Sampling From a Moving Window Over Streaming Data

Brian Babcock * Mayur Datar * Rajeev Motwani *

Abstract

We introduce the problem of sampling from a moving window of recent items from a data stream and develop the "chain-sample" and "priority-sample" algorithms for this problem.

1 Introduction

In many applications, the timeliness of data is important, and the most recent data is considered to be most interesting. Outdated data is "expired" and no longer used when evaluating queries. We consider the problem of maintaining a uniform random sample of a specified size k over a "moving window" of the most recent elements in a data stream. (For an overview of the streaming data model, see [2].) We present memory-efficient algorithms for this problem under two definitions of a moving window. A *sequence-based window* of size n consists of the n most recent data elements to arrive, while a *timestamp-based window* of duration t consists of all data elements whose arrival timestamp is within a time interval t of the current time.

The problem of how to maintain a sample of a specified size k over data that arrives online has been studied in the past. The standard solution is to use Vitter's reservoir sampling techniques developed in [6]. Reservoir sampling works well when the incoming data contains only inserts and updates but runs into difficulties if the data contains deletions, as is the case when data expires. The solution used in [3] is to periodically regenerate the sample when there have been too many deletions by an expensive scan of the entire database. The approach for dealing with deletions in [4] is to keep counts of the most common data elements using probabilistic counting rather than attemping to maintain a random sample.

*Dept of Computer Science, Stanford Univ, CA 94305. E-mail: {babcock,datar,rajeev}@cs.stanford.edu

2 Sequence-Based Windows

One algorithm for sampling with a sequence-based moving window is to maintain a reservoir sample for the first n data elements in the stream, and thereafter to stop maintaining the sample except that when the arrival of a new data element causes an element present in the sample to expire, the expired element is replaced with the newly-arrived element. This algorithm maintains a uniform random sample over a window of the last n elements while requiring only enough memory to store k data elements, but it has the disadvantage that it is highly periodic: if the data element with sequence number i is included in the sample, then so will be the data element with sequence number $i + cn$ for all integers $c > 0$. This regularity makes this technique unacceptable for many applications.

Another simple algorithm is to add each new data element to a "backing sample" with probability $\frac{2ck \log n}{n}$ and generate the sample of size k by down-sampling from the backing sample. As data elements expire they are removed from the backing sample. An argument using Chernoff bounds shows that the size of the backing sample will be between k and $4ck \log n$, except with probability $c'n^{-c}$. With high probability, the algorithm will both keep a large enough backing sample to supply the desired sample of size k and also use only $O(k \log n)$ memory.

The expected memory usage of the previous algorithm is $O(k \log n)$; a novel technique that we call "chain-sample" improves this to $O(k)$ while preserving the same high-probability upper bound of $O(k \log n)$. (The chain-sample algorithm described below generates a sample of size 1. To produce a sample of size k, maintain k independent chain-samples.)

In the "chain-sample" algorithm, when the ith element arrives it is chosen to become the sam-

ple with probability $Min(i,n)/n$. If the ith element is chosen as the sample, the algorithm also selects the index of the element that will replace it when it expires (assuming that it is still present in the sample when it expires). This index is picked uniformly at random from the range $i+1 \ldots i+n$, representing the range of indexes of the elements that will be active when the ith element expires. When the element with the selected index arrives, the algorithm stores it in memory and chooses the index of the element that will replace *it* when it expires, etc., building a chain of elements to use in case of the expiration of the current element in the sample.

The expected length of the chain of elements when the element in the sample is the ith oldest non-expired element is given by the recurrence:

$$\begin{aligned} T[1] &= 1 \\ T[i+1] &= 1 + \frac{1}{n}\sum_{j=1}^{i} T[j] \end{aligned}$$

which bounds the expected length by $T[n] \leq e$.

We can also derive an $O(\log n)$ high-probability upper bound on the memory usage for a single chain. The number of possible chains of elements with more than x data elements is bounded by the number of partitions of n into x ordered integer parts, which is $\binom{n}{x}$. Since each such chain has probability n^{-x}, the probability of any such chain occuring is less than $\binom{n}{x}n^{-x}$, which by Stirling's approximation is less than $(\frac{e}{x})^x$. When $x = O(\log n)$ this probability is less than n^{-c} for constant c.

3 Timestamp-Based Windows

The techniques described in the previous section will not work for timestamp-based windows because the number of data elements in the moving window may vary over time. We have developed an algorithm we call "priority-sample" for use with timestamp-based windows. As each data element arrives, it is assigned a randomly-chosen priority between 0 and 1. The element selected for inclusion in the sample is the "active" (non-expired) element with the highest priority. (To maintain a sample of size k, generate k priorities $p_1 \ldots p_k$ for each element and choose the element with the highest p_i for each i.)

The only data elements that we need to store in memory are those for which there is no element with both a later timestamp and a higher priority, since only these elements can ever be used in the sample. We can easily maintain a linked list of all elements with this property, ordering the linked list by decreasing priority and increasing timestamp.

The linked list maintained by the algorithm is analogous to the right spine of a "treap" where the timestamps are fully ordered and the priorities are heap ordered. Therefore, by the argument in [1], the expected length of this list when there are n active elements is $H(n)$, the nth harmonic number. Furthermore, an application of the Chernoff bound on the harmonic numbers (see [5]) demonstrates that the probability that the length of the list will exceed $2c\ln n+1$ when there are n active elements is less than $2(n/e)^{-c}\ln(c/e)$. Thus $O(k \log n)$ is both the expected memory usage of "priority-sample" and also a high-probability upper bound on the memory usage.

4 Acknowledgements

The authors thank Adam Meyerson and Sergey Brin for helpful suggestions.

References

[1] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proc. 30th IEEE FOCS*, 1989.

[2] S. Babu and J. Widom. Continuous queries over data streams. Technical report, Stanford University Database Group, March 2001.

[3] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *Proc. 23rd VLDB*, 1997.

[4] Y. Matias, J. S. Vitter, and M. Wang. Dynamic maintenance of wavelet-based histograms. In *Proc. 26th VLDB*, 2000.

[5] K. Mulmuley. *An Introduction through Randomized Algorithms.* Prentice Hall, 1993.

[6] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. on Math. Software*, 11:31–35, 1985.