Big Data Counting: How to count a billion distinct objects using only 1.5KB of Memory

Thursday, April 5, 2012 at 9:15AM

Todd Hoff in Strategy

This is a guest post by Matt Abrams (@abramsm), from Clearspring, discussing how they are able to accurately estimate the cardinality of sets with billions of distinct elements using surprisingly small data structures. Their servers receive well over 100 billion events per month.

At Clearspring we like to count things. Counting the number of distinct elements (the cardinality) of a set is challenge when the cardinality of the set is large.

To better understand the challenge of determining the cardinality of large sets let's imagine that you have a 16 character ID and you'd like to count the number of distinct IDs that you've seen in your logs. Here is an example:

4f67bfc603106cb2

These 16 characters represent 128 bits. 65K IDs would require 1 megabyte of space. We receive over 3 billion events per day, and each event has an ID. Those IDs require 384,000,000,000 bits or 45 gigabytes of storage. And that is just the space that the ID field requires! To get the cardinality of IDs in our daily events we could take a simplistic approach. The most straightforward idea is to use an in memory

hash set that contains the unique list of IDs seen in the input files. Even if we assume that only 1 in 3 records are unique the hash set would still take 119 gigs of RAM, not including the overhead Java requires to store objects in memory. You would need a machine with several hundred gigs of memory to count distinct elements this way and that is only to count a single day's worth of unique IDs. The problem only gets more difficult if we want to count weeks or months of data. We certainly don't have a single machine with several hundred gigs of free memory sitting around so we needed a better solution.

One common approach to this problem is the use of bitmaps. Bitmaps can be used to quickly and accurately get the cardinality of a given input. The basic idea with a bitmap is mapping the input dataset to a bit field using a hash function where each input element uniquely maps to one of the bits in the field. This produces zero collisions, and reduces the space required to count each unique element to 1 bit. While bitmaps drastically reduce the space requirements from the naive set implementation described above they are still problematic when the cardinality is very high and/or you have a very large number of different sets to count. For example, if we want to count to one billion using a bitmap you will need one billion bits, or roughly 120 megabytes for each counter. Sparse bitmaps can be compressed in order to gain space efficiency, but that is not always helpful.

Luckily, cardinality estimation is a popular area of research. We've leveraged this research to provide a open source implementation of cardinality estimators, set membership detection, and top-k algorithms.

Cardinality estimation algorithms trade space for accuracy. To illustrate this point we counted the number of distinct words in all of Shakespeare's works using three different counting techniques. Note that our input dataset has extra data in it so the cardinality is higher than the standard

reference answer to this question. The three techniques we used were Java HashSet, Linear Probabilistic Counter, and a Hyper LogLog Counter. Here are the results:

Counter	Bytes Used	Count	Error
HashSet	10447016	67801	0%
Linear	3384	67080	1%
HyperLogLog	512	70002	3%

The table shows that we can count the words with a 3% error rate using only 512 bytes of space. Compare that to a perfect count using a HashMap that requires nearly 10 megabytes of space and you can easily see why cardinality estimators are useful. In applications where accuracy is not paramount, which is true for most web scale and network counting scenarios, using a probabilistic counter can result in tremendous space savings.

Linear Probabilistic Counter

The Linear Probabilistic Counter is space efficient and allows the implementer to specify the desired level of accuracy. This algorithm is useful when space efficiency is important but you need to be able to control the error in your results. This algorithm works in a two-step process. The first step assigns a bitmap in memory initialized to all zeros. A hash function is then applied to the each entry in the input data. The

result of the hash function maps the entry to a bit in the bitmap, and that bit is set to 1. The second step the algorithm counts the number of empty bits and uses that number as input to the following equation to get the estimate.

n=-m ln Vn

In the equation m is the size of the bitmap and Vn is the ratio of empty bits over the size of the map. The important thing to note is that the size of the original bitmap can be much smaller than the expected max cardinality. How much smaller depends on how much error you can tolerate in the result. Because the size of the bitmap, m, is smaller than the total number of distinct elements, there will be collisions. These collisions are required to be space-efficient but also result in the error found in the estimation. So by controlling the size of the original map we can estimate the number of collisions and therefore the amount of error we will see in the end result.

Hyper LogLog

The Hyper LogLog Counter's name is self-descriptive. The name comes from the fact that you can estimate the cardinality of a set with cardinality Nmax using just loglog(Nmax) + O(1) bits. Like the Linear Counter the Hyper LogLog counter allows the designer to specify the desired accuracy tolerances. In Hyper LogLog's case this is done by defining the desired relative standard deviation and the max cardinality you expect to count. Most counters work by taking an input data stream, M, and applying a hash function to that set, h(M). This yields an observable result of S = h (M) of {0,1}^∞ strings. Hyper LogLog extends this concept by splitting the hashed input stream into m substrings and then maintains m observables for each of the substreams. Taking the average of the additional observables yields a counter whose accuracy improves as m

grows in size but only requires a constant number of operations to be performed on each element of the input set. The result is that, according to the authors of this paper, this counter can count one billion distinct items with an accuracy of 2% using only 1.5 kilobytes of space. Compare that to the 120 megabytes required by the HashSet implementation and the efficiency of this algorithm becomes obvious.

Merging Distributed Counters

We've shown that using the counters described above we can estimate the cardinality of large sets. However, what can you do if your raw input dataset does not fit on single machine? This is exactly the problem we face at Clearspring. Our data is spread out over hundreds of servers and each server contains only a partial subset of the the total dataset. This is where the fact that we can merge the contents of a set of distributed counters is crucial. The idea is a little mind-bending but if you take a moment to think about it the concept is not that much different than basic cardinality estimation. Because the counters represent the cardinality as set of bits in a map we can take two compatible counters and merge their bits into a single map. The algorithms already handle collisions so we can still get a cardinality estimation with the desired precision even though we never brought all of the input data to a single machine. This is terribly useful and saves us a lot of time and effort moving data around our network.

Next Steps

Hopefully this post has helped you better understand the concept and application of probabilistic counters. If estimating the cardinality of large sets is a problem and you happen to use a JVM based language then you should check out the stream-lib project — it provides implementations of the algorithms described above as well as several other stream-processing

utilities.

Related Articles

- On HackerNews
- Fast, easy, realtime metrics using Redis bitmaps
