}{2^L}\text{; each hashed value is independent.}$$

$$\begin{aligned}\mathbb{P}(\text{no collision after k elems.}) &= \frac{2^L - 1}{2^L} * \frac{2^L - 2}{2^L} * ... * \frac{2^L - (k-2)}{2^L} * \frac{2^L - (k-1)}{2^L} \\ &\approx e^{\frac{-k(k-1)}{2*2^L}}\text{; approximation due to Taylor expansion of } e^x. \\ &= e^{\frac{-k(k-1)}{2^{L+1}}}\end{aligned}$$

$$\begin{aligned}\mathbb{P}(\text{collision after k elems.}) &= 1 - e^{\frac{-k(k-1)}{2^{L+1}}} \\ &\approx \frac{k(k-1)}{2^{L+1}}\text{ ; since: } 1 - e^{-x} \approx x \\ &\approx \frac{k^2}{2^{L+1}}\text{ ; for large values of } k : k(k-1) \approx k^2.\end{aligned}$$

Due to the $k^2$ in the numerator, it can be seen how the $\mathbb{P}$(collision) rapidly increases with the cardinality. Therefore, the bigger the cardinality, the more collisions we can expect, and the more collisions the more distinct elements will be hashed to the same vale.

The solution presented on the paper is again to switch to the Linear Counting estimator but now with the following remarks:

- Since we want to count the number of distinct elements **before being hashed** (before the collisions), and we know that several distinct elements have been hashed to the same value, we can think now that the buckets for the Linear Counting are the total number of possible hashed values, namely $2^{\text{Maximum possible 0-bit run + 1}}$, and that the total number of touched buckets come in the output of the original HyperLogLog estimator, then:
  - Total number of buckets: $2^L$. (i.e. all possible hash values)
  - Number of hit buckets $\approx E$. (Original HyperLogLog estimate)
  - Therefore, number of empty buckets $\approx 2^L - E$.

  Then, by replacing the variables in the original Linear Counting estimator, our new corrected estimator $E^*$ is:

$$E^* = -2^L \log \frac{2^L - E}{2^L} = -2^L \log\left(1 - \frac{E}{2^L}\right)$$

  which is the correction the authors presented in the paper.

# 4 Overview of the Proof

Give the cardinality estimator: $E = \alpha_m m^2 Z$, where $\alpha_m$ is the bias correction constant, $m$ is the number of registers and $Z$ is the "indicator" function: $Z = (\sum_{j=1}^{m} 2^{-M^{(j)}})^{-1}$ ($M$ is the registers collection). We want to prove the asymptotic behavior of its Expectation and Variance (standard error) as stated in the following theorem:

**Theorem 1.** *Let the algorithm HYPERLOGLOG be applied to an ideal multiset of (unknown) cardinality $n$, using $m \geq 3$ registers, and let $E$ be the resulting cardinality estimate:*

*(i) $E$ is asymptotically almost unbiased in the sense that:*

$$\frac{1}{n}\mathbb{E}_n(E) \underset{n\to\infty}{=} 1 + \delta_1(n) + o(1)$$

*(ii) The standard error defined as $\frac{1}{n}\sqrt{\mathbb{V}_n(E)}$ satisfies as $n \to \infty$:*

$$\frac{1}{n}\sqrt{\mathbb{V}_n(E)} \underset{n\to\infty}{=} \frac{\beta_m}{\sqrt{m}} + \delta_2(n) + o(1)$$

*where:*

- *$\delta_1$ and $\delta_2$ are oscillating functions of tiny amplitude. ($\approx 0$ for all practical purposes)*
- *The constant $\beta_m$ being bounded, with $\beta_{16} = 1.106$, $\beta_{32} = 1.070$, $\beta_{64} = 1.054$, $\beta_{16} = 1.06$, and $\beta_\infty = \sqrt{3\log 2 - 1} = 1.03896$.*

## 4.1 Mean Value Analysis

Steps of the proof of part (i) of Theorem 1:

1. Define the expectation of $Z$:

$$\mathbb{E}_n(Z) = \sum_{k_1,\ldots,k_m \geq 1} \frac{1}{\sum_{j=1}^{m} 2^{-k_j}} \sum_{n_1+\ldots+n_m=n} \binom{n}{n_1,\ldots,n_m} \frac{1}{m^n} \prod_{j=1}^{m} \gamma_{n_j,k_j},$$

where $\gamma_{v,k} = (1 - \frac{1}{2^k})^v - (1 - \frac{1}{2^{k-1}})^v$, for $v, k \geq 1$ and $\gamma_{0,k} = 0$.

