Final Project: HyperLogLog

Your Name(s): Anonymous

University of Washington: CSE 312 Summer 2020

1 Application

Large amounts of data are often stored in sets. There are sets of passwords, sets of names, and sets of many different types of objects. When dealing with sets in a number of different situations, a common operation that arises is determining the number of elements in a set. This is known as computing the cardinality of a set. While it would be intuitive and iterative to simply add up the number of elements in the set, checking for duplicates, such an operation would require quite a bit of computational resources. Computing the cardinality of a set this way that contains, say, 100 million elements would be unnecessarily expensive.

The HyperLogLog data structure is a probabilistic data structure that provides the ability to estimate, with a high degree of accuracy, the cardinality of a set. It is able to do so in a way that is much less computationally expensive compared to the aforementioned methods.

Being able to provide an accurate estimate for the cardinality of a set extends to many problems. Such problems include determining the number of unique queries to a search engine over a period of time [1], monitoring network systems [1], and monitoring network packets for denial of service attacks [2]. With the increasing amount of data being generated by users on the internet, analysing the data for distinct elements across large data sets becomes computationally expensive. The HyperLogLog data structure provides a manner in which such operations can be performed utilizing resources efficiently. HyperLogLog is so effective that it is used by large companies such as Google and Reddit [3].

2 Core Functionality

The HyperLogLog algorithm takes in, as its input, a multiset of elements. The algorithm uses randomization in order to provide an approximation for the cardinality of this inputted multiset. A hash function, h, is used over every element. h should be mostly uniform, so it's important that it produces uniform outputs. We observe the maximum number of leading zeroes that occur for all the hashed values. Then, assuming there are 0^{b-1} leading zeros, a good estimate for the size of the multiset is 2^b. However, a single estimation inevitably has a lot of variability, so the next step in the algorithm is performing stochastic averaging:

We divide the inputted multistream, S, into m substreams of equal size, each S_i. The number of substreams, m, is determined by the first p bits of the hash values. $m = 2^p$.

Then, we measure the maximum number of leading zeros independently per stream. These maximums are stored in an array of registers, M. For substream with index i, $E[i]$ denotes the maximum number of leading zeros in the binary representation of the input element, x, + 1. $E[i] = max_{x \in X_i}(p(x))$, where $p(x)$ is a function that computes the number of leading zeros + 1 for input element x.

The algorithm then computes the cardinality estimate, E:

$E := \alpha_m * m^2 * \sum_{j=1}^{m} 2^{-M[j]}$.

E is the harmonic mean of the estimations from each of the substreams, with the normalized bias correction. Lastly, α_m is defined to be $(m * \int_0^\infty (log_2(\frac{2+u}{1+u}))^m du)^{-1}$.

The algorithmic pseudocode is detailed as follows:

Algorithm 1 HyperLogLog[2]

function INITIALIZE(M,m)
    Initialize an array, M, of counters, each with size m, set to 0.
function ADD(v)
    Hash input data, v, with a chosen hash algorithm
    i = h(v)[0...$log_2(m) + 1$]
    Get the position of the first 1 from h(v)[$log_2(m) + 1$...len(h(v))], $p(w)$
    Set the value of $M[i]$ to $Max(p(w), M[i])$
function COUNT(x)
    Calculate the harmonic mean of the counters, $Z = \left(\sum_{j=1}^{M} 2^{-M[j]}\right)^{-1}$
    if $m = 16$ **then**
        $\alpha_m = 0.673$
    else if $m = 32$ **then**
        $\alpha_m = 0.697$
    else if $m = 64$ **then**
        $\alpha_m = 0.709$
    else
        $\alpha_m = \frac{0.7213}{1+\frac{1.079}{m}}$
    $E = \alpha_m \cdot m^2 \cdot Z$
function MERGE(HLL_1, HLL_2)
    for i = 0; i less than M; i++ **do**
        $HLL_{merged}[i] = max(HLL_1[i], HLL_2[i])$

The HyperLogLog algorithm has 3 main functions, briefly detailed as follows:

- add(v): In the add function, we hash our input data and then determine the position of the first 1. We then set the value of $M[i]$ to the element with the largest number of leading zeros.

- count(x): In the count function, we calculate the harmonic mean of the counters using the formula. Then, depending on m, we determine α_m in order to compute our final estimation for the cardinality.

- merge(HLL_1, HLL_2): We run the HLL algorihtm twice for each substream. In the merge function, we take the maximum of either of the two's maximum number of leading zeros (+1). We do this for all M_i in M. This helps us reduce over or under estimation.

