The Count Distinct Problem

Steven Rosendahl

Processing data efficiently has always been a goal of organizations that collect data; companies like Facebook and Google have been collecting and storing data for as long as they have existed. One important statistic that companies can use is the number of distinct elements in a large set of data. The count-distinct problem poses the question of how many distinct elements are in a given set. There are several algorithms that exist that can solve the count distinct problem; we will consider three of them: the hash table, the Linear Probabilistic Counter, and the HyperLogLog. In order to analyze these algorithms, we will apply them to real world problems, namely (1) The Pokémon Problem, which poses the question "How many unique Pokémon encounters will a player face in a given play through of a game from each generation?," (2) The Facebook Problem, which asks "How many unique Facebook app installs are made on any given day?", and (3) The Twitter Problem, where we will determine how many unique hashtags are created on any given day. We can use the hash table, the Linear Probabilistic Counter, and the HyperLogLog to solve each problem, respectively.

The first solution to the count distinct problem is called a hash table. A hash-table is a matrix that is stored in memory where each entry was uniquely obtained from another value, called a key. The algorithm is as follows:

Let $\mathbb{S}$ be a set of random elements. In order to count the number of distinct elements in $\mathbb{S}$,

1. Apply a bijection, $h(n)$ to all $n \in \mathbb{S}$, and store the result in a set $\mathbb{V}$.
2. Count the number of elements in $\mathbb{V}$. $||\mathbb{V}||$ will be the number of unique elements in $\mathbb{S}$.

Since $\mathbb{V}$ is a bijection, it is surjective. As a result, if we take any $x \in \mathbb{S}$ and $y \in \mathbb{S}$, then if $x = y$, they will hash to the same value, and we can ignore it [8, p. 6]. For example, Say we want to count the number of distinct names in the set $\mathbb{S} = \{$"Alice", "Emily", "Alice"$\}$. We can create a simple hash function where we map each letter to its corresponding value in the alphabet, sum up total of all the letter values in a name, and mod that value with the number of letters in the name. Applying the function on each element of $\mathbb{S}$

yields a set $\mathbb{V} = \{0, 4\}$. $||\mathbb{V}|| = 2$, so there were two distinct elements in $\mathbb{S}$. In this example, the hash function we chose was bijective. However, this may not always be the case. In fact, it is practically impossible to create a hash function that is truly surjective for huge amounts of data [8, p. 6]. For smaller sets of data, however, we can create a hash function that is surjective. The pokémon problem provides a good example of the benefits of a hash table.

The most beneficial aspect of the hash is that it has a 0% error rate. In other words, the cardinality of $\mathbb{V}$ will be exactly how many distinct elements are in $\mathbb{S}$, assuming the hash function is surjective. We know that for very large $\mathbb{S}$, the hash table method is incredibly inefficient, but for the Pokémon problem, we are dealing with $\mathbb{S}$ small enough to apply the hash table. In this particular case, 721 pokémon can be encountered [7], which means that the maximum size needed for our set $\mathbb{V}$ is 721. Typically, a player will encounter in the neighborhood of 1000 wild pokémon a game, and with a total of 6 generations of games, we have a set $\mathbb{S}$ of size 6000. The hashing function we created earlier will not be good enough here; instead we can create a new hashing function where we sum up the value of the name as before, then add the pokémon's unique id, and finally mod that value with 721.

```
array.each do |pokemon|
    hash_table[pokemon.custom_hash] = pokemon.custom_hash
end
total = 0
hash_table.each do |val|
    if val != -1
        total = total + 1
    end
end
puts "The total number of unique encounters is #{total}"
```

Figure 1: Hashing Pokémon based on name and type

In order to come up with a set $\mathbb{S}$, we can randomly populate an array with 6000 pokémon; we are assuming that a player has an equal chance to run into any pokémon, and that they can encounter all pokémon more than once. Running the code in *Figure 1* gives us a final answer of 461 unique encounters with wild pokémon. In the case of the pokémon problem, a simple hash function sufficed, since the only collisions that were encountered were pokémon where the name was the same. If we consider a larger data set, however, we may not be able to create a surjective hash function.