- The probability expression $\gamma_{v,k}$ of each *summand* in the expectation comes from the following analysis:

The position of the first 1-bit in each hashed value is modeled as a Random Variable geometrically distributed (i.e. 1-bit=success, 0-bit= failure), lets call it $Y$. Then:

$$\begin{aligned}\mathbb{P}(Y \geq k) &= \mathbb{P}(\text{Position. of first 1-bit} \geq k)\\ &= \mathbb{P}(\text{All bits before k (i.e. k-1 bits) be 0})\\ &= \frac{1}{2^{k-1}}, \text{after } v \text{ tries} = \left(\frac{1}{2^{k-1}}\right)^v.\end{aligned}$$

Similarly:

$$\begin{aligned}\mathbb{P}(Y \leq k) &= \mathbb{P}(\text{Position of first 1-bit} \leq k) \\ &= \mathbb{P}(\textbf{Not all} \text{ bits before or equal k (i.e. k bits) be 0}) \\ &= 1 - \mathbb{P}(\text{All bits before or equal k (i.e. k bits) be 0}) \\ &= 1 - \frac{1}{2^k}, \text{after } v \text{ tries} = \left(1 - \frac{1}{2^k}\right)^v.\end{aligned}$$

Finally:

$$\begin{aligned}\mathbb{P}(Y = k) &= \mathbb{P}(Y \leq k) - \mathbb{P}(Y < k) = \mathbb{P}(Y \leq k) - [\,1 - \mathbb{P}(Y \geq k)] \\ &= \left(1 - \frac{1}{2^k}\right)^v - \left(1 - \frac{1}{2^{k-1}}\right)^v = \gamma_{v,k}\end{aligned}$$

2. Express $\mathbb{E}(Z)$ under the Poisson Model:

   - We need the "Poissonization" of the problem in order to be able to use the Mellin transform in following steps. "Poissonization" intuitively means that now our unknown cardinality $n$ is allowed to vary according to a Poisson law.

   Then, under the Poisson Model, $\mathbb{E}(Z)$ is defined as:

$$\mathbb{E}_{\mathcal{P}(\lambda)}(Z) = \sum_{k_1,\ldots,k_m \geq 1} \frac{1}{\sum_{j=1}^{m} 2^{-k_j}} \prod_{j=1}^{m} g\left(\frac{\lambda}{m 2^{k_j}}\right), \text{ where } g(x) = e^{-x} - e^{-2x}.$$

   where the variation of $n$ is distributed according to the Poisson law of parameter: $\lambda$ : $\mathbb{P}(N = n) = e^{-\lambda}\frac{\lambda^n}{n!}$.

3. Prove the asymptotic behavior of $\mathbb{E}(Z)$ under the Poisson model.

   We want to prove that $\mathbb{E}_{\mathcal{P}(\lambda)}(Z)$ satisfies:

$$\mathbb{E}_{\mathcal{P}(\lambda)}(Z) \underset{\lambda \to \infty}{=} \frac{\lambda}{m}\left(\frac{1}{m\alpha_m} + \epsilon_m\left(\frac{\lambda}{m}\right) + o(1)\right)$$

   where $|\epsilon_m(t)| < 5 * \frac{10^{-5}}{m}$ for $m \geq 16$.

   Steps of the proof:

   (a) The Poisson expectation is first expressed in integral form:

   Applying the identity: $\frac{1}{a} = \int_0^\infty e^{-at}dt$, and the further change of variables $t = xu$, leads us to the expression:

$$\mathbb{E}_{\mathcal{P}(x)}(Z) = H\left(\frac{x}{m}\right), \text{where } H\big(x\big) := x \int_0^{+\infty} G(x, xu)^m du.$$

   (b) Use the Mellin transform,

$$f^*(s) := \int_0^{+\infty} f(t) t^{s-1} dt$$

where $f(t)$is a function defined on $\mathbb{R}_{>0}$, to prove the following Lemma 1, which then applied to $\mathbb{E}_{\mathcal{P}(x)}(Z)$ will end the proof of the asymptotic behavior under Poisson Model:

**Lemma 1** For each fixed $u > 0$, the function $x \mapsto G(x, xu)$ has the following asymptotic behaviour as $x \to +\infty$,

$$G(x, xu) = \begin{cases} f(u)(1 + O(x^{-1})) + u\epsilon(x, u) & \text{if } u \leq 1 \\ f(u)(1 + 2\widetilde{\epsilon}(x, u) + O(x^{-1})) & \text{if } u > 1 \end{cases}$$

where $f(u) = \log_2(\frac{2+u}{1+u})$, the error terms are uniform in $u > 0$, and $|\epsilon|, |\widetilde{\epsilon}| \leq \epsilon_0 \simeq 7 * 10^{-6}$ for $x \leq 0$

(c) Finally, apply Lemma 1 to the function $H(x)$ of the first step (a), decomposing the domain of the integral, which lead us to:

$$H(x) = x\left( \int_0^\infty f(u)^m du + \epsilon_m(x) + o(1) \right), \text{where } |\epsilon_m(x)| < \frac{5 * 10^{-5}}{m} \text{ for } m \geq 16.$$

This concludes the proof of the asymptotic behavior of the expectation under the Poisson model. We need to prove now that this also applies to the "fixed-n" model.