3 Explanation and Intuition

HyperLogLog is a probabilistic data structure that gives us a good estimate for the cardinality of a set. It does so using a pretty neat intuitive trick. In a random stream of binary integers, where getting a 0 and a 1 occur at equal probability ($\frac{1}{2}$ for both), approximately half of all integers will start with a 0 and the other half will start with a 1, for large data sets of course. By the same logic, about a fourth will start with 01. Using this information, we can make the deduction that, if we observe a string that is, say 0001, which occurs with probability $\frac{1}{16}$, there are about 16 distinct elements in the stream. This is a rather powerful probabilistic trick [4].
From this, if we observe the number of leading zeros in a stream of hashed inputs, for the largest prefix, we estimate the cardinality as $2^{prefix-size}$. In order to improve the estimation in our HLL algorithm, we break the initial stream up into m sub-streams, each of which we calculate the maximum number of leading zeros. Afterwards, we estimate the final cardinality by taking the mean of each of maximum values from each sub-stream. The size of m largely determines the accuracy of our algorithm. In fact, the relative accuracy of the algorithm is given to be $\frac{1.30}{\sqrt{(m)}}$. So, we must determine an appropriate m for a compromise between time complexity and relative accuracy. A larger m will result in a longer run-time, and as m approaches the size of the input stream, HLL loses some of its efficient edge.
The randomization in the HLL algorithm comes from the hash function. It's important that it produces uniform outputs. We make the assumption that the bits of hashed values occur with equal probability of 12 [1]. This will greatly impact our estimate, as a biased hash function will lead us to make biased estimates.
In our algorithm, we hash on 32 bits, which will work for our situation given the size of our input streams. However, in order to have a cardinality estimate in excess of 10^9, 64 or 128 bits might be necessary. It all depends on the scope of the application [1].

4 Advantage(s) over Deterministic Counterpart(s)

In general, HyperLogLog provides an accurate estimation for the cardinality of a set. For applications where the exact cardinality is necessary, HLL cannot be used, but for all other applications where a strong estimate will suffice, HLL is a great tool to use. HLL has advantages over it's deterministic counterparts in two areas, space complexity and time complexity:

4.1 Space Complexity

HyperLogLog is extremely space efficient. It's deterministic competitors, namely a HashSet, have $O(n)$ space complexity because we iterate over the list and each element must be stored in the list. With HLL, we don't store each element like we do in a HashSet. Rather, we are only responsible for storing the maximum number of leading zeros (for each subset) for the hashed values from the inputted multi-set. Storing this amount of data is bounded by $O(log(log(n)))$ [1,2].

To illustrate the strength of this space complexity, let's consider a HashSet that requires 10 MB of space to store n elements. Using HyperLogLog to produce the cardinality estimate on the same set of n elements requires only 0.845 Bytes of memory space. This over 10,000,000x less space than using a HashSet. Obviously, we couldn't just use HLL if we are interested in more than a cardinality estimate, but for it's only function, it does a very good job in terms of space complexity.

4.2 Time Complexity

The time required to determine the exact cardinality of a multi-set deterministically is bounded by $O(n)$. This is because, for each new element we're counting, we have to check to see if it's distinct or not. This is pretty bad if you think about it, because, for very large data sets, the algorithm won't be able to finish in a reasonable time. We don't want the time required to run our algorithm to be proportional to the amount of data, especially for large data sets.

HyperLogLog, on the contrary, has $O(1)$ time complexity based on an analysis of the operations in the algorithm. It actually has $O(m)$ complexity, where m is the number of sub-streams we have, but since m is a constant, we simplify it to $O(1)$. Having a constant run-time makes HLL applicable to use on very large data sets, on which deterministic algorithms would be rather slow [1,2]. This is the greatest strength of HLL.

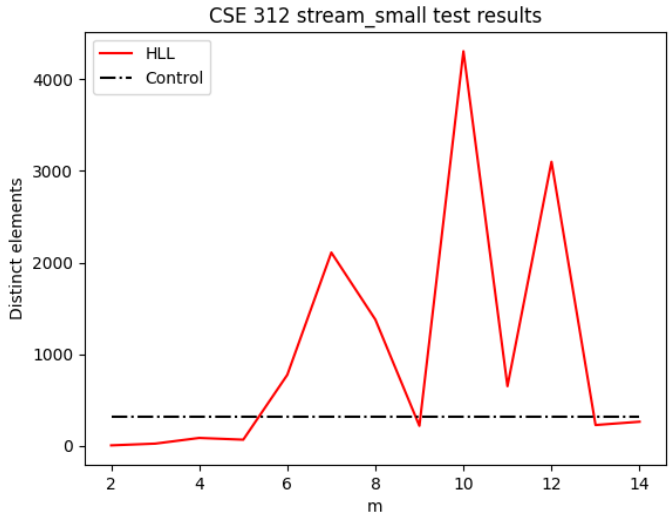
It's important to note that although HLL has constant time complexity, it can still become slow if m becomes big. In fact, as m approaches the size of the input stream, the time complexity approaches $O(n)$. So, a compromise must be made based on the required accuracy of the program. This flexibility can be nice though, as the implementer can adjust m to suit their needs.

5 Implementation in Python

Not Included.

