Data Sampling / Sketches

Outline

- Motivation
- Sampling Problems
  - Stream Sampling
  - Filtering Streams
  - Counting Distinct Element in a Stream
  - Moments

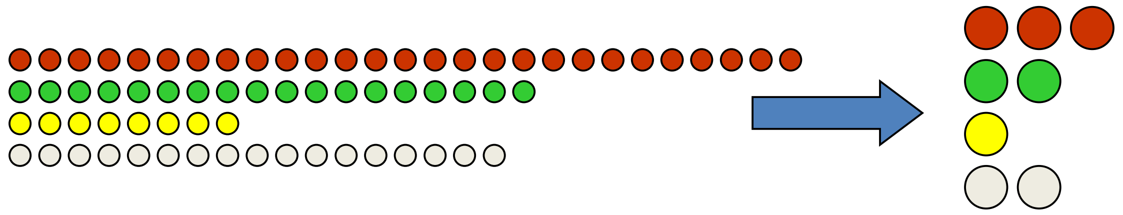
- Research papers on applications of sampling / sketches
  - TopicSketch: Real-time Bursty Topic Detection from Twitter [PDF]
  - Streaming Anomaly Detection Using Randomized Matrix Sketching [PDF]
  - Counting YouTube Videos via Random Prefix Sampling [PDF]

Processing Data Streams: Motivation

- A growing number of applications generate streams of data
  - Performance measurements in network monitoring and traffic management
  - Call detail records in telecommunications
  - Transactions in retail chains, ATM operations in banks
  - Log records generated by Web Servers
  - Sensor network data
  - Search queries (Google) , Twitter feed, etc.
- Application characteristics
  - Massive volumes of data (several terabytes)
  - Records arrive at a rapid rate
  - Cannot store entire stream
- **Goal:** Mine patterns, process queries and compute statistics on data streams in real-time.

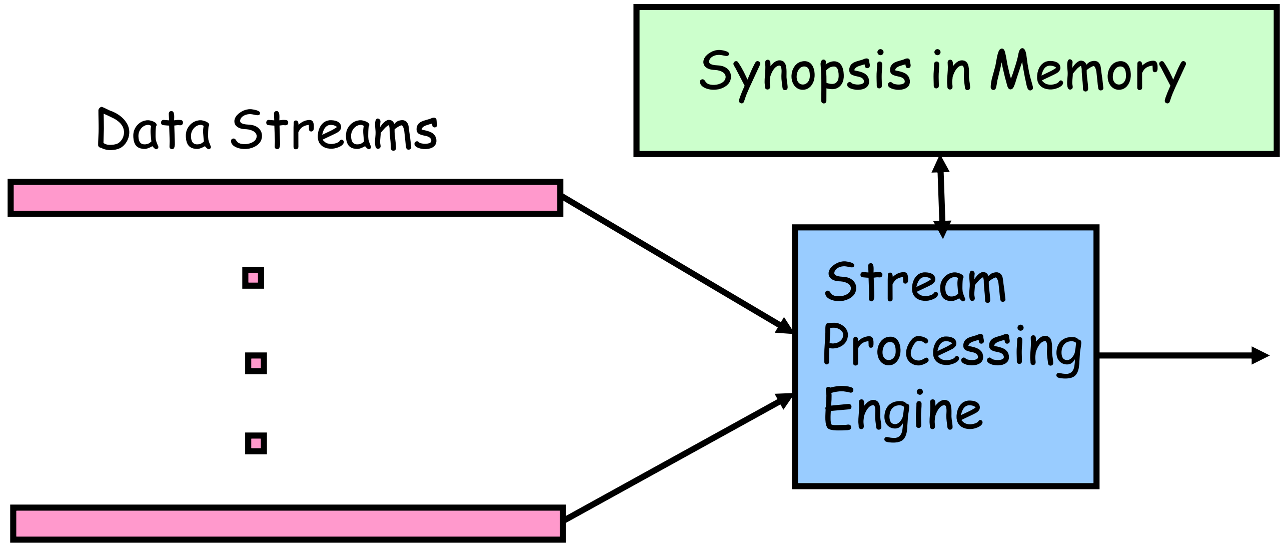
Why Sample?

