HyperLogLog Executive Summary
Dillan Khurana, Eric Wolpert, Sebastian Wiktorowicz
04/22/2025

<u>Introduction / Problem Statement</u>:

With an ever-growing amount of data, even simple tasks such as counting can be memory-intensive. Consider the task of a website like Reddit counting the views on a post. If a user revisists the same post, they should not increment the view counter. However, in order for Reddit to verify whether a given visit is new or recurring, they must keep a record of every single user who has visited the post in the past. This would be extremely memory-intensive! Reddit gets over 100 million daily active users, who each view multiple posts a day and each have 4 byte usernames that must be stored separately for every single post they visit.

This problem is known as the count distinct problem and it extends to websites tracking unique visitors, databases tracking unique queries, and networks tracking unique IP addresses. However, the upshot is that most of these values don't need to be *exact.* If Reddit displays that a post has 1000 views, when in actuality it has 990, the platform is virtually the same. So, we would be willing to sacrifice some amount of accuracy to substantially reduce the space complexity of counting unique elements.

This is what French computer scientist Philippe Flajolet sought to accomplish in 1984 with the Flajolet-Martin algorithm: maintain the same time complexity as the naive approach, while *substantially* reducing space complexity for large streams of input, and maintaining a reasonable low margin of error for estimates. Flajolet later built on this foundation with the LogLog algorithm in 2003. In the same paper, Flajolet and Marianne Durand introduced the SuperLogLog algorithm and in 2007, Flajolet and others introduced the HyperLogLog algorithm, providing mathematical optimizations for reducing margin of error with no added space or time complexity.

In particular, the HyperLogLog algorithm takes a multiset as input, and returns an estimated count of the number of distinct elements in the multiset. It achieves a time complexity which reduces O(n) for n items in the multiset (the same as the native hash-set approach). It achieves a space complexity of O(m) for m registers, an extraordinary improvement from the O(n) space complexity of the naive approach. And finally, it achieves a standard error of 1.04/sqrt{m}. As a result, the exact space complexity and error depends on the exact implementation of the algorithm, but as an example, Redis' (not Reddit's) HyperLogLog implementation uses up to 12 KB and provides a standard error of 0.81%. And this 12 KB is fixed and reliable, rather than variable based on input size!

In this summary, we'll explore how this algorithm works at a high-level, delve deep into its implementation, and look into a neat programming challenge which uses the algorithm with a special twist.

Overall Intuitive Approach / Solution:

At a high level, the algorithm passes through every element in the multiset, hashes it into a bitstring, and then cleverly uses probability to make an estimate of the multiset's cardinality merely using the number of leading zeros in the bitstring.

Beginning with the hash function, we necessitate that the hash function produces uniformly distributed output. Think of this as treating each bit as a coin flip that has an equal chance of heads and tails. Therefore, every bitstring is just a sequence of coin flips, each with an equal probability of occurring.

So, we go through the multiset, passing each value into our designated uniformly distributing hash function. Then, we take bitstring output and split the bits into a "bucket segment" and an "offset". The motivation behind using buckets is to aggregate different predictions together to get a more reasonable estimate. If we let one singular power of 2 prediction dictate our estimate, then we'd have extremely high error. So, we designate some number of buckets m and let the first log_2(m) bits denote which bucket has its prediction updated from our offset.

The prediction itself is then made using the offset bits. We begin by counting the number of leading zeros of the offset. So, for example, the segment 00100 contains 2 leading zeros and the segment 00001 contains 4 leading zeros. Now, here's the clever part. Going back to the analogy of coin flips, if we have 4 consecutive leading zeros, followed by a 1, that's as likely as flipping 4 heads followed by a tails, which would be 1/(2^5). And based on probability theory, we can therefore expect to flip the sequence of coins 2^5 times before we get that result. So, the fact that we found a hash with a 1 in the 5th position (1-indexed) suggests that we have hashed, on average, 32 values.

However, our bucket technically doesn't predict 32. It stores some maximum 1 position and then compares our current offset to that maximum. Let's say a few elements ago, we found a sequence with a 1 in the 7th position. Then, the bucket will keep its max of 7 and ignore the 5. Notice that the bucket needs very little storage–just enough bits to store whatever the max will be. This is logarithmic with respect to the actual max number, which itself is logarithmic with respect to the number of elements in the multiset. Therefore, we have a space complexity of O(log(logn)), hence the name Hyperloglog.

Each bucket's maximum is then aggregated with a harmonic mean. We use a harmonic mean rather than an arithmetic mean since the distribution of predictions is heavy-tailed. Consider the fact that half of the maxes will be 1s, a quarter 2s, but after a thousand elements we might get an outlier like 20 or 25 which, when treated as an exponent, will drastically overestimate our estimate. A harmonic mean weighs down these larger values to compensate.

We also then must multiply by m to account for the fact that each bucket will only estimate its share of the cardinality, since we chopped the bucket bits off of the actual hashed input.

Finally, we must multiply by some constant alpha. Through mathematical analysis too advanced for an undergraduate algorithms class, it was found that we consistently overestimate the actual cardinality by a fixed factor (as a function of m). So, we simply multiply by alpha to counteract that bias.

Implementation:

As mentioned earlier, we start the algorithm by converting all items to their bit representation, which is typically a 64-bit representation. In order to get this in some language like C++, we take each element in the multiset and hash it using a SHA1 algorithm which results in a 160 bit sequence. Because we only end up using 64 bits, we actually take the lower order 64 bits of this hash to get *mostly* unique 64 bit strings associated with each element. In languages like python however, we simply use the built in 64-bit hash function.

