COMP 480/580 - Probabilistic Algorithms and Data Structure *Sep 26, 2024*

Lecture 9

Lecturer: Anshumali Shrivastava **Scribe by:** Yuanyuan Xu (yx102), Davy He (dh72)
Lily Gao (qg8), Pranav Donepudi (pd53)

---

1 Data Stream and Basic Random Sampling

1.1 Data Stream

To process data streams efficiently, we have to first understand the properties of data streams. We do not know the entire dataset in advance, so it is convenient to think of the data as infinite. And since we are accepting a stream of data, input data comes one after another, as compared to pre-stored data that can be queried for any element at any time. Using data streams is efficient for processing real-time high-volume and high-velocity data, that's why big tech companies use data streams for applications like mining query streams, click streams, and social network news feeds. For example, by mining query streams, Google can analyze what queries are more frequently accessed today than yesterday. In this class, we want to explore the key question: how can we make critical calculations about the stream using a limited amount of memory?

1.2 One Pass Model

Consider the scenario where we have a large set of hashtags that we want to keep track of weekly. However, we don't have enough space to store all hashtags, especially when factoring in traffic. Therefore, instead of a traditional data storage approach, we want to utilize the One Pass Model which processes each data entry with a single pass and moves on.

To formalize the One Pass setup, at time t, we observe x_t. We assume that the observation sequence is $D_n = \{x_1, x_2, \ldots, x_n\}$, with n unknown. Our goal is to calculate $f(D_n)$, with f being a function of our interest. However, we have at any time t, a limited memory budget that is much less than the t (or n), so we can't store every observation. Essentially, the algorithm should at any point in time t, be able to compute $f(D_t)$, because this ensures that we can get useful information from data streams continuously with minimum storage requirements.

Due to the impossibility of revisiting previous data entry, we think there are other reasonable approaches that we can employ, such as Basic Random Sampling.

1.3 Basic Random Sampling

Let's revisit the hashtag scenario described in 1.2, we approximate n = 5 million, meaning we expect to process around 5 million hashtags. This would solve many problems because as long as the estimation is not far off, we don't care about whether in the end, we processed 5 million or

5.02 million hashtags. We can perform Random Sampling by getting a representative sample of the stream and performing analysis on it, let's discuss 2 ways to get random sampling.

First, we can sample a fraction. Considering sampling 1/10 data of the total size, we can generate a random number between [1-10] and if that number is 10, then use the sample. However, the size is unknown, so this can go unbounded. Also, there could be sampling bias caused by the difference in the fraction of duplicates in the original data stream and sampled data stream. A simple illustration of duplicates is as follows:

- The original dataset has U unique elements and D elements with one duplicate, giving us a fraction of the original dataset = $\frac{2D}{U+2D}$
- The sampling dataset has $\frac{D}{100}$ pairs of duplicates, with $\frac{18D}{100}$ duplicates appearing only once.
- The sampling dataset would underestimate the fraction of duplicates.

Second, we can sample a fixed number of data entries. Say we want to sample s elements from the stream, which means we have exactly s elements, and when stopped after nth elements, we would have every element having s/n probability of being sampled. The detailed sampling technique is Reservoir Sampling discussed in the next section.

2 Reservoir Sampling

2.1 Problem Description

Here when faced with a more complex sampling problem from a data stream, if we want to select s elements from the stream at any time t when the sampling ends at the n-th element. There are two essential conditions to meet during this process:

1. Every element, from the first up to the n-th, should have an equal chance of being selected, which could be 1 when n ≤ s and translates to $\frac{s}{n}$ when n > s;
2. By the end of the sampling, the total number of chosen elements should be precise s (when n > s).

2.2 Algorithm

Input: $x_t \leftarrow$ new observation at time t; $s \leftarrow$ sample size

Output: $S \leftarrow$ uniform samples of time t.

- Observe x_n
- If $n < s$: keep x_n
- Else: With probability $\frac{s}{n}$, select x_n to replace one of the elements in S uniformly.