- Sampling has an intuitive semantics
  - We obtain a smaller data set with the same structure
- Estimating a sample is often straightforward
  - Run the analysis on the sample that you would on the full data
  - Some rescaling/reweighting may be necessary
- Sampling is general and agnostic to the analysis to be done
  - Other summary methods only work for certain computations
  - Though sampling can be tuned to optimize some criteria
- Sampling is (usually) easy to understand
  - So prevalent that we have an intuition about sampling



Data Streams: Computation Model

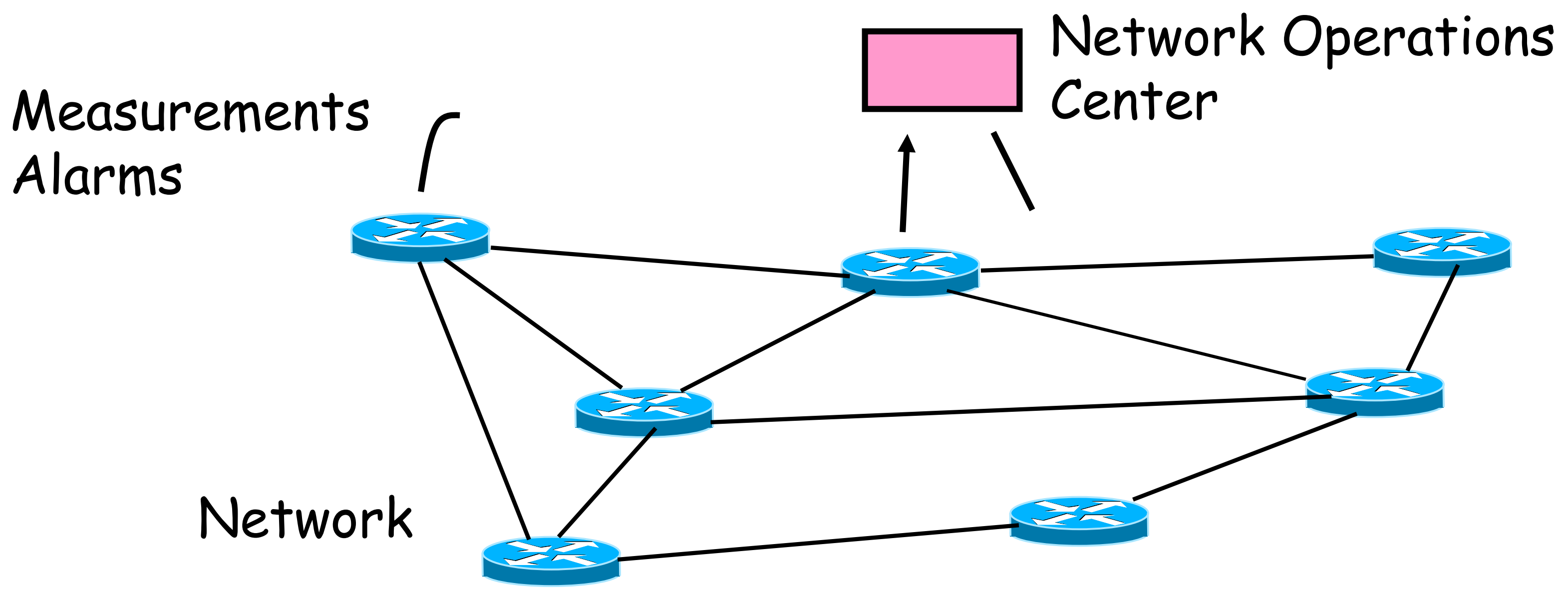
- A data stream is a (massive) sequence of elements: $e_1, \ldots, e_n$



- Stream processing requirements
  - **Single pass:** Each record is examined at most once
  - **Bounded storage:** Limited Memory (M) for storing synopsis
  - **Real-time:** Per record processing time (to maintain synopsis) must be low

Network Management Application

- Network Management involves monitoring and configuring network hardware and software to ensure smooth operation
  - Monitor link bandwidth usage, estimate traffic demands
  - Quickly detect faults, congestion and isolate root cause
  - Load balancing, improve utilization of network resources



Example: IP Network Measurement Data

- IP session data (collected using NetFlow)

Source	Destination	Duration	Bytes	Protocol
10.1.0.2	16.2.3.7	12	20K	http
18.6.7.1	12.4.0.3	16	24K	http
13.9.4.3	11.6.8.2	15	20K	http
15.2.2.9	17.1.2.1	19	40K	http
12.4.3.8	14.8.7.4	26	58K	http
10.5.1.3	13.0.0.1	27	100K	ftp
11.1.0.6	10.3.4.5	32	300K	ftp
19.7.1.2	16.5.5.8	18	80K	ftp

- AT&T collects 100 GBs of NetFlow data each day!