Another key part of this algorithm is choosing a *p* value. This p value corresponds with the number of buckets *m*, where m is equal to 2^p. A typical p value that this algorithm uses is 16, which results in 65536 buckets. A key point to notice here is that each bucket can be uniquely identified by a sequence of bits of length p. In this case, each of the 65536 buckets can be represented with 16 bits. In addition to this, the algorithm decides on specific alpha constants based on the number of buckets, and this is used later when combining buckets.

After hashing all the elements and determining the number of buckets that your algorithm will use, we must divide up the bit sequence to use in the different parts described earlier in the paper. Because p bits can be used to represent a unique register, we take the higher order p bits from the bitstring of length 64 to represent the "bucket segment" and we take the remainder of the bits to represent the "offset." Based on this segmentation, every time we hash and prepare a new bitstring, we will simply drop it into the "bucket segment" represented by its higher order p bits.

The next step within each bucket is actually utilizing the "offset" to compute a meaningful value for finding the number of distinct elements. As mentioned earlier, we will use the number of leading zeros in the “offset,” and the implementation of the algorithm within each bucket is actually quite intuitive. In each bucket, we store a single integer value that represents the maximum number of leading zeros encountered within this bucket. This is stored as an array or list of size m and is initialized with zeros before the algorithm is run. As each element is read or inserted into its respective bucket, we get the number of leading zeros of the offset, compare it to the current maximum number of leading zeros, and update it if the current number of leading zeros is greater. This is all done in constant time for each insertion.

After all of the elements from the multi-set are inserted, we have the maximum number of leading zeros that each bucket encountered in an array of length m. Now, the results of each of the buckets can be combined to estimate a cardinality of the entire set. To do this, we begin by taking the harmonic mean Z*m of each bucket’s estimate. We use this harmonic mean because this is a probabilistic solution that reduces the variance across the buckets. Additionally, we record a variable *V* that stores the number of buckets that still have a 0 in them, which can be used for optimization later.

$$Z = \left(\sum_{j=1}^{m} 2^{-M[j]} \right)^{-1}$$

(where m*Z is the harmonic mean)

Next, since the harmonic mean only gives the estimate *for each bucket*, we need to multiply by m to get the estimate for the entire multiset. Then, we multiply by the alpha we calculated earlier to counteract the fixed mathematical bias.

$$E = \alpha * m^2 * Z$$

After calculating this raw estimate, there are two edge cases that can be considered to get a better estimate. Firstly, if this raw result is less than the 2.5 * the number of buckets, then we can adapt the estimate formula to be the number of buckets times the log of the number of buckets divided by the number of buckets that have a zero (V) rounded up to the nearest integer.

$$E = m * log(\frac{m}{V})$$

On the other hand, if the raw result is very large (> (2^64) / 30), then we can adapt the raw estimate formula to be as follows (typically rare):

$$E = -2^{64} * log(\frac{1-E}{2^{64}})$$

Now that we have the code for initializing and inserting into our algorithm, we can add an additional helper method for merging separate HyperLogLog algorithms together. This is a relatively simple algorithm that can be thought of as dumping one algorithm's buckets into the corresponding buckets of another, and taking the max of the maxes.

Summary of Programming Challenge:

To test the student's knowledge of the HyperLogLog problem, we will have the student solve a problem involving pokemon collections and their value in a virtual pokemon card convention.

In this problem, you're presented with a set of vendors all selling their own trading card collections. Since your goal is to complete your collection and you only need one of each card, you want to maximize the number of distinct cards that you can get. This is where the count-distinct problem comes in. You need to HLL merge your current collection with each vendor's collection and then HLL count the number of distinct cards in the unioned collection.

These collections get very large, so you must use HLL as opposed to a naive solution to stay within memory constraints.

The twist is that each collection has a price and you have a budget constraint. You're not just finding the number of cards you get from each collection, but solving a maximization problem where you decide the set of collections to purchase to grow your own collection the most.

Key Ideas for Solving Programming Challenge:

This algorithm leans on 3 main parts to create a working implementation. The student should create a working HyperLogLog class, a merge operation for merging buckets from two different HyperLogLog objects, and an optimization for using their budget, like knapsack to arrive at their final result. Figuring out that this problem is designed for HyperLogLog is not difficult, but using those results effectively can pose to be more of a challenge.

Because the implementation of HyperLogLog already includes the merge operation and there are not many methods in the object, I think that asking the student to find and utilize this method when finding unique elements is fair. It allows the student to have to think beyond the scope of just implementing the data structure and have to apply the operations in multiple instances.

The twist of this problem can be quite challenging, but it is on par with the twists that the previous programming assignments have had this semester. The key to solving the optimization of budgeting is to use the knapsack problem, which can pose a challenge as this is an NP-complete problem. However, the knapsack problem is a relatively well-known algorithm that the student likely knows and the writeup for the assignment notes that the student should take a look at it. Although the implementation for this problem has dynamic programming, there are tons of implementations of this online that the student can access for this problem. Additionally, the difficulty and complexity of this problem is similar to that of the Transformation homework as we ask the student to use a data structure learned in class (segment trees) and then apply this data structure to a well known problem (simplex in the case of Transformation).

Conclusion:

HyperLogLog is an extremely effective solution to the count distinct problem, offering exceptionally low space complexity with little margin of error as a cost. As a result, it continues to be used by Google Analytics, Redis, Presto (by Meta), and many more platforms. Extracting probabilistic estimates from hashed bitstrings is unlike any other algorithm we've studied in class and is truly clever. As data volume continues to grow, it's evident that HyperLogLog will remain essential.