4. "Depoissonization" of the problem: Prove that the asymptotic behavior of the expectation under the Poisson Model also applies for the fixed-$n$ model, namely: $\mathbb{E}_n(Z) = \mathbb{E}_{\mathcal{P}(x)}(Z) + O(1)$. The authors proved this by applying the*"Analytic Depoissonization"* method, by Jacquet and Szpankowski, to our known integral expression of the expectation (step 3.a).[1]

   $\Longrightarrow$ This concludes the proof of the part (i) of Theorem 1.

## 4.2 Variance Analysis

Only the proof of section (ii) of Theorem 1 is missing which refers to the variance of the indicator function, namely: $\mathbb{V}_n(Z) = \mathbb{E}_n(Z^2) - \mathbb{E}_n^2(Z)$. This proof goes through the same steps of the proof of part (i):

1. Express $\mathbb{E}_n(Z^2)$ under the Possion Model $\mathbb{E}_{\mathcal{P}(\lambda)}(Z^2)$.

2. Use the identity $\frac{1}{a^2} = \int_0^\infty te^{-at}dt$ to express $\mathbb{E}_{\mathcal{P}(\lambda)}(Z^2)$ in integral form.

3. Use Lemma 1 to prove the asymptotic behavior of the estimate under the Poisson Model.

4. Use the *"Analytic Depoissonization"* method to prove that it also applies for the "fixed-$n$ model.

[1] For details of the *"Analytic Depoissonization"* method please refer to the page 136 of the HyperLogLog paper.

### 4.3 Constants

We have 2 constants still to discuss: Bias-correction constant: $\alpha_m$ and the Standard-error constant: $\beta_m$. They are defined as:

$$\alpha_m = \frac{1}{mJ_0(m)}, \beta_m = \sqrt{m}\sqrt{\frac{J_1(m)}{J_0(m)^2} - 1}$$

where the functions $J_s(m)$ are defined through the Laplace method as:

$$\begin{cases} J_0(m) = \frac{2\log 2}{m}\big(1 + \frac{1}{m}(3\log 2 - 1) + O(m^{-2})\big) \\ J_1(m) = \frac{(2\log 2)^2}{m^2}\big(1 + \frac{3}{m}(3\log 2 - 1) + O(m^{-2})\big) \end{cases}$$

thus, when $m \to +\infty$:

$$\alpha_m \sim \frac{1}{2\log 2} = 0.72134,\ \beta_m \sim \sqrt{3\log 2 - 1} = 1.03896.$$

Replacing $\beta_m$ in (ii) of Theorem 1, we get that the standard error (i.e. relative accuracy) is bounded (1.03 when $m \to +\infty$ ) and that for any particular $m \geq 3$, the accuracy is $\approx \frac{\beta_m}{\sqrt{m}}$.

## 5 Conclusions

The HLL algorithm is a splendid mathematical method that allows counting large amounts of distinct elements, in a single pass, and with a very small amount of memory. It has a bounded error (relative accuracy) which provides the required reliability when dealing with random and probabilistic methods.

## References

[1] P. Flajolet, Éric Fusy, O. Gandouet, and et al., "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm," in *IN AOFA '07: PROCEEDINGS OF THE 2007 INTERNATIONAL CONFERENCE ON ANALYSIS OF ALGORITHMS*, 2007.

[2] "Sketch of the day: Hyperloglog — cornerstone of a big data infrastructure." `http://research.neustar.biz/2012/10/25/sketch-of-the-day-hyperloglog-cornerstone-of-a-big-data-infrastructure/`. Accessed: 2016-01-16.

[3] M. Durand and P. Flajolet, "Loglog counting of large cardinalities (extended abstract)," in *Proceedings of ESA 2003, 11th Annual European Symposium on Algorithms*, vol. 2832 of *Lecture Notes in Computer Science*, pp. 605–617, Springer, 2003.

[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, pp. 208–229, June 1990.

[5] S. Heule, M. Nunkesser, and A. Hall, "Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm," in *Proceedings of the EDBT 2013 Conference*, (Genoa, Italy), 2013.

[6] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *J. Comput. Syst. Sci.*, vol. 31, pp. 182–209, Sept. 1985.

[7] "Hyperloglog++: Google's take on engineering hll." `http://research.neustar.biz/2013/01/24/hyperloglog-googles-take-on-engineering-hll/`. Accessed: 2016-01-16.

[8] A. Rajaraman and J. D. Ullman, *Mining of massive datasets*. New York, N.Y., Cambridge: Cambridge University Press, 2012.