Imagine that we are solving the Twitter problem where we want to count the number of distinct hashtags on twitter on a given day. Our set $\mathbb{S}$ will now contain every hashtag that was made on a given day. We know that we want a hash function that is surjective, but this is an unrealistic, since we could have some $x \in \mathbb{S}$ and $y \in \mathbb{S}$ where $x \neq y$ but $h(x) = h(y)$. To model this issue, we can consider the following scenario:

> On October 30th, 2015, Mojang released a special Halloween cape to all players of the popular video game Minecraft. As a result, "#cape" began trending on Twitter. On the same day, Target

announced that all their *Bob's Burgers* costumes would be on sale for the low price of $12.95, and "#bob" began trending on twitter.

Our hash function from earlier will not work in this case, since both "#bob" and "#cape" hash to the same value. On a larger scale, such as the Facebook or Twitter problem, we would need to either handle collisions or create a better hash function. Creating a way to handle collisions would not be beneficial, since the data is being processed in real time. We would not be able to tell if we were hashing a value we already have seen or hashing a new value that happens to hash to the same value as another element we have already seen; as a result we would have no way to know which keys were different and which keys were the same when looking at $\mathbb{V}$ [8, p. 6]. Since the collision policy would not help us, we need to come up with a better hashing function.

More complex hash functions, such as SHA-2 (Secure Hash Algorithm 2) or MD5 (Message Digest 5) offer a solution to the collision problem. SHA-2 is used by organizations such as the NSA to encrypt data, and has yet to produce any collisions when hashing values [2, p. 301]. However, this is not a viable solution to the count distinct problem either due to the volume of the data being processed. In both the Twitter and Facebook problems, we are analyzing exabytes of data. The SHA-2 function outputs a 512 byte long integer for every input, which means that the resulting size of the set $\mathbb{V}$ would be in the worst case $512 \times ||\mathbb{S}||$. All of $\mathbb{V}$ would have to be held in memory, which means that analyzing 50 million hashtags (which is a low estimate)[1] could possibly result in $(50 \times 10^{10}) \times 512$ bytes of data being stored at one time, assuming every hashtag was 10 bytes long. A hash table of this size could not be loaded all at once into main memory; it would have to be stored into secondary memory which would exponentially increase the amount of time required to read the data [12, p. 209]. MD5 outputs smaller sized integers (128 in this case), but the collision probability is much higher [11, p. 22 - 23]. The hash method is not a viable solution to the count distinct problem when we are dealing with a large volume of data, but hash functions are still used in the other solutions to the count-distinct problem.

The hash-table method provided a 0% error, but was not efficient in handling large sets of data. However, if we are looking for a *good enough* estimate (within the neighborhood of 1% to 5% error) of the number of distinct elements in $\mathbb{S}$, we can use an algorithm called a Linear Probabilistic Counter. With the Linear Probabilistic Counter, the user can specify how large they want the error to be. This method uses a data structure called a bitmap. bitmaps are a matrix where either a 1 or 0 is stored in a cell, which allows for the map to be incredibly space efficient. The Linear Probabilistic Counter also uses a hash, but it only needs to temporarily produce a position and then it can "forget" the value it just produced, which in turn reduces the overhead memory needed to store the elements that was required by the hash table method [12]. The number of distinct elements is determined by the number of 1's in the bitmap when the algorithm is finished.

The Linear Probabilistic Counter algorithm is divided into several steps. In the first step, the algorithm creates an arbitrarily sized bitmap in memory, and then hashes the values into the bitmap by determining the address in the bitmap at which the corresponding bit should be changed to a 1. In the next step, the algorithm counts the number of 0's in the bitmap, and stores the result. Finally, the algorithm estimates the cardinality of the unique set by dividing the count of 0's by the size of the bitmap, and then substituting in the result to

$$\hat{n} = -m \ln(\Lambda_n),$$

where m is the size of the bitmap and Λ_n is the ratio [12]. Figure 2 shows the Ruby implementation of the Linear Probabilistic Counter algorithm.

```ruby
def count(m, s)
    bitmap = Array.new(m)
    i = 0
    for i in 0..m do
        bitmap[i] = 0
    end
    s.each do |value|
        address = m_hash(value, m)
        bitmap[address] = 1
    end
    zeros = 0.0
    bitmap.each do |one_or_zero|
        if one_or_zero == 0 then
            zeros += 1.0
        end
    end
    lamb = (zeros / m)
    cardinality = (-1 * m) * Math.log(lamb)
end
```