Network Data Processing

- Traffic estimation / analysis
  - What fraction network IP addresses are active?
  - List the top 100 IP addresses in terms of traffic
  - What is the average duration of an IP session?
  - What is the median of the number of bytes in each IP session?

- Fraud detection / Denial of Service
  - List all sessions that transmitted more than 1000 bytes
  - Identify all sessions whose duration was more than twice the normal
  - List all IP addresses that have witnessed a sudden spike in traffic
  - Identify IP addresses involved in more than 1000 sessions

Other Applications

- **Mining query streams**
  - Google wants to know what queries are more frequent today than yesterday
- **Mining click streams**
  - Yahoo wants to know which of its pages are getting an unusual number of hits in the past hour
- **Mining social network news feeds**
  - E.g., look for trending topics on Twitter, Facebook
- **Sensor Networks**
  - Many sensors feeding into a central controller
- **Telephone call records**
  - Data feeds into customer bills as well as settlements between telephone companies

Data Stream Processing Algorithms

- Generally, algorithms compute approximate answers
  - Difficult to compute answers accurately with limited memory

- Approximate answers - Deterministic bounds
  - Algorithms only compute an approximate answer, but bounds on error

- Approximate answers - Probabilistic bounds
  - Algorithms compute an approximate answer with high probability

ε

- Single-pass algorithms for processing streams also applicable to (massive) terabyte databases!

Next

- Motivation
- Sampling Problems
  - Stream Sampling
  - Filtering Streams
  - Counting Distinct Element in a Stream
  - Moments

- Research papers on applications of sampling / sketches
  - TopicSketch: Real-time Bursty Topic Detection from Twitter [PDF]
  - Streaming Anomaly Detection Using Randomized Matrix Sketching [PDF]
  - Counting YouTube Videos via Random Prefix Sampling [PDF]

Outline

- Motivation
- Sampling Problems
  - Stream Sampling
  - Filtering Streams
  - Counting Distinct Element in a Stream
  - Moments

- Research papers on applications of sampling / sketches
  - TopicSketch: Real-time Bursty Topic Detection from Twitter [PDF]
  - Streaming Anomaly Detection Using Randomized Matrix Sketching [PDF]
  - Counting YouTube Videos via Random Prefix Sampling [PDF]

PROBLEM 1 :

Sampling from a Data Stream: Sampling a fixed proportion

Sampling from a Data Stream

- **Problem:** When data is arriving too rapidly and we cannot store the entire stream (high volume and velocity) then we may have to consider sampling the stream. The sample must be unbiased and should preserve the properties / query answers we want to examine.

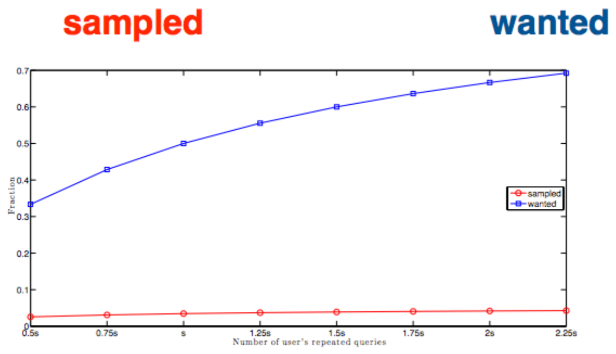
- **Motivating Example:**
  - Consider a search engine like Google which receives a stream of search queries continuously. Suppose Google would like to answer, *what fraction of queries were unique in the past month.*
  - **Naïve Approach:** We may decide to store only a fraction (lets say 1/10th) of the stream in which the elements stored is chosen at random. However, this sampling method will not generate a sample that accurately reflects the fraction of unique queries in the stream.

Sampling from a Data Stream (Cont.)

