Scalable Data Science (CS-563)

Lecture 2-3: Introduction to Streaming Algorithms, and Bloom Filters

August, 2021

Lecturer: Rameshwar Pratap *Scribe: Mohit Kumar and Punit Pankaj Dubey*

1 Overview

In the past, the most common problem programmers encountered during computation was memory restrictions. Computers used to have up to 32 MB of RAM, but nowadays one can find up to 32 GB of RAM. But as the size of the hard disk and RAM increases, the size of the data also increases with time. So excessive research has been done to derive memory-efficient algorithm to overcome this problem.

Notation: Lets denote stream of data as $\sigma = \langle a_1, a_2, ...a_m \rangle$, where every elements present in the stream are from universe $\mathcal{U} = [n]$ implies $a_i \in [n]$, and the number of element present in the stream is denoted by m, i.e $|\sigma| = m$.

2 Streaming Algorithms

Streaming algorithms are designed to efficiently solve problems that can only be solved when all data are available at once instead of data-stream i.e when data points of the dataset are received sequentially. When developing a streaming algorithm, the following two aspects come to the fore:

- **Space efficient algorithm:** If the length of the stream is m and the size of the domain is n then algorithm should take order of logarithmic in n and m space.
- **Pass efficient algorithm:** Algorithm make a few passes (most of the time single pass) over the stream of data points, since pass over data stream is time inefficient and costly.

2.1 Data stream

A data stream is a configuration in which each data point in a dataset is received in turn, much like a network data packet received by a router. Data points are usually received in large numbers and storing these data points is inefficient, which makes evaluation of problems such as frequent elements in dataset, no of distinct elements in dataset, etc. very problematic.

2.2 Toy problems in Streaming Algorithm

Streaming algorithms for some toy problems are described below:

2.2.1 Computing the mean of elements in data stream:

Problem statement: Let a data stream defined as $\sigma = \{a_1, \ldots a_i \ldots a_m\}$, where $a_i \in \mathbb{R}$. The problem is to estimate the mean of all the elements observed so far, of the stream.
In general, the formula to calculate the mean of m elements for datasets is:

$$Mean = \frac{a_1 + a_2 + \ldots + a_m}{m}.$$

Streaming algorithm to compute mean: To calculate the mean of $m + 1$ elements in a stream, the mean of the first m elements must be known in advance.

$$Mean_{(m+1)} = \frac{a_1 + a_2 + \ldots + a_m + a_{m+1}}{m+1} = \frac{Mean_{(m)} \times m + a_{m+1}}{m+1}.$$

where $Mean_{(m)}$ is the element of the first m elements, algorithm for mean of the data stream σ is presented in Algorithm 1.

Algorithm 1 Algorithm to compute mean of elements in stream

1: **Input:** Data-Stream $\sigma = \langle a_1, a_2, \ldots a_m \rangle$
2: **Output:** Mean of σ
3: **Initialize** : $result \leftarrow 0$, $m \leftarrow 0$ ▷ maintain a counter of the number of elements see so far.
4: **Process** (token a_i):
5: $result \leftarrow (result \times m + a_i) \div m + 1$
6: **Step:** $m \leftarrow m + 1$

Lets observe an example for the given algorithm:

Example 2.1. *Let a data stream* $\sigma = \langle 1, 2, 3, 4, 5, 4, 3, \ldots \rangle$ *the mean of first four elements of* σ *is computed as follows: let take first three element i.e* $1, 2$ *and* 3 *then by using Algorithm 1,* $result = (1 + 2) \div 2 = 1.5$ *and* $m = 2$ *and* $next = 3$*. Then the output will be:*

$$result = \frac{result \times m + next}{m+1} = \frac{1.5 \times 2 + 3}{3} = 2.$$

Space complexity analysis: Space complexity for streaming algorithm to compute mean is defined as follows:

$$\begin{aligned} \text{Total space} &= \text{space of storing counter + space for storing average.} \\ &= O(\log m) + O(\log n). \\ &= O(\log m + \log n). \end{aligned}$$

2.3 Sampling an element from a steam

Problem Statement: To find a uniform sample S from a stream σ of unknown length.

Algorithm 2 Algorithm to sample an elements from stream

1: **Input:** Data-Stream $\sigma = \langle X_1, X_2, \ldots X_t \rangle$
2: **Output:** Sample S
3: **Initialize** : $S \leftarrow X_i$
4: **Process** (token X_t):
5: $S \leftarrow X_t$ with probability $\frac{1}{t}$. ▷ if X_t is included, discard the current element of S.

