COMP-580: Probabilistic Algorithms and Data Structures
Rice University
Fall 2023
Prof. Anshumali Shrivastava

Stream Estimation 1: Count - Min Sketch

Naman Gupta(ng63), Shambhavi Kurup(sk223), Suhas Achanta(sa181)

1 Streaming

Data transmission comprises a multitude of techniques that are primarily bucketed by streams. A stream is essentially a continuous flow of data where the input data elements arrive one after another. One of the critical drawbacks of streams is that they cannot be stored entirely because of the need to deal with vast sizes that are not easy to process, thereby raising a question on the overall scalability of streams. Hence, making critical calculations for streams with a limited amount of memory is a challenge that requires employing algorithms which are designed for memory-efficient streaming. These algorithms are used for processing streams in a space-efficient fashion, making it possible to make computations without storing the entire stream in the memory. They are particularly used when large datasets cannot fit into the memory entirely.

A stream can be further defined in a quantifiable manner as a sequence A = $<a_1, a_2, ..., a_n>$ of m items where each $a_i \epsilon$[m], and it takes log(m) bits of space to represent each item a_i in the stream. log(m) bits is generally considered as it provides an efficient representation for a wide range of values. Furthermore, for counting elements a space of log(n) bits is required.
Let g(A) be the quantifiable result that is intended to be computed from the stream. Utilising an efficient streaming algorithm helps in achieving this goal which can be any particular statistic depending on the data obtained from the stream. To understand the count frequency, $c_i = |a_i \epsilon A, a_i = j|$ is introduced to denote the number of stream elements with the fixed value j.

2 Heavy Hitters Problem

Often during data stream processing, there is a fundamental understanding of conducting a frequency analysis of the elements within the data stream. The elements that occur most frequently in the stream are known as the "heavy hitters" of the stream. The key challenge is that when the stream size is large, it is rather difficult to store it in its entirety, thereby making traditional data structures like arrays and hash tables rather infeasible for counting frequencies of stream elements. Hence the goal is to devise a memory-efficient solution (streaming algorithm) that can detect these heavy hitters.

2.1 Motivating Example - Twitter's Data Mining Methodology

On Twitter, there are many tweets comprising certain words or phrases that are commonly repetitive. This is used to detect which particular topic is trending on the platform and the phrases are deemed as the heavy hitters. To put the problem in layman's terms, in a stream of tweets we are to find the top 50 phrases on the Twitter feed. For this, we need a concrete streaming algorithm with scalable memory and runtime.
A naive solution to this problem would be as follows:

- For every tweet, find the 4 most contiguous words and store them in a dictionary.
- The keys for the dictionary will be all the possible combinations of the four phrases (4-grams).
- While iterating through the stream, the count value is incremented as and when a particular combination is encountered.

The solution proposed above utilizes a more naive approach to counting heavy hitters. For a million words, the number of possible phrases is $(10^6)^4$ which is 10^{24}. The glaring issue of memory allocation persists as it presents an unrealistic and impractical memory cost. In an attempt to reduce this complexity, if we are to make a smart selection of the 4-grams and not analyse all possible combinations, it would still require $1.6TB$ to simply store the count array. Therefore the solution is not scalable and a more efficient solution needs to be devised.

2.2 More Applications of Heavy Hitters

The need to find the most frequent elements in a data stream is a rather common practice observed in various cases. Some of the examples are as follows:

- Google Trends tracks search queries made in real-time, and to ensure that these queries are served efficiently, the heavy hitters are tracked to make up for the traffic.
- Heavy TCP flows Identification. IP addresses or services generating the most network traffic without keeping a complete record of every data packet allowing you to monitor and manage your network effectively while conserving valuable resources.
- Identifying popular products for a store under a given set of constraints.
- Stock trends.

2.3 Can we do better?

It isn't always possible to do better given a heavy hitter's problem. No algorithm solves this problem for all given inputs within one pass while using a sublinear amount of auxiliary space. This result can be proved using the pigeonhole principle. Hence the idea should be to make appropriate assumptions and filter out the inputs to gain some improvements out of the algorithms.

3 Majority Element Problem

Finding the majority element is one of the most common and intriguing problems in data analysis and computer science, and it has numerous implications in various fields such as determining the winner in voting systems or identifying popular products in market research.