- **Simple question:** What fraction of queries by an average search engine user are duplicates?
  - Suppose each user issues x queries once and d queries twice (total of x+2d queries)
    - Correct answer: *d/(x+d)*
- **Proposed solution:** We keep 10% of the queries
  - # queries sampled that were issued once: x/10
  - # queries sampled twice that were issued twice: d / 10 * 10 = d/100
  - # queries sampled once that were issued twice : $-\dfrac{\frac{d}{100}}{\frac{d}{100}+\frac{x}{10}+\frac{18d}{100}} = \dfrac{d}{10x+19d}$

# queries sampled twice

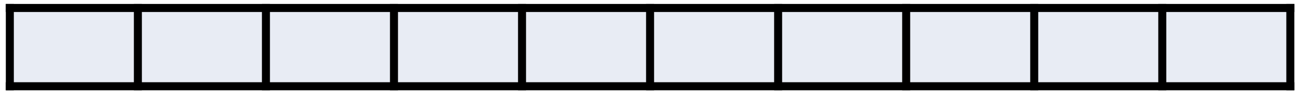
All queries sampled



Solution: Sample Users

- Solution:
  - Pick *1/10th* of users and maintain all their searches in the sample
  - Use a hash function that hashes the user name or user id uniformly into *10* buckets
  - Maintain users and their queries that hash to 1st bucket (to get 1/10th)

- To get a sample of **a/b** fraction of the stream:
  - Hash each tuple's key uniformly into *b* buckets
  - Pick the tuple if its hash value is at most *a*



Hash table with **b** buckets, pick the tuple if its hash value is at most **a.**

How to generate a 30% sample?

Hash into b=10 buckets, take the tuple if it hashes to one of the first 3 buckets

PROBLEM 2 :

Sampling from a Data Stream: Sampling a fixed size

Controlling the Sample Size

- **Problem:** What if the total sample size is limited, and you don't know the number of elements in the stream (n)?

- **Solution:** *Reservoir sampling*
  - Add each new element to S with probability *M / n*, where *M* is the fixed size sample and *n* is the current number of stream elements.

  - **Example:** Lets say you want to sample 1K elements evenly distributed from a stream.
  - Create a reservoir array/list of size 1K and fill with the first 1K elements of the stream.
  - Next, we will process the i^{th} element (with i=1001)
  - Generate a random number between 0 and 1, if its less than 1000 / i, then add element i.
  - If element i is to be added, then choose a random element from reservoir to be replaced with this element.

Weighted Reservoir Sampling Variation

- **Problem:** What if we want to sample a fixed size from a weighted distribution, where each element has a given weight associated with it in the stream?

- **Solution:**
  - As the stream is processed, assign each item a “key”.
  - For each item in the stream i, let k_i be the key and w_i be the weight of that item, and u_i be a random number between 0 and 1.
  - The key k_i is a random number to the nth root where n is weight of that item in the stream:
    - $k_i = u_i^{\frac{1}{w_i}}$
  - At any step i, we simply keep the top n elements ordered by their keys (k_i), where n is the size of the sample.

PROBLEM 3 :

Filtering elements from a stream: Bloom Filters

Filtering Streams

- **Problem:** Lets say we would like to filter elements based on some criterion or against a blacklist. This blacklist can be large cannot be stored in memory, thus we need a different approach.

- **Example:**
  - Consider a company called Proofpoint, a cybersecurity firm that provides security and data protection solutions to other companies …. namely Fourtune 500 companies.
  - Proofpoint must filter malicous emails, which can be a complex task that has mulitple steps.
  - One easy step in this process is passing through whitelisted emails / ip addresses (those emails that are deamed to be non-spam). However this list of whitelisted emails can be quite large and cannot be stored in main memory to compare against.

- **Solution:** use Bloom Filters
  - Have huge advantage over other data structures... Compact storage

Bloom Filters

- A Bloom Filter consists of :
  - An array of *n* bits, initially all *0's*
  - A collection of *k* hash functions $h_1, h_2, \dots, h_k$. Each hash function maps key values to *n* buckets, corresponding to the *n* bits of the bit-array.
  - A set **S** (ex. Whitelist of email address) of *m* key values.

- The purpose of Bloom Filter is to allow elements whose keys are in S, while rejecting most of the stream elements whose keys are not in S.
  - Set bit array (of size *n*) to all *0* bits.
  - Take each key value in S, and hash it using *k* hash functions h_1(Key), h_2(Key), …. h_k(Key)
  - If all set bits are 1's, then let the element through, otherwise if one or more bits are 0, then the Key could not be in S so reject the element.

Bloom Filters Analysis

- The disadvantage of Bloom Filters is that it allows for false positives, i.e. items that were not in set S (in our example, whitelisted emails).