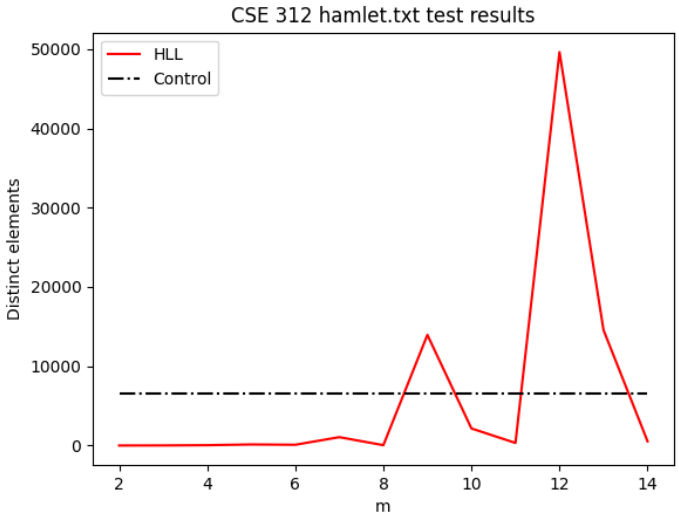
6 Visual Results and Analysis

The estimation capabilities of the HyperLogLog were tested on the small dataset of roughly 23000 numbers of which 312 numbers were unique, as illustrated by the dotted line titled control.
The HyperLogLog was tested on a variety of m values, to see which m parameter fit the estimation the best. As seen in the graph, the m value that best fit the actual estimate lied around $m = 6, 9, 13$.



CSE 312 stream_small.txt results

Similarly, the aforementioned test was performed using the CSE 143 Huffman Coding hamlet.txt file to search for an estimate of the number of distinct words that appear in the text. Illustrated below are the variations in the HyperLogLog estimate in comparison to the control value of the actual count obtained by placing all elements into a set and retrieving the size of the set.



As noticed, for larger m values the estimation capabilities of the HyperLogLog tend to improve in comparison to values of $m \leq 5$ and $m \leq 8$ respectively in order of the results illustrated. Verifying the cardinatlity of larger sets of elements using the HyperLogLog would require larger computation power. Hence the algorithm was not tested for larger sets. However, from the graphs, it can be determined that while greater m values for these sets may result in larger estimates, for larger sets, the variance between the actual cardinality of the set and the estimated cardinality using the HyperLogLog would differ less as the number of distinct elements in the set increases for larger data sets.

Thus, it can be inferred that the HyperLogLog serves as an excellent estimator of the cardinality of sets in terms of the efficiency of the algorithm for larger sets whose cardinality may not be computed using traditional means due to the computational expense associated with performing such an operation.

For large data sets such as monitoring internet traffic or credit card fraud, the HyperLogLog serves as an excellent data structure to estimate cardinality.

7 Probabilistic Analysis

As mentioned in the 'Core Functionality' section of the paper, Z is the harmonic mean of the counters, also known as the indicator function. In this section, we will compute the expected value for Z given an ideal multiset with cardinality n [2].

We start by letting M be a multi-set of cardinality n. We deduce that $Max(M) = max_{x \in M} p(x)$ is the maximum of n independent random variables, where $p(x)$ is a function that computes the number of leading zeros. Each of the n random variables, X, are randomly distributed according to a Geometric distribution because we consider observing the first '1' to be our first success. Thus, $P(X \geq k) = 2^{1-k}$ for $k \geq 1$. The probability that the multi-set is equal to some particular value, k, is the following: $P(M == k) = (1 - \frac{1}{2^k})^n - (1 - \frac{1}{2^{k-1}})^n$, because we consider the pdf of the geometric distribution for all of the individiual sub-sets [2].

Now let the multiset be split into m sub-sets of random cardinalities $M^1, ..., M^m$.

Thus,

$$E_n(Z) = \sum_{k_1,...,k_m} \frac{1}{\sum_{i=1}^{m} 2^{-k_i}} * \sum_n \binom{n}{n_1,...,n_m} * \frac{1}{m^n} * \prod_{i=1}^{m} (1 - \frac{1}{2^k})^n - (1 - \frac{1}{2^{k-1}})^n \text{ [2].}$$

for $m, k \geq 1$.

The variance of the HyperLogLog is experimentally determined to be $\frac{1.30}{\sqrt{m}}$ for large data sets by Meunier, Frédéric, et al. By applying bias correction such that the largest 30% of elements are removed followed by removing the smallest 30% of elements from the altered set, the variance is improved to $\frac{1.03}{\sqrt{m}}$ determined through experimentation as outlined in the same paper. Although sorting the large set of elements would be expensive, applying bias correction to the set of registers in the HyperLogLog would only require sorting $log_2(log_2(n))$ elements in a set of n elements. Hence the data structure earns the name HyperLogLog.

References

[1] Heule, Stefan, Marc Nunkesser, and Alexander Hall. "HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm." Proceedings of the 16th International Conference on Extending Database Technology. 2013.

[2] Meunier, Frédéric, et al. "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm." Discrete Mathematics & Theoretical Computer Science (2007).

[3] Nadalin, A., 2020. My Favorite Algorithm (And Data Structure): Hyperloglog. [online] Odino.org. Available at: https://odino.org/my-favorite-data-structure-hyperloglog/ [Accessed 17 August 2020].

[4] Lopes, J., 2020. How Does The Hyperloglog Algorithm Work?. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/12327004/how-does-the-hyperloglog-algorithm-work [Accessed 15 August 2020].