Figure 2: Determining the cardinality

The Linear Probabilistic Counter algorithm also allows us to specify the amount of error we are willing to tolerate. This is accomplished by changing the size of the bitmap; if the bitmap is large, or close to the size of the original set $\mathbb{S}$, then the error will be small. Alternatively, if we decrease the size of the bitmap, we use less memory, but the error grows exponentially [6].

We can use the Linear Probabilistic Counter algorithm to solve the Facebook problem; unfortunately, we have no way of accessing the actual data we need, so we will have to do some guessing. We know that Facebook users make about 7,000,000 app installs a dat [9]. This means that the size of the set $\mathbb{S}$ will be 7,000,000. Now we need to make some guesses; we expect that the applications that are most installed will appear a lot in our set, and that the applications that are installed less will appear less in $\mathbb{S}$. We do have access the to the top 20 Facebook application installs in August 2015, so we can plot those points and look for a trend in the data. Doing so tells us that the data follows an exponential curve [9], which means that Spotify (the top application at the time) will appear frequently in $\mathbb{S}$, whereas LINE, the 20$^{\text{th}}$ most popular Facebook app, will appear sparingly. This means that we will need a bitmap relatively close in size to the original set, since only a select few of the application have a high appearance rate in $\mathbb{S}$. In other words, we expect a lot of distinct elements in $\mathbb{S}$.

We can model the exponential function by choosing random numbers with a certain weight towards different numbers. We will let the each number correspond to a different application. For example, if we let

200 be the number for Spotify, we would expect a high appearance rate of 200 in $\mathbb{S}$. We can use an algorithm (courtesy of GitHub user Danieth5x) that will generate a pseudo-random sample based on a function, which will be an exponential function in our case. We can argue that it is not important as to which exponential function we choose, since the distribution will be more or less the same; in our case we will pick e^x. Running the pseudo-random generator produces a set $\mathbb{S}$ of size 7,000,000, and we can now apply the Linear Probabilistic Counter to the data set. For our purposes, we will use the hash function supplied by Ruby; the only change we need to make is to mod the result of the Ruby hash with the number of spaces in the bitmap, which ensures that we will never produce a hashed value out of the bounds of our bitmap. Finally, running the algorithm tells us that the total number of unique application installs is 3216.

The Linear Probabilistic Counter and the hash table algorithms handled relatively small data sets well, but the Twitter problem deals with a data set of nearly 200,000,000 tweets a day [10]. As a result, the Linear Probabilistic Counter would be too inefficient to process all the data, but the HyperLogLog algorithm allows for data to be processed more efficiently than the Linear Probabilistic Counter or the hash table. The HyperLogLog algorithm actually uses parts of the hash table and Linear Probabilistic Counter algorithms; values are still hashed to a bitmap held in memory. The hash function hashes values to binary numbers, and the algorithm decides whether or not that value has already been seen the value before by analyzing the leading 0's in the binary number [5, pp. 685, 689]. In other words, the algorithm keeps a record of the first location of a 1 in a binary string, called ρ [4, pp. 130]. If the sequence of 0's is identical to another sequence of 0's, the probability that the original values were the same is very high, and the algorithm will ignore what it finds to be a duplicate value. The amount of memory that the HyperLogLog algorithm requires per bitmap is expressed by

$$\text{Memory Required } = \log_2\left(\log_2\left(M\right)\right),$$

where M is the size of the original set of data [4, pp. 129]. The error amount can be found by

$$\text{Error } = \frac{\sqrt{3\log\left(2\right) - 1}}{\sqrt{m}},$$

where m is the number of spaces in the bitmap. Finally, the algorithm takes the harmonic average of all the totals of the separate bitmaps, given by

$$H = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}},$$

which allows the algorithm to increase in accuracy as the size of the original set grows [13].