- Assume we have n keys, and a filter of size m with k hash functions.
- The probability that a particular bit is still 0 is :
- $p_0 = \left(1 - \frac{1}{m}\right)^{kn}$

- The probability of a false positive is given by :

- $p_{err} = (1 - p_0)^k$

Hence, we can tweak the false positive rate by increasing the number of hash function utlized.

PROBLEM 4 :

Counting Distinct Elements in a Stream : Flajolet – Martin Algorithm

Counting Frequency of Elements in a Stream: Count-Min Sketch

Sketches

- The algorithm we will discuss next belong to a type of class of algorithms called “sketches”.
- Sketches are in-memory data structures that contain compact, often lossy, synopses of the streaming data.
- These synopses are designed to capture temporal properties of the stream so that queries can be answered.
  - Ex. Finding the most frequent occurring items in a stream
  - Ex. Finding the number of distinct elements in a stream

Count-Distinct Problem

- **Problem:** Given a stream of elements, we would like to know how many different elements have appeared in the stream, counting either from the start of the stream or a pre-defined time in the past.

- **Example:**
  - Amazon would like to gather statistics on how many unique users it has seen in each given month.
  - The number of distinct users is quite large, and we cannot store this list of all users who have logged-in to Amazon.

- **Solution:** Use Flajolet – Martin Algorithm [FM85]
  - Estimate the count in an unbiased way
  - Accept that the count may have a little error, but limit the probability that the error is large.
  - If the stream of size n and m of those elements are unique, the algorithm runs in O(n) time and uses O(log (m)) space.

Flajolet – Martin Algorithm (Cont.)

- Pick a hash function ***h*** that maps each of the *n* elements to values in the range $[0, 2^L - 1]$, where the outputs are uniformly distributed.
  - Note the set of integers $[0, 2^L-1]$ correspond to binary strings of length L

- For each stream element *x*, let *r(x)* be the number of trailing 0's in *h(x)*.
  - I.e. r(x) is the position of the least-significant bit in the binary representation of h(x)
  - Ex: say h(x) = 12, since 12 is 1100 in binary, r(x) = 2.
  - Ex: say h(x) = 13, since 13 is 1101 in binary, r(x) = 0

- Given the output of h(x) is uniformly distributed, the probability of h(x) ending in k zeros is 2^{-k}

Flajolet – Martin Algorithm (Cont.)

<u>Basic Algorithm:</u>

1. Initialize a bit-vector BITMAP of length L (of all zeros)
2. For each element x in the stream:
    1. Calculate i = hash(x)
    2. Let p(i) be the least-significant bit in the binary representation of i
    3. Set BITMAP[i] = 1
3. Let R denote the smallest index i such that BITMAP[i] = 0
4. Estimate the distinct count of the stream as 2^R

It seems a bit odd to say that each stream has a distinct count that is a power of 2. So, usually, this is performed with k hash function and the results are combined from the different runs.

Flajolet – Martin Algorithm (Cont.)

- **Basic Idea:**
  - Run the algorithm multiple times with k hash functions, then combine the results.
  - But the question is how to combine the results.
- **Idea 1 :** Take the mean of the results
  - But the problem is that this method is susceptible to outliners
- **Idea 2 :** Take the median of the results
  - But the problem is that this method can only take form 2^R
- **Common Solution:** median of means
  - Create k * l hash function and split into k groups
  - For each group, use median to aggregate the results of the l functions in that group
  - Then, take the mean of the k group estimates
- **HyperLogLog** is another method for the count distinct problem, which is derived from Flajolet – Martin Algorithm.

Find Frequency Vector Problem

- **Problem:** Given a stream of elements, how do we compute the frequency of a particular element within a certain probability of accuracy.

