Gregable

Greg Grothaus' Blog.
Discussing geekery, the environment, and life in Silicon Valley.

Oct 8, 2007

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 be the item's "key", let be the weight of that item and let u_i be a random number between 0 and 1. The "key", , is a random number to the n'th root where n is weight of that item in the stream: . Now, simply keep the

About Me

Greg Grothaus

Software engineer of Google Search Quality in the CA Bay Area. I like to write about random geekery, climate change, and life in silicon valley.

ggrothau@gmail.com
+Gregable on Google+
@gregable on Twitter
View my complete profile

Blog Archive

- ► 2014 (9)
- ► 2013 (6)
- ► 2012 (4)
- ► 2011 (13)
- ► 2010 (22)
- ► 2009 (20)
- ► 2008 (32)
- ▼ 2007 (29)
  - ► December (6)
  - ► November (2)
  - ▼ October (5)
    - Sentinel Dome, Jeffrey Pine in Yosemite
    - Standing an Egg on Equinox
    - Reservoir Sampling - Sampling from a stream of ele...
    - Path of Technology
    - Santa Cruz Monarch Butterflies
  - ► September (4)
  - ► August (2)
  - ► July (1)
  - ► May (3)
  - ► April (4)
  - ► March (1)
  - ► February (1)
- ► 2006 (7)

Labels

algorithms (4)
bay area (19)
environment (15)
google (17)
outdoors (24)
software (14)

Radu's Flight Training

Grace: Oklahoma Artist

Cristin's Bracelets & Necklaces

Natural Fiber and Color

Boston Big Picture

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). 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_j. 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:

k =

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.

Posted by Greg Grothaus at 10:27 PM

Labels: algorithms

23 comments:

Mark said...

Great post!

I used a similar technique in a paper of mine years ago. We needed a way of converting a stream of examples to a manageable batch and hit upon the same random replacement idea.

I didn't realise it was called reservoir sampling and didn't spend any time doing a proper analysis. Good to see someone has though!

Wed Jun 25, 03:48:00 AM 2008

George said...

Interesting article. It's hard to find good blogs on practical algorithms. Thanks!

Wed Jun 25, 03:55:00 AM 2008

Michael said...

To reduce the number of steps in the uniform case (1000 out of i), it could be easier to

- draw a random number R from the interval [1,i] (i > 1000, and assuming we index from 1),

- if R <= 1000, replace element R from your current sample with the new element

Wed Jun 25, 04:53:00 AM 2008

martin said...

Great post, I didn't know this had a name, or about the weighted version.

For the "batch" version, you say:

The right answer is generating random integers between 0 and N-1, then retrieving the elements at those indices and you have your answer.

That's sampling with replacement; the reservoir sampling algorithm is without replacement. So they're not quite equivalent, but in the case you talked about where the dataset is much bigger than the sample size, they're pretty close.

Wed Jun 25, 06:17:00 AM 2008

Eric said...

Another tiny (yet important) extension is given by Knuth that if you can't put all 1000 samples in memory, you can store the sample in external file as reservior and store the index of them in the memory. Finally sort the index and retrive the samples from reservior. This result is given at TAoCP, Vol 2, page 144, algorithm R. This result can be extended to the distributed sampling you mentioned above. I met this problem as an interview problem at Google.

Wed Jun 25, 06:37:00 AM 2008

Scott Turner said...

In practice, one problem with reservoir sampling is that the cost of generating a random number for each element in the stream may greatly outweigh the cost of counting the length of the stream.

Wed Jun 25, 07:20:00 AM 2008

Orthopteroid said...

Interesting post. Your weighted selection method reminds me of "remainder stochastic sampling" for weighted selection when using genetic algorithms.

Wed Jun 25, 08:39:00 AM 2008

senko said...

You don't actually need 1000 elements (for the basic version of the algorithm).
You can always use just one current element, and in each step, keep it with probability N/N+1, or replace it with the next element from your scan, with the probability 1/N+1 (where N == number of already scanned elements). Using induction it's easy to show it gives correct probabilities for each step of the way (assuming the random number generator you use is good).

I don't know if this variation is also called Reservoir Sampling or can be considered another algorithm, but looks conceptially the same as the one you described (I figured it out when trying to efficiently get a random quote from IRC logs, so I don't know if it has a name).

Wed Jun 25, 11:34:00 PM 2008

Greg said...

A friend of mine just pointed me at http://www.sciencedirect.com/science?_ob=ArticleURL&_udi=B6V0F-4HPK8WS-2&_user=10&_rdoc=1&_fmt=&_orig=search&_sort=d&view=c&_acct=C000050221&_version=1&_urlVersion=0&_useri
as another good reference on the topic.