We can solve the Twitter problem with the HyperLogLog algorithm. We will use the code in *Figure 3* to aggregate a random sampling of the most recent 2000 tweets that contain a "#" every two minutes for 24 hours. This will allow for a large random sample of tweets that will closely model the real problem that we are trying to solve. We also need a hashing function that will produce a binary string of 1's and 0's given any key; luckily this hash function already exists. A murmur hash will produce a 32 bit binary string, which is exactly what we need. Finally, we actually need to implement the algorithm. We will create a class to perform the calculation that is initialized with the amount of memory we are willing to spend on the bitmap for the HyperLogLog algorithm. We know that the HyperLogLog can process nearly 1 billion elements with only 1.5 kilobytes of space to an accuracy of 2% [4]. This is given by the equation we have for memory. Using a similar argument, we can determine the amount of memory we will need.

If we want the error to be 1%, then we can calculate the number of spaces required in the bitmap by using the equation we have for the error.

```
while true do
    open('twitter_data.txt','a') do |f|
        client.search("#", result_type: 'recent', lang: 'en').take(2000).each do |tweet|
            twt = tweet.text
            array = twt.split(' ')
            array.each do |val|
                if val[0] == "#" and val.length > 1 then
                    f.puts val
                    tags += 1
                end
            end
        end
    end
```

Figure 3: An infinite while loop can be useful

$$
\begin{aligned}
0.01 &= \frac{\sqrt{3\log(2) - 1}}{\sqrt{m}} \\
0.01\sqrt{m} &= \sqrt{3\log(2) - 1} \\
\sqrt{m} &= \frac{\sqrt{3\log(2) - 1}}{0.01} \\
m &= \frac{\sqrt{3\log(2) - 1}}{0.01^2} \\
m &\approx 10,794.4 \text{ spaces in the bitmap.}
\end{aligned}
$$

Now we can find the total memory required for our logarithm. If we let the program from *Figure 1* run for 24 hours, at worst it will parse 1,440,000 hashtags. We can now calculate the memory required by

$$
\begin{aligned}
\text{Memory } &= \log_2(\log_2(1,440,000)) \\
&= 4.35457 \,\text{bytes per entry.}
\end{aligned}
$$

Multiplying this number by the total number of entires gives us a grand total of 47005 bytes, or 4.7005 kilobytes of required memory.

We now have all the necessary information to parse the data. We will implement the HyperLogLog in an object oriented manner; that is, we will create a Ruby class called HyperLogLog that we can initialize with the number of bytes per entry. We also need a method to calculate the unique elements, and we need a way to actually call this routine.

```ruby
def initialize(log2m, register_set = nil)
    @log2m = log2m
    @count = 2 ** log2m
    @register_set = register_set || RegisterSet.new(@count)

    case log2m
    when 4
        @alphaMM = 0.673 * @count * @count
    when 5
        @alphaMM = 0.697 * @count * @count
    when 6
        @alphaMM = 0.709 * @count * @count
    else
        @alphaMM = (0.7213 / (1 + 1.079 / @count)) * @count * @count
    end
end
```

Initialization

```ruby
def cardinality
    register_sum = 0.0
    zeros = 0.0
    @register_set.each do |value|
        register_sum += 1.0 / (1 << value)
        zeros += 1 if value == 0
    end
    estimate = @alphaMM * (1 / register_sum)
    if estimate <= (5.0 / 2.0) * @count
        (@count * Math.log(@count / zeros)).round
    else
        estimate.round
    end
end
```

Counting

```ruby
def Worker.calculate(len)
    mhll = Hyperll::HyperLogLog.new(4)
    File.open("twitter_data.txt","r") do |file|
        file.each_line do |line|
            mhll.offer line
        end
    end
    str = "\tUnique Elements: #{mhll.cardinality}"
    File.open("tw_log.txt",'a') do |file|
        file.puts str
        puts str
    end
end
```