Theorem 2.2. *Algorithm 2 correctly maintains an uniform sample.*

Proof. If for a stream $\sigma = \{X_1, X_2, X_3, \ldots, X_m, \ldots, X_t\}$. The Probability that $S = X_i$ at time $t \geq i$ is:

$$\begin{aligned}\mathbf{Pr}[S = X_i] &= \frac{1}{i} \times \left(1 - \frac{1}{i+1}\right) \times \ldots \left(1 - \frac{1}{t}\right). \\ &= \frac{1}{t}.\end{aligned}$$

This proves the uniformity. □

3 Set-Membership Queries

Problem Statement: *Let a set S from universe $\mathcal{U}$, $S \subseteq \mathcal{U}$ with $S = \{X_1, X_2, X_3, \ldots, X_m\}$, the aim is to construct a data structure which solve a membership problem i.e $x_i \in S$ or $x_i \notin S$ and further data structure is both time and memory efficient.*
Lets observe some simple approach.

3.0.1 Binary Search tree (BST):

Binary Search tree which is also known as a sorted or ordered binary tree is a data structure mostly used to maintain a sorted stream of data. In BST, element are stored in a sorted list, and then to compare the element, binary search like procedure is used.

- The space required to store each element of the set S will be the $\log$ of the size of the universe $|U|$ i.e. $O(\log |\mathcal{U}|)$ bits.
- The space required for storing the tree, since there are n elements and each element is from the universe $(|\mathcal{U}|)$, so will be $O(n \log |\mathcal{U}|)$ bits.
- The search time will be equal to the $O(\log n)$.

BST is mostly used because the search time is the order of log of the number of elements, but the space required is of order $O(n \log |\mathcal{U}|)$ which is not practical in the case of streaming algorithms.

3.0.2 Bit Vector :

A bit vector is a data structure consisting of a number of bits equal to the size of universe, where the value of index i equal to 1 if element i is present in set S. It can be summarized as follows:

- For storing each element of universe $\mathcal{U}$, one bit is required and then set the corresponding bits present in the set s.
- The space required for storing the bit array is $O(|\mathcal{U}|)$ bits.
- The search time will be equal to the $O(1)$ which is fast as it is taking constant time to run the algorithm each time.

Limitations: Streaming algorithms need to be both memory and space efficient, but both the naive approach i.e BST and bit-vectors take more space which is not efficient for a streaming algorithm, furthermore BST tke longer time to search an element as compared to bit-vector.

3.1 Improved solution via Bloom filter:

Bloom Filter [Broder and Mitzenmacher, 2004] is a probabilistic data structure that is solves set membership problem and also fast and space-efficient, but comes at the cost of small false positive rate.

3.1.1 Working of bloom filter :

Empty bloom filter: An empty bloom filter is similar to a bit vector array having a size of m bits and all values set to zero.

Hashing an element in bloom Filter: To hash an element $x \in S$ to an empty bloom filter, k hash functions are used, which yields k values, which corresponds to index in the bloom filter, these indexes are then set to 1. Lets complement it via an example given below.

Example 3.1. *Hashing an element of set in Bloom filter.*
Insert X ***in bloom filter:*** *Insert an element* X *in bloom filter* A*, let the size of the bit vector array is* $m = 10$*, and* $k = 3$*, i.e three hash functions. Now lets first calculate the hash values for an element* X*:*

- $h_1(X) = 3X\%10 = 1.$
- $h_2(X) = 2X\%10 = 4.$
- $h_3(X) = 5X\%10 = 7.$

Now, set indices $1, 4$ *and* 7 *as* 1 *of a bloom filter* A*.*
Insert Y ***in bloom filter:*** *Similarly, insert* Y *to the same bloom filter and then calculate the hash value as:*

- $h_1(Y) = 3Y\%10 = 3.$
- $h_2(Y) = 2Y\%10 = 4.$
- $h_3(Y) = 5Y\%10 = 5.$

Now, set indices $3, 4$ *and* 5 *as* 1 *of a bloom filter* A*.*
The insertion procedure of bloom filter is efficiently illustrated in figure 1.

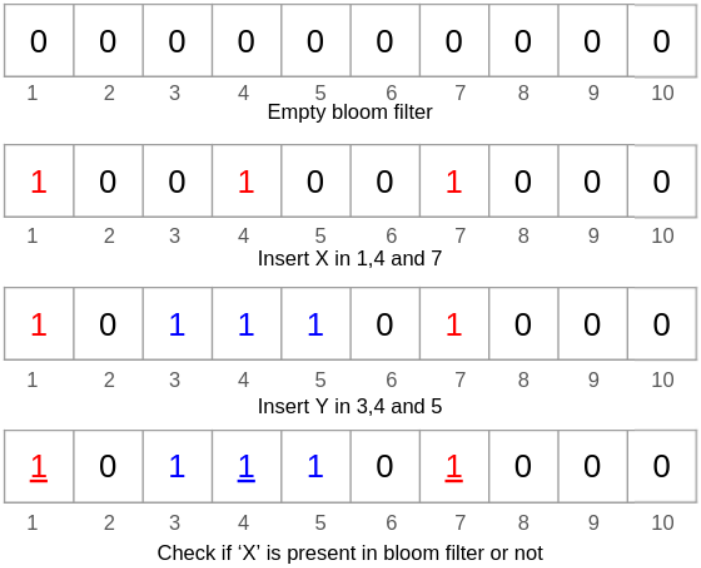

Figure 1: Illustration of insertion of elements in bloom filter and checking the membership queries.