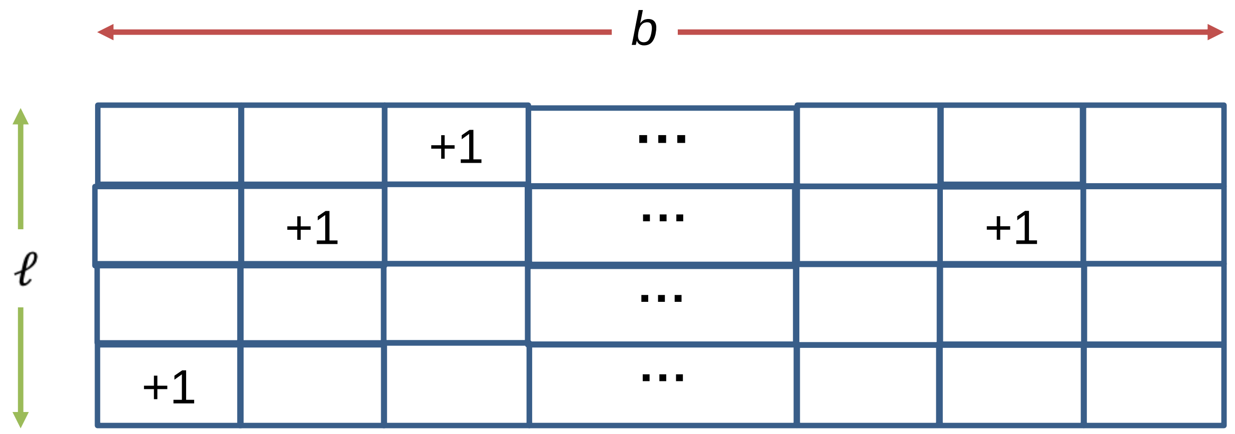
- **Example:**
  - On a network, we may want to monitor network traffic to identify the high hitters?
  - On a site, we may want to monitor the requests (IP addresses) to identify high hitters that might potentially be malicious requests on the site.

- **Solution:** use count–min sketch (CM sketch) [CM05]

Count-Min Sketch

- Its based on Bloom Filters
  - Recall, Bloom Filters (BF) are data structures that are responsible for remembering what's been inserted.
  - Basically Bloom Filters are similar to hash tables, but require much less space.

- Count-Min sketch supports two operations: Include(x) and Count(x)
  - Count(x) returns the number of times Include(x) is called
- The count-min sketch has two parameters, the number of buckets ***b*** and the number of hash functions ℓ.
- The goal is to fill the CMS 2-D array



```
for each object:
    for i = 1, 2, … , $\ell$ :
            increment CMS [ i ] [ $h_i(x)$ ]
```

Assuming h(x) can be evaluated in constant time, the running time of the operation is $O(\ell)$

Count-Min Sketch

-
- CMS [i] [h_i(x)] $\geq f_x$, where f_x denotes the frequency of object x
  - Thus, we are able to answer point queries of the form, what is the frequency of object x

- There is the possibility of collisions
  - So while CMS [i] [h_i(x)] cannot underestimate f_x, it generally overestimates f_x.

- The ℓ rows of the count-min sketch give ℓ different estimates of f_x , how should we aggregate these estimates?
  - When dealing with unbiased estimates one usually averages them.
  - Here, our estimates are not unbiased, but rather suffer one-sided error — all of them can only be bigger than the number f_x we want to estimate.
  - The *smallest* of the estimates is clearly the best estimate
  - Thus, we modify it to $min_{i=1}^{\ell}$CMS[i][h_i(x)]

Analysis of Count-Min

- Count-Min takes an accuracy parameter ϵ and an error probability parameter δ.
- The values of ℓ and b (2-D array), are based on ϵ and δ:

Initialize

```
1.  C [1 … ℓ ] [1 … b ] ← ℓ = 2/ϵ  and b = ⌈log 1/δ⌉
2.  Choose ℓ independent hash functions  h_1, h_2, … h_ℓ
Process (x):
For i = 1 to ℓ do :
        C [i][ h_i(x) ] ← C[i][h_i(x)] + 1

Output
On query a, report f̂_a = min_{1 ≤ i ≤ ℓ} C [i][ h_i(x) ]
```

$$f_a \leq \hat{f}_a \leq f_a + \epsilon$$

The error is bounded by ϵ and the right inequality holds with probability δ.