3.1 Problem Statement

The Majority Element Problem entails recognizing an element within an array that occurs more frequently than any other element and accounts for more than half of the total items in the array. Locating an element x in an array A of length n where the count of x in A exceeds $\frac{n}{2}$.

3.2 Solutions

- Naive Approach: The simplest approach is using two loops to track the maximum number of different components. When this maximum count exceeds $\frac{n}{2}$, the procedure is terminated, and the element associated with this maximum count is returned as the majority element.
  Time Complexity: $O(n^2)$ Space Complexity: $O(1)$

- Binary Search Tree: Insert elements into BST one by one, and if an element is already there, increase the node count. If the count of a node exceeds $\frac{n}{2}$ at any point, then return.
  Time Complexity: $O(n^2)$ Space Complexity: $O(n)$

- Dictionary - Make an empty dictionary and iterate through the elements. If the element already exists in the dictionary, increment its value; otherwise, insert the element as the dictionary's key with a value 1.
  Time Complexity: $O(n)$ Space Complexity: $O(n)$

- Counter: Iterate through the list, incrementing, decrementing, and changing the value of the counter.
  Time Complexity: $O(n)$ Space Complexity: $O(1)$

Algorithm 1 Majority Element Algorithm

Require: Array A of length n
Ensure: The majority element

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

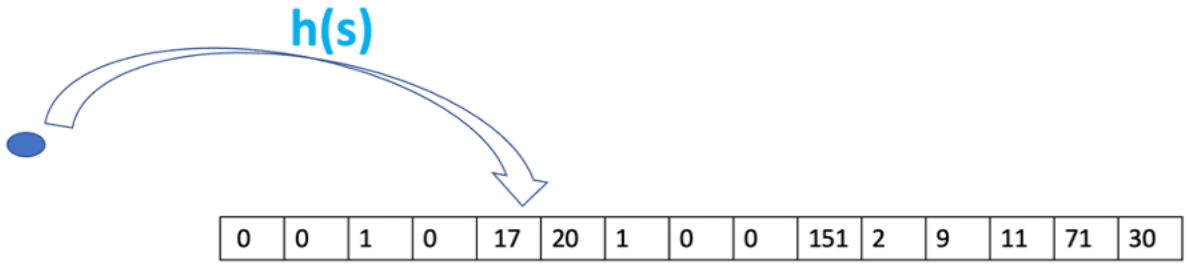
We may use simple intuition to determine whether the above algorithm will work or not because if the majority element exists in such a way that it occurs more than $\frac{n}{2}$ times, the decrement will never drop the current count for the majority element to zero.

3.3 Power Law

The solutions to the majority element problem do not apply in the general case, that is where the existence of a majority element is not guaranteed. However, the power law states that it is a common observation that real-world data distribution represents a tiny fraction of things or elements that appear frequently, while the overwhelming majority appears infrequently. By recognizing the most frequently occurring elements organizations can optimize resource allocation, personalize recommendations, and enhance user experiences.