Implementing

Note that we settled with 4 as the size of each entry in the bit-map. This value is close enough to 4.35 to safely say that the error will be just about 1%. Finally, in our main program, we will loop infinitely (it will be up to us as to when to kill the process), and read in $\sim$ 2000 tweets every two minutes. After running the algorithm for 24 hours, we will have a set of data that represents the number of unique hashtags found (by using the HyperLogLog algorithm) as the number of tweets being analyzed increases. The data grows in a linear fashion; this is expected since we are gathering a random sample from all over the world, which means that at any given time, we expect to see more or less the same amount of unique hashtags. Plotting the data and applying a best fit line using Mathematica gives us

$$f(x) = 0.284356x + 7,361.39.$$

From this data, we can make an estimate that at 200,000,000 tweets, we will have

$$\begin{aligned} f(200,000,000) &= 0.284356(200,000,000) + 7,361.39 \\ &= 56,878,500, \end{aligned}$$

which means that we will have roughly 56 million unique hashtags by the HyperLogLog algorithm.

We have looked at three different approaches to the count distinct problem. We used the HyperLogLog to solve only the Twitter problem, but we could have used the algorithm to solve all of the problems. The count distinct problem is heavily applicable in the real world; companies are able to turn the numbers that are determined by algorithms such as the HyperLogLog to increase profits, target ads, and keep more data. Finding an efficient way to count large sets of data may never be competely accurate, but with algorithms like the HyperLogLog and the Linear Probabilistic counter, we can efficiently and with nearly 100% accuracy solve the count distinct problem.

References

[1] #numbers. (2011, March 14). Retrieved October 29, 2015, from https://blog.twitter.com/2011/numbers

[2] Chaves, Ricardo, Kuzmanov, Georgi, Sousa, Leonel & Vassiliadis, Stamatis (2006). Improving SHA-2 Hardware Implementations. , 4249, 298-310.

[3] Facebook Statistics. (2015, September 20). Retrieved October 30, 2015, from http://www.statisticbrain.com/facebook-statistics/

[4] Flajolet, P., Fusy, É., Gandouet, O., & Meunier, F. (n.d.). HyperLogLog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm. 2007 *Conference on Analysis of Algorithms, 07,* 127-143. Retrieved October 27, 2015, from http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf

[5] Heule, S., Nunkesser, M., & Hall, A. (2013). HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology* (Vol. EDBT '13, pp. 683-692). New York, NY, Genoa: ACM.

[6] Hoff, T. (2012, April 5). Big Data Counting: How to count a billion distinct objects using only 1.5KB of Memory - High Scalability -. Retrieved October 27, 2015, from http://highscalability.com/blog/2012/4/5/big-data-counting-how-to-count-a-billion-distinct-objects-us.html

[7] How many Pokemon are there altogether, before and after Omega Ruby and Alpha Sapphire? (2014, October 27). Retrieved October 29, 2015, from http://pokemondb.net/pokebase/219332/pokemon-there-altogether-before-after-omega-alpha-sapphire

[8] Maurer, W. D. & Lewis, T. G. (1975). Hash Table Methods. ACM Comput. Surv., 7, 5-19.

[9] Most popular Facebook apps 2015 — Statistic. (2015, August 1). Retrieved November 3, 2015, from http://www.statista.com/statistics/276371/most-popular-facebook-apps-by-monthly-active-users/

[10] Twitter Statistics. (2015, September 15). Retrieved November 3, 2015, from http://www.statisticbrain.com/twitter-statistics/

[11] Wang, Xiaoyun & Yu, Hongbo (2005). How to Break MD5 and Other Hash Functions. , 3494, 19-35.

[12] Whang, Kyu-Young, Vander-Zanden, Brad T. & Taylor, Howard M. (1990). A Linear-time Probabilistic Counting Algorithm for Database Applications. ACM Trans. Database Syst., 15, 208-229.

[13] Yousra Chabchoub, Georges Hébrail. *Sliding HyperLogLog: Estimating cardinality in a data stream.* 2010.