Querying an element in bloom Filter: To check if an element X is present in the set or not, follow the same procedure as of hashing an element in a bloom filter, which means firstly compute the hash value of X by using h_1, h_2, and h_3 and then observers that all the indices are set to 1 or not in the bloom filter, if all the bits are 1, then X is probably in a set (false positive) and if any of bit it is 0 then, element is surely not present in a set, i.e bloom filter never gives false negative. Querying procedure can be easily illustrated by an example given below.

Example 3.2. *Query in bloom filter can be processed as follows: Let for query element Y_1, Y_2 for which hash values are defined as $H(Y_1) = \{1, 5, 8\}$ and $H(Y_2) = \{1, 3, 4\}$, lets find out elements are present in bloom filter or not? So for the first query i.e. $H(Y_1) = \{1, 5, 8\}$ the corresponding bit at an index 8 in the bloom filter is 0 and so it can be said that it is not in bloom filter but for the second query i.e. $H(Y_2) = \{1, 3, 4\}$ all of the corresponding bits are 1 and so it can be said that it is probably present in set which may be false positive.*

Theorem 3.3. *The optimal value of k, i.e number of hash function used in bloom filter that minimizes the false positive rate is $\frac{m}{n} \cdot \ln 2 = 0.7 \cdot fracmn$, and the corresponding false positive value is defined by: $f = \left(1 - [1 - \frac{1}{m}]^{kn}\right)^k = (1 - e^{\frac{-kn}{m}})^k$, where n is no of element in set S, i.e $|S| = n$ and m is no of bits in bloom filter.*

Proof. Let's define a bloom filter A, having m bits, with k hash functions, and a set S with $|S| = n$, i.e n no of element in a set S. Then, the probability p that a bit in a bloom filter is set to 0 after hashing of all the elements of set S is defined by :

$$p = \left(1 - \frac{1}{m}\right)^{nk} \approx e^{\frac{-kn}{m}}. \tag{1}$$

The probability that the bloom filter outputs a false positive is computed by finding the probability that a set of k specific bits are assigned as 1 is computed as:

$$f = (1-p)^k. \tag{2}$$

$$= \left(1 - \left[1 - \frac{1}{m}\right]^{nk}\right)^k. \tag{3}$$

$$\approx \left(1 - e^{\frac{-kn}{m}}\right)^k. \tag{4}$$

Increasing the value of k can increase or decrease the value of f. To find the optimal value of k which minimizes the value f is done by differentiating f with respect to k. f can also be expressed as:

$$f = \exp\left(k \ln\left(1 - e^{\frac{-kn}{m}}\right)\right). \tag{5}$$

Let g = $k \ln\left(1 - e^{\frac{-kn}{m}}\right)$,

$$\frac{\partial g}{\partial k} = \ln\left(1 - e^{\frac{-kn}{m}}\right) + \frac{kn}{m} \times \left(\frac{e^{\frac{-kn}{m}}}{1 - e^{\frac{-kn}{m}}}\right). \tag{6}$$

Now,

$$\frac{\partial g}{\partial k} = 0, \quad \text{when, } k = \frac{m}{n} \ln 2. \tag{7}$$

$$\frac{\partial^2 g}{\partial k^2} > 0, \quad \text{when, } k = \frac{m}{n} \ln 2 \quad \text{attains a global minima.} \tag{8}$$

Thus, putting $k = \frac{m}{n} \ln 2$,

$$f = \left(1 - e^{\frac{kn}{m}}\right)^k \tag{9}$$

$$= \left(\frac{1}{2}\right)^k \tag{10}$$

$$= 0.6185^{\frac{m}{n}}. \tag{11}$$

This proves the theorem. □

3.2 Applications of Bloom filter

Bloom filter have been heavily studied and applied in various applications such as weak password detection, Internet Cache Protocol [Geravand and Ahmadi, 2013], safe browsing in Google Chrome, Wallet synchronization in Bitcoin, Hash based IP Trace-back, cyber security like virus scanning and many more.

4 Next time:

In the subsequent lecture, we are going to learn about Universal hashing and also estimate Heavy-Hitters via Count-Min-Sketch.

References

[Broder and Mitzenmacher, 2004] Broder, A. and Mitzenmacher, M. (2004). Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509.

[Geravand and Ahmadi, 2013] Geravand, S. and Ahmadi, M. (2013). Bloom filter applications in network security: A state-of-the-art survey. *Computer Networks*, 57(18):4047–4064.