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. The count-distinct problem poses the question of how many distinct elements are in a given set. The three problems that we will analyze are

1. The Twitter Problem: How many unique hashtags are created on any given day?
2. The Facebook Problem: How many unique Facebook app installs are made on any given day?
3. The Pokémon Problem: How many unique Pokémon encounters will a player face in a given play through of a game from each generation.

Each of these problems can be solved by using one of three different solutions to the count distinct problem: the hash table, the linear probabilistic counter, and the hyperloglog algorithm.

The count-distinct problem has not always been a problem. Before the internet was widely used by the population, storing data was expensive, and as a result most companies did not want to spend money on the hardware and space that would be required to store large amounts of data. In addition, data was not transferred and gathered at a fast enough rate to warrant finding a way to count the data; even if companies wanted to count the data, the computers at the time were not powerful enough to perform the necessary calculations. The problem of counting big data did not arise until the 1990's, when the internet began to facilitate the easy transfer and collection of data. Today, more than 25 exabytes of data are created and stored a day. The main factor that contributes most to the problem is that gathering and storing data is cheap, but analyzing it is not. The volume of the data leads to the count distinct problem that we will analyze [11, pp. 1].

The first solution to the count distinct problem is called the *Hash Table*. A hash-table is a matrix that is stored in memory where each entry was uniquely obtained from another value. 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 [3, 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 [3, 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 [8], 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. The following Java code can be used to model our hash table:

```
public int getHashValue(){
    char[] array = nameOfPokemon.toCharArray();
    int total = 0;
    for(char c : array){
        total += (int)c;
    }
    total += type;
    total %= 721;
    return total;
}
```

```
...
for(Pokemon pm : S){
    int hashValue = pm.getHashValue();
    storeHashValue();
}
calculateAndPrintUniqueValues();
```

Running this code will give us a final answer of 458 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 $\mathbb{V}$ [3, 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 [4, 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)[7] 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 [2, p. 209]. MD5 outputs smaller sized integers (128 in this case), but the collision probability is much higher [5, 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 LPC, the user can specify how large they want the error to be. This method uses a data structure called a bit-map. Bit-maps 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 LPC 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 [2]. The number of distinct elements is determined by the number of 1's in the bit-map when the algorithm is finished.

In order to either increase or decrease the error, the user of the LPC algorithm can either increase or decrease the size of the bit-map that is held in main memory. The following equation gives us a relationship between the size of the bitmap and the error that will be produced:

$$bias\left(\frac{\hat{n}}{n}\right) = \frac{e^t - t - 1}{2n},$$

where $bias\left(\frac{\hat{n}}{n}\right)$ is the error. The value t is the ratio of the number of distinct elements determined by the algorithm (denoted by n) to the size of the bitmap [2]. If t is close to 0, then the error will be very small; if t is much greater than 1, the error will be very large. In other words, if the hash function being used produces a large amount of collisions, then the error will be high. Luckily, an algorithm such as SHA-2 will be okay to use, since the LPC never stores the hashed value in main memory. We can also calculate the number of distinct elements by

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

where m is the size of the bit-map and Λ_n is the ratio of the number of 0's in the bit map to the size of the bit map [2, p. 212].

The Facebook problem can be solved efficiently by using the LPC algorithm. We would have set $\mathbb{S}$ of size 20 million [1], and we can choose an arbitrary sized bit map $\mathbb{V}$ of 10 million bits, or roughly 1.25 Mb, which lies on the upper limit of the efficiency of the LPC. If the memory required for the bit map increases beyond 1.25Mb, most operating systems will move the I/O operations to secondary memory, which defeats the purpose of the LPC algorithm [2, p. 211]; as a result, the Twitter problem would not be efficiently solved by this method, since the amount of hashtags created in a day far exceeds 20 million. In the case of the

Facebook problem, we can apply the LPC algorithm analytically, rather than actually processing 20 million application installs.

We know that on a given day, Facebook users make 20 million app installs, and that there are currently about 7 million apps on Facebook [1]. However, not all 7 million applications are installed every day; in fact, the amount of installs decays exponentially starting from the number one most installed application on Facebook, which is Spotify as of August 2015 [9]. We would expect that apps such as Spotify would dominate the market, whereas apps in the lower spectrum of the downloads would be negligible.

The LPC 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 LPC would be too inefficient to process all the data, but the HyperLogLog algorithm allows for data to be processed more efficiently than the LPC or the hash table. The HLL algorithm actually uses parts of the hash table and LPC algorithms; values are still hashed to a bit map 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 [6, pp. 685, 689]. In other words, the algorithm keeps a record of the first location of a 1 in a binary string, called ρ [12, 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 HLL algorithm requires is expressed by

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

where M is the size of the original set of data [12, pp. 129]. Finally, the algorithm takes the harmonic average of all the totals of the separate bit maps, which allows the algorithm to increase in accuracy as the size of the original set grows [11].

We can solve the Twitter problem with the HLL algorithm. We will make the assumption that all 200,000,000 tweets made will have hashtags, which means we will have a set $\mathbb{S}$ of size 200,000,000.

By the formula for above, we get that the total required memory per bit map will be

$$\begin{aligned} \text{Memory} &= \log_2(\log_2(200,000,000)) \\ &\approx \text{b} \\ &\approx 1.79 \text{ Kb}, \end{aligned}$$

which is roughly the size of a one page Microsoft Word document. The LPC algorithm would take at least ten times the amount of space just to hold the bit map in main memory. In addition to the memory efficiency of the HLL, the algorithm also has a relatively smaller error than the LPC, expressed by

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

Finally, the HyperLogLog finds a harmonic mean of the hashed values in the bit map, rather than a geometric mean, which increases the accuracy of the HLL even more.

References

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

[2] 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.

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

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

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

[6] 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.

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

[8] 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`

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

[12] 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`