4 Bloom Filter with Counter

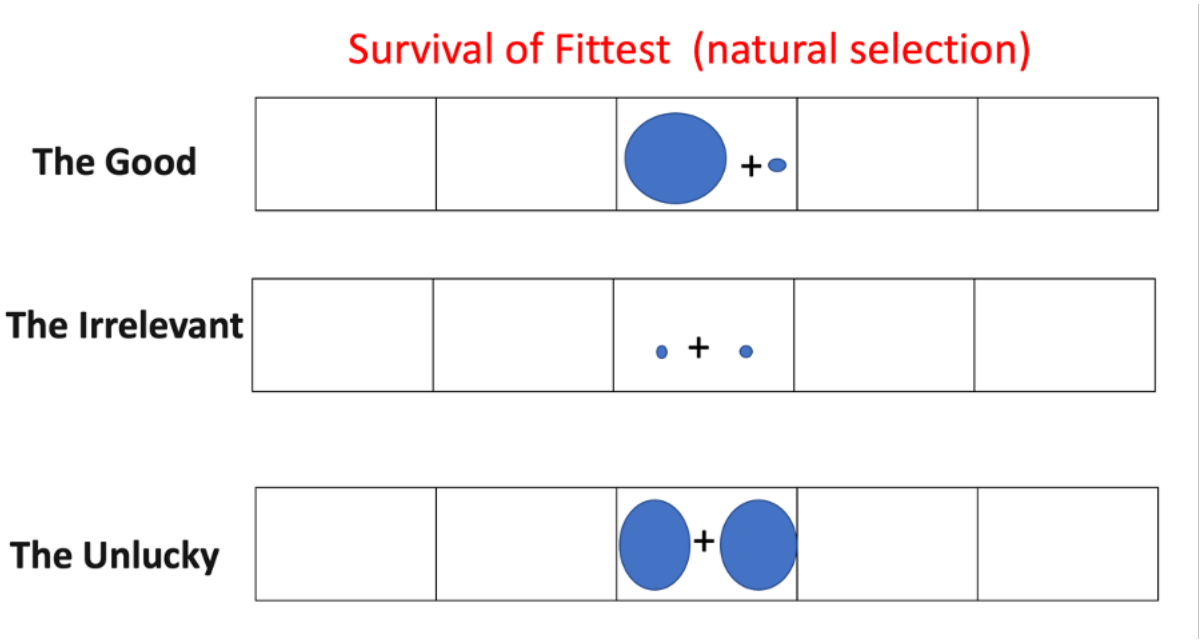
We can solve the heavy hitter's problem by making a minor tweak to the Bloom filter's implementation. Rather than utilizing a bit array and flipping the bits between 0 and 1, we will use the bucket as a counter that is when we hash a value from the array, we will set that specific counter to 1 or increment it by 1 if a value is already present. If element x and element y clash, we simply add their counts that is, bucket $h(x)$ will contain $x + y$ counts. With the universal hash function, the probability of $Pr(h(s) = c) = \frac{1}{R}$ where R is the size of the counter array (taught in the previous class).



4.1 Limitations and Analysis

With the bloom filter indicated above, four different outputs are possible.

- We have a perfect hash function and there are no collisions.
- The hash value of a heavy hitter is the same as another item in the data that is present in a very small amount. In this case, there will be a collision but the bloom filter will give a good estimate of the heavy hitter.
- The hash values of two extremely uncommon elements in the data are the same. In this case, the bloom filter counter value will be less than that of a heavy hitter.
- A heavy hitter's hash value is the same as another heavy hitter's. In this case, there will be a collision, and the bloom filter will grossly overstate the number.



The expectation of a hash value in the revised bloom filter can be defined as:

$$E[h(s)] = E\left[c_s + \sum_{\substack{\text{h(s)=h(i)} \\ i \neq s}} \frac{c_i}{R}\right]$$

where c_s is the count of the string s, $i \neq s$ means element s which is not same as i but for which the hash value is same as i and R is the length of the data.
As we wish to estimate c_s. The expected error can be written as :

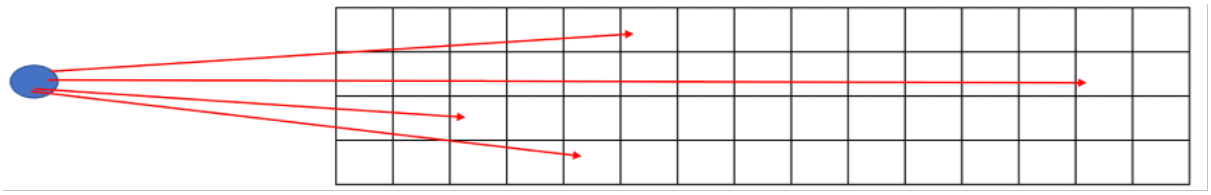
$$E[Error] = \sum_{i \neq s} \frac{c_i}{R}$$

For simplicity lets replace $\sum_{i \neq s} \cdot c_i = \sum$ so the equation above becomes:

$$E[Error] < \frac{\sum}{R}$$

5 Count-Min Sketch

We see that using a counting bloom filter yields an unlucky situation where we have collisions between two or more elements, all having large counts. The resulting hash values become extremely large and we grossly overestimate the occurrence of each of those those keys. To do better, we use the idea of counting bloom filters and make it stronger using the power of k choices. This is called the Count-Min Sketch. We have d hash functions (h_1, h_2, ..., h_d) and maintain d corresponding hash tables of length R (A_1, A_2, ..., A_d). For every string s that we see in the stream, we update by incrementing d counters - $A_1[h_1(s)]$, $A_2[h_2(s)]$, ..., and $A_d[h_d(s)]$. To estimate the count of the string s, we return the minimum of the d counters - $min(A_1[h_1(s)]$, $A_2[h_2(s)]$, ... $A_d[h_d(s)])$.