Tue Nov 11, 01:27:00 PM 2008

danvk said...

It's not necessary to flip a coin to decide whether to keep each element. Rather, you can generate a random number of elements to skip.

Example: say you're sampling 1000 elements out of a billion. Towards the end of the stream, you're sampling with probability 1/1,000,000. Rather than generating a million random numbers to decide whether to keep each element, you can decide which of the next million elements to keep, saving yourself a million random number generations. Getting the distribution right is a bit tricky, but it can be done.

See http://www.cs.umd.edu/~samir/498/vitter.pdf for details.

Thu Jan 08, 09:00:00 AM 2009

Greg said...

Dan, nice observation. You can definitely do that if you know that you have at least X elements remaining in the stream.

If you know how many elements are in the entire set in advance, you can just select random indexes into the set and go from there though.

I'm not convinced you can use your trick if you don't know how many elements are remaining. I could be wrong.

Fri Jan 09, 11:40:00 AM 2009

liminescence said...

Hi Mark,
I was wondering, how the approach be if your weight has floating point precisions?

Wed May 04, 02:26:00 PM 2011

edwardw said...

Came here from a Stackexchange theoretical computer science Q&A. Great explanation, Greg, far better than Wikipedia entry on the topic. I am also curious whether your offer of helping on Google application still stand.

Sat Mar 10, 12:31:00 AM 2012

Tommy said...

Greg, as DanVK pointed out, the paper describes an algorithm (well, 3 in fact) that DOES work when you don't know the length of the stream. You "simply" calculate how many elements to skip, then skip them. If you reach the end, you're done... I encourage you to read it!

But of course, I haven't verified the math behind the paper...

Mon Mar 19, 10:12:00 PM 2012

Miguel said...

I was helping a friend solve a shuffling problem, and immediately remembered this post. Am I correct in thinking that:

1. Assuming it all fits in memory, we can make some variant of this that allows us to shuffle the items (rather than just sample them) as they come in (possibly by starting with a reservoir of size 1, and increasing the reservoir size as elements come in?)

2. This would actually be a terrible way to do a weighted random **shuffle**?

Wed May 30, 04:03:00 AM 2012

Greg said...

Miguel, I'm not sure what a "weighted shuffle" would be, but yeah it would probably not be the best way to do a random ordering. It's insertion sort of N items which is O(N^2). At the very least just assign random numbers to your entire set and run mergesort or such to get O(NlgN), though there is probably a cheaper way to randomly reorder elements if I thought about it.

Fri Jun 15, 08:06:00 PM 2012

Adam Ashenfelter said...

Just wanted to say thanks. This blog post pointed me in the right direction for integrating weighted reservoir sampling into my Clojure sampling library. For any other Clojure users out there:
https://github.com/bigmlcom/sampling

Tue Jan 22, 12:51:00 PM 2013

Matthias Görgens said...

Greg, why do you think it is insertion sort? Just keep your reservoir as a heap, and make it equivalent to heap sort.

Thu Feb 07, 04:25:00 AM 2013

Greg said...

Matthias, that's a good point. I didn't think of using a heap.

Thu Feb 07, 12:52:00 PM 2013

Zheyi RONG said...

Thanks. This post gave me the thought to implement my distributed sampling.

Just wanted to say if one wants to do a weighted random sampling with replacement, he can concurrently run k instances of the reservoir sampling algorithm without replacement (but with reservoir size=1) over the data stream.

However, it could be slow, since for each item it needs to decide for k times. The time complexity is thus O(kN), where k is the size of the sample and N is the length of the stream.

So I would like to ask if it is possible to reduce this time complexity?
Though the paper "Weighted random sampling with a reservoir" proposes a good way called 'AExpJ', it still needs to decide k times for each item.

Fri Feb 15, 05:55:00 AM 2013

George O said...

We have a sub-linear algorithm for reservoir sampling with replacement. See Byung-Hoon Park, George Ostrouchov, Nagiza F. Samatova, Sampling streaming data with replacement, Computational Statistics & Data Analysis, Volume 52, Issue 2, 15 October 2007, Pages 750-762, ISSN 0167-9473, 10.1016/j.csda.2007.03.010.
(http://www.sciencedirect.com/science/article/pii/S0167947307001089)

We know the population size up to any point so to maintain a reservoir for any point we can compute skipping probabilities.

Fri May 03, 08:29:00 AM 2013

Maverick(Pramit) said...

Nice explanation. I enjoyed it.

Sun Mar 16, 06:15:00 PM 2014

Maverick(Pramit) said...

http://www.cs.umd.edu/~samir/498/vitter.pdf

Sun Mar 16, 06:16:00 PM 2014

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)