It is obvious that the algorithm satisfies the second required condition of the problem, which means it sampled precisely s samples. So all we need to do is to prove that it can also uniformly select elements from the data stream. This can be proved by **induction**.

2.3 Correctness of the algorithm

2.3.1 Claim

Claim: At any time t , any element in the sequence $x_1, x_2, \ldots, x_n$ has the same chance of being in the sample. The chance could be 1 if n < s else exactly $\frac{s}{n}$.

We can consider the time when the (n+1)th element arrives. If we have no more than s elements right now, then all the elements will be kept in the sample S, resulting in a chance of 1. However, if the n is larger than s, we need to further prove that our claim is right.

2.3.2 Proof by Induction

We can prove the left part of this claim by induction:

- **Inductive hypothesis:**
  After n elements, the sample S contains each element seen so far with prob.
- When the element x_{n+1} arrives:
- **Inductive step:**
  - For elements already in S, the probability that the algorithm keeps it in S is: $(1 - \frac{s}{n+1}) + \frac{s}{n+1}(1 - \frac{1}{s}) = \frac{n}{n+1}$. So after x_{n+1} arrives the probability is:
    $p_{stay} = p_{already\ in\ S\ at\ time\ n} \cdot p_{still\ in\ S\ after\ (n+1)\ arrived} = \frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1}$
  - Also for x_{n+1}, the probability that it's added to S is $\frac{s}{n+1}$.

Therefore, we prove that the reservoir with s samples is uniformly selected from the data stream with the same chance. So the algorithm can satisfy all the requirements of the sampling problem.

While this algorithm efficiently selects a random sample from a stream of data with equal probability, many real-world scenarios require giving more importance or ***weight*** to certain elements in the stream. This leads us to the concept of Weighted Reservoir Sampling, an extension of the basic problem because not all items should be treated equally, which will be explored in the next section.

3 Weighted Reservoir Sampling (WRS) Problem

3.1 Problem Description:

Weighted reservoir sampling is a technique for randomly selecting items from a data stream, where each item is assigned a weight that influences its probability of being chosen.

Given That:

a) A stream of n items, where n is unknown or very large
b) Each item i has a weight $w_i > 0$
c) Select a random sample of k items, where k < n

Goal: The probability of selecting an item should be proportional to its weight.

3.2 Algorithm A (Efraimidis-Spirakis Algorithm):

The most efficient algorithm for this problem is known as Algorithm A (Efraimidis-Spirakis Algorithm):

1. Initialize: R = ∅ (empty reservoir)
2. For each item i in the stream:
   a) Generate a random number u_i between 0 and 1.
   b) Calculate the key $k_i = u_i^{(\frac{1}{w_i})}$
   c) If |R| < k:
      - Add (i, k_i) to R
   d) Else if k_i > smallest key in R:
      - Replace the item with the smallest key in R with (i, k_i)
3. The final reservoir R contains the weighted random sample.

This algorithm can be implemented efficiently using a min-heap to maintain the reservoir, resulting in O(log k) time complexity per item.

3.3 Proof:

The correctness of this algorithm is difficult, we can use a key lemma that provides insight into its behavior. Here's a simplified proof approach:

Given that: Random variables $\{r_1, ..., r_n\}$, uniformly distributed on the interval [0, 1], and weights $\{w_1, ..., w_n\}$.

1. The algorithm generates a key $k_i = u_i^{(\frac{1}{w_i})}$ for each item, where ui is uniformly distributed between 0 and 1.
2. For any two items i and j, the probability that $k_i > k_j$ is: $P(k_i > k_j) = \frac{w_i}{w_i + w_j}$
3. Assume the algorithm works correctly for n-1 items. When the nth item arrives:
   - It replaces an existing item in the reservoir with a probability $\frac{w_n}{w_1 + \ldots + w_n}$
   - This is the probability given by the lemma
4. This process ensures that, at any point, the probability of an item being in the reservoir is proportional to its weight relative to the total weight of all items encountered up to that point.