We know that every counter we maintain for a string s is either an overestimate (in case of collisions) or the exact count for s (in case of no collisions). None of the d counters maintained for s can ever be an underestimate. Using the Count-Min Sketch method, we are in a way ensuring that we take the best overestimate of d overestimates as the count of an element. The intuition behind this method performing better than a simple counting bloom filter is that for a string s, unless all our d counters mess up simultaneously, we have a good estimate of the count for s. Even though the minimum of d overestimates is still an overestimate, it is the best overestimate in d-universes.

Let's look back at the problem we were facing with heavy hitters while using a counting bloom filter. When we had a collision between two elements with relatively large counts, we were overestimating the count of both elements by a large margin. For the same problem to carry over in the Count-Min Sketch scenario i.e., for the estimate of a heavy hitter to be distorted by a large degree, it must be distorted in *all* d universes. Two or more heavy hitters colliding using one hash function is in itself not a common event. Two or more heavy hitters colliding using d independent hash functions is exceptionally rare. So, even if there exists *one* universe in which a heavy hitter s did not collide with another heavy hitter, we have a reasonable estimate for s.

The update operation takes constant time - $O(d)$ - since d is set at the beginning.

5.1 Error Analysis

When we were using the counting bloom filter (one hash function), we found the expected error to be $E(err) = \epsilon\Sigma$, where $\epsilon = \frac{1}{R}$.

Applying Markov inequality:

$$Pr(err > 2\epsilon\Sigma) < \frac{E(err)}{2\epsilon\Sigma} = \frac{\epsilon\Sigma}{2\epsilon\Sigma} = \frac{1}{2}$$

This means that the probability of our error being worse than $2\epsilon\Sigma$ is bounded by 0.5^d. For heavy hitters, this error is quite low and we get a good estimate with a d value of 4 or 5.

5.2 Memory Analysis

We want to ensure that for a given string s, with probability $1 - \delta$, we have the following bound for $\hat{c_s}$:

$$c_s < \hat{c_s} < c_s + 2\epsilon\Sigma$$

Here, c_s denotes the true count of string s, $\hat{c_s}$ denotes the estimate of c_s, and δ is the probability of error for a given string s.

We know that $Pr(err > 2\epsilon\Sigma)$ is bounded by 0.5^d. This means that $0.5^d < \delta$.

$$0.5^d < \delta \Rightarrow d \cdot log(0.5) < log\delta \Rightarrow d < \frac{log\delta}{log0.5} \Rightarrow d < -log\delta$$

The memory requirement for this model is $R \cdot d$ since we are using d bloom filters each of length R. Using the above calculations for one input string, we can say that the memory cost in terms of ϵ and δ is $O\left(\frac{1}{\epsilon} log \frac{1}{\delta}\right)$. However, our error constraints must be satisfied for all N strings and not just some string s.

The probability of error for a given string s is bounded by δ, so the probability of error for any of N strings is bounded by $(N \cdot \delta)$. We scale our δ to $\frac{\delta}{N}$ and now have a memory requirement of $O\left(\frac{1}{\epsilon} log \frac{N}{\delta}\right)$ which is much better since it only increases logarithmically in terms of N.

6 How to identify top-k elements?

Given the problem of identifying the top-k elements within a stream while maintaining efficient time and space complexity, we make use of heaps to arrive at an optimal solution.

6.1 Use of heaps

- A minheap is created with a max value of k set to it. This will keep track of k items with the highest estimated frequencies that were observed.

- As the stream passes, whenever a new element is encountered, the frequency of the element that has already been counted through Count-Min Sketch is compared with the frequency of the top-k elements estimated in the minheap.
- If the estimate of the element (say s) is higher than what is present in minheap, then replace the minimum element in the heap with the new element s and update the heap.
- Time complexity of each update operation is $O(log(k))$.
- In the worst case, if d updates are performed for d distance of elements, then the time complexity would then be $O(d * log(k))$.