Reservoir Sampling - Sampling from a stream of elements

If you don't find programming algorithms interesting, stop reading. This post is not for you.

On the other hand, if you do find algorithms interesting, in addition to this post, you might also want to read my other posts with the algorithms tag.

Problem Statement

Reservoir Sampling is an algorithm for sampling elements from a stream of data. Imagine you are given a really large stream of data elements (queries on google searches in May, products bought at Walmart during the Christmas season, names in a phone book, whatever). Your goal is to *efficiently* return a random sample of 1,000 elements **evenly distributed** from the original stream. How would you do it?

The right answer is generating random integers between 0 and N-1, then retrieving the elements at those indices and you have your answer. (Update: reader Martin astutely points out that this is sampling with replacement. To make this sampling without replacement, one simply needs to note whether or not your sample already pulled that random number and if so, choose a new random number. This can make the algorithm pretty expensive if the sample size is very close to N though).

So, let me make the problem harder. You don't know N (the size of the stream) in advance and you can't index directly into it. You can count it, but that requires making 2 passes of the data. You can do better. There are some heuristics you might try: for example to guess the length and hope to undershoot. It will either not work in one pass or will not be evenly distributed.

Simple Solution

A relatively easy and correct solution is to assign a random number to every element as you see them in the stream, and then always keep the top 1,000 numbered elements at all times. This is similar to how mysql does "ORDER BY RAND()" calls. This strategy works well, and only requires additionally storing the randomly generated number for each element.

Reservoir Sampling

Another, more complex option is reservoir sampling. First, you want to make a reservoir (array) of 1,000 elements and fill it with the first 1,000 elements in your stream. That way if you have exactly 1,000

elements, the algorithm works. This is the base case.

Next, you want to process the i'th element (starting with i = 1,001) such that at the end of processing that step, the 1,000 elements in your reservoir are randomly sampled amongst the i elements you've seen so far. How can you do this? Start with i = 1,001. With what probability after the 1001'th step should element 1,001 (or any element for that matter) be in the set of 1,000 elements? The answer is easy: 1,000/1,001. So, generate a random number between 0 and 1, and if it is less than 1,000/1,001 you should take element 1,001. In other words, choose to add element 1,001 to your reservoir with probability 1,000/1,001. If you choose to add it (which you likely will), then replace any element in the reservoir chosen randomly. I've shown that this produces a 1,000/1,001 chance of selecting the 1,001'th element, but what about the 2nd element in the list? The 2nd element is definitely in the reservoir at step 1,000 and the probability of it getting removed is the probability of element 1,001 getting selected multiplied by the probability of #2 getting randomly chosen as the replacement candidate. That probability is 1,000/1,001 * 1/1,000 = 1/1,001. So, the probability that #2 survives this round is 1 - that or 1,000/1,001.

This can be extended for the i'th round - keep the i'th element with probability 1,000/i and if you choose to keep it, replace a random element from the reservoir. It is pretty easy to prove that this works for all values of i using induction. It obviously works for the i'th element based on the way the algorithm selects the i'th element with the correct probability outright. The probability any element before this step being in the reservoir is 1,000/(i-1). The probability that they are removed is 1,000/i * 1/1,000 = 1/i. The probability that each element sticks around given that they are already in the reservoir is (i-1)/i and thus the elements' overall probability of being in the reservoir after i rounds is 1,000/(i-1) * (i-1)/i = 1,000/i.

This ends up a little complex, but works just the same way as the random assigned numbers above.

Weighted Reservoir Sampling Variation

Now take the same problem above but add an extra challenge: How would you sample from a weighted distribution where each element has a given weight associated with it in the stream? This is sorta tricky. Pavlos S. Efraimidis figured out the solution in 2005 in a paper titled Weighted Random Sampling with a Reservoir. It works similarly to the assigning a random number solution above.

As you process the stream, assign each item a "key". For each item in the stream *i*, let k_i be the item's "key", let w_i be the weight of that item and let r_i be a random number between 0 and 1. The "key", k_i, is a random number to the n'th root where n is weight of that item in the stream:

$$k_i = u_i^{(1/w_i)}$$

. Now, simply keep the top n elements ordered by their keys, where n is the size of your sample. Easy.

To see how this works, lets start with non-weighted elements (ie: weight = 1). w_i is always 1, so the key is simply a random number and this algorithm degrades into the simple algorithm mentioned above. Now, how does it work with weights? The probability of choosing i over j is the probability that $k_i > k_j$. k_i can have any value from 0 - 1. However, it's more likely to be closer to 1 the higher w is. We can see what the distribution of this looks like when comparing to a weight 1 element by integrating k over all values of random numbers from 0 - 1. You get something like this:

If $w = 1$, $k = 1 / 2$. If $w = 9$, $k = 9 / 10$. When replacing against a weight 1 element, an item of weight 5 would have a 50% chance and an element of weight 9 would have a 90% chance. Similar math works for two elements of non-zero weight.

Distributed Reservoir Sampling Variation

This is the problem that got me researching the weighted sample above. In both of the above algorithms, I can process the stream in O(N) time where N is length of the stream, in other words: in a single pass. If I want to break break up the problem on say 10 machines and solve it close to 10 times faster, how can I do that?

The answer is to have each of the 10 machines take roughly 1/10th of the input to process and generate their own reservoir sample from their subset of the data using the weighted variation above. Then, a final process must take the 10 output reservoirs and merge them.

The trick is that **the final process must use the original "key" weights computed in the first pass**. For example, If one of your 10 machines processed only 10 items in a size-10 sample, and the other 10 machines each processed 1 million items, you would expect that the one machine with 10 items would likely have smaller keys and hence be less likely to be selected in the final output. If you recompute keys in the final process, then all of the input items would be treated equally when they shouldn't.

Birds of a Feather

If you are one of the handful of people interested in Reservoir Sampling and advanced software algorithms like this, you are the type of person I'd like to see working with me at Google. If you send me your resume (ggrothau@gmail.com), I can make sure it gets in front of the right recruiters and watch to make sure that it doesn't get lost in the pile that we get every day. **Update**: Despite the fact that this post was published in 2008, this offer still stands today.