4 Heavy Hitters Problem & Some Basic Knowledge

4.1 Heavy Hitters Problem

4.1.1 What is a heavy hitters problem?

In data stream processing, it is common to perform a frequency analysis of the elements within the stream. The elements that appear most often are referred to as the "heavy hitters." The main challenge arises when the stream is too large to be stored in full, making conventional data structures like arrays and hash tables impractical for counting the frequencies of elements. Therefore, the objective is to develop a memory-efficient algorithm that can identify these heavy hitters in the stream.

4.1.2 Simple example

On Twitter, many tweets contain repetitive words or phrases, which helps identify trending topics. These commonly repeated phrases are considered the "heavy hitters." To simplify the problem, imagine we want to find the top 50 phrases on the Twitter feed. To achieve this, we need a streaming algorithm that is both memory-efficient and scalable.

A basic approach to this problem could work as follows: for each tweet, identify all possible contiguous sets of four words (4-grams) and store them in a dictionary. The keys of this dictionary would be every possible 4-word combination, and we would increment the count for each combination whenever it appears in the stream.

However, this naive method has serious limitations. For a stream containing a million words, the number of potential 4-grams is around $(10^6)^4$, or 10^{24}. This presents an enormous memory allocation challenge, making it both unrealistic and impractical. Even if we try to optimize by selectively choosing specific 4-grams instead of analyzing every possible combination, storing just the count array would still require around 1.6TB of memory. Consequently, this solution lacks scalability, highlighting the need for a more efficient approach.

4.1.3 Can we do better? Not always

In many cases, achieving a significantly more efficient solution for the heavy hitters problem is not always feasible. No single algorithm can solve this problem for every possible input in just one pass while using a sublinear amount of auxiliary space. This limitation can be demonstrated using the pigeonhole principle. Therefore, the strategy should involve making reasonable assumptions and filtering the input data to optimize the algorithm's performance.

In practice, this means focusing on techniques that prioritize certain elements or conditions within the data stream. By narrowing the scope, we can reduce the memory requirements and improve accuracy, even if an exact solution remains unattainable. The key lies in balancing the

trade-offs between precision, memory efficiency, and processing speed to achieve practical results.

4.2 Majority Element Problem

4.2.1 Problem statement

If we are given an input array A of length n, with the promise that it has a majority element — a value that is repeated in strictly more than n/2 of the array's entries. Our task is to find the majority element.

4.2.2 Best Solution

Here are lots of approaches to solve this problem, like the Naive approach, Binary Search Tree, Dictionary Approach… But what we want to use is the most excellent and simple one based on Counter, which has a Time Complexity of O(n) and Space Complexity of O(1).

Algorithm: Majority Element Algorithm

Input: Array A of length n
Output: The majority element

```
for i = 0 to n − 1 do
        if i==0 then
                current = A[i]
                Counter = 1
        else
                if current == A[i] then
                        Counter++
                else
                        Counter - -
        if Counter == 0 then
                current = A[i]
                Counter = 1
return current
```

4.2.3 Brief Illustration

We can rely on basic intuition to assess the effectiveness of the algorithm. If a majority element appears more than *(n/2)* times, the decrement process will not reduce its count to zero. This suggests that the algorithm can successfully identify the majority element, as its count will persist throughout the iterations.

4.3 Power Law and Bloom Filter with Counter

4.3.1 Power Law

The solutions for the majority element problem are not applicable in cases where a majority element is not guaranteed to exist. However, the power law indicates that, in real-world data, a

small number of elements tend to occur very frequently, while most others are much less common. Identifying these frequently occurring elements allows organizations to optimize resource allocation, tailor recommendations, and improve user experiences effectively.

4.3.2 Bloom Filter with Counter

To address the heavy hitter's problem, we can modify the Bloom filter's implementation slightly. Instead of using a simple bit array that flips between 0 and 1, we use each bucket as a counter. When an element is hashed, the corresponding counter is set to 1 or incremented by 1 if that element is already present. In cases where two elements, say x and y, collide in the hash, we simply add their counts—meaning bucket $h(x)$will store the sum of x and y counts. With a universal hash function, the probability that any specific counter $h(s)$equals a certain value c is $(1/R)$, where R is the total size of the counter array.