PROBLEM 5

Estimating Moments: AMS method

Moments

- Suppose a stream has elements chosen from a set A of N values

- Let m_i be the frequency of the ith element for any i. The kth order moment of the stream is:
  - kth moment is $\sum_{i \in A} (m_i)^k$

- 0th moment = number of distinct elements
  - Problem we just considered
- 1st moment = count of the numbers of elements = length of the stream
  - Easy to compute … sum of all m_i
- 2nd moment
  - a measure of the unevenness of the distribution
  - Sum of all m_i^2 (surprise number S)

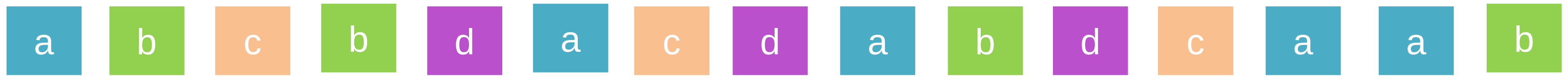
Example: A stream with 100 elements
(S1) 10 elements occur 9 times , 1 element occurs 10 times
(S2) 10 elements occur 1 time, 1 element occurs 90 times
$10 \times 9^2 + 1 \times 10^2 = 910$
$1 \times 90^2 + 10 \times 1^2 = 8110$

AMS Method

- AMS method works for all moments, but we will look at the 2nd moment
- We pick and keep track of many variables X, and for each variable X
  - An integer X.value, which is the value of the variable. To determine the value of a variable X, we choose a position in the stream between 1 and n, uniformly and at random.
  - Set X.element to be the element found there, and initialize X.value to 1. As we read the stream, add 1 to X.value each time we encounter another occurrence of X.element

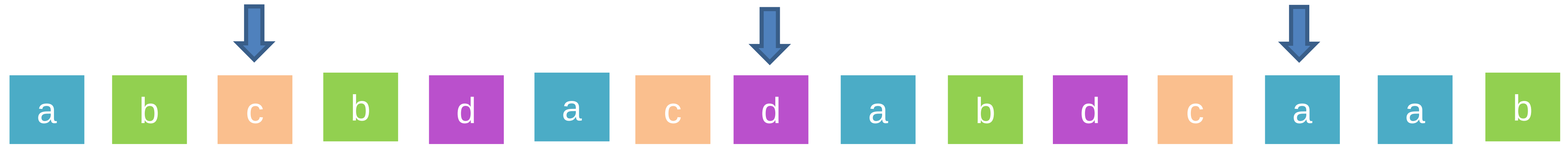
AMS algorithm for 2nd order moments

- Lets define:
  - X = (element, value)
  - X.element : element of universal set
  - X.value : counter of X.element in the stream starting at a randomly chosen position
  - Example



  - n = 15
  - $m_a^2 + m_b^2 + m_c^2 + m_d^2 = 5^2 + 4^2 + 3^2 + 3^2 = 59$

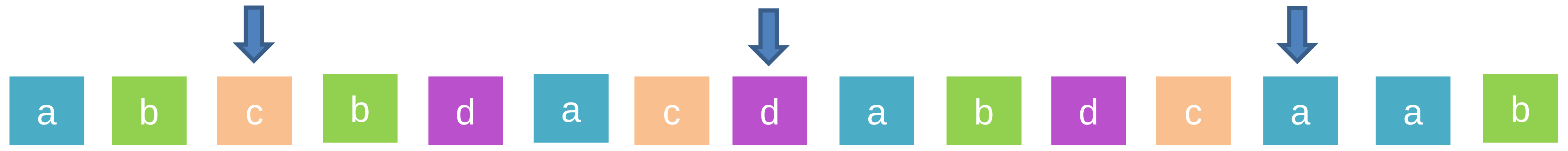
AMS algorithm for 2nd order moments (Cont.)



- Randomly pick 3 positions (3 variables) to compute the 2nd order) from the stream with known length.

- At 3rd position of the stream, first variable is set $X_1 = (c, 1)$
- When the second c is encountered, first variable is set to $X_1 = (c, 2)$
- At 8th position of the stream, second variable is set $X_2 = (d, 1)$
- When the second d is encountered, second variable is set to $X_2 = (d, 2)$
- When the third c is encountered, first variable is set to $X_1 = (c, 3)$
- At 13th position, third variable is set to $X_3 = (a, 1)$
- When the second a is encountered, third variable is set to $X_3 = (a, 2)$

AMS algorithm for 2nd order moments (Cont.)



- Randomly pick 3 positions (3 variables) to compute the 2nd order) from the stream with known length.
- Scan scanning the stream, until we reach the first selected position.
- Update 1st variable with variable encountered: X_1= (c, 1)
- When the second c is encountered, first variable is set to X_1 = (c, 2)
- At 8th position of the stream, second variable is set X_2 = (d, 1)
- When the second d is encountered, second variable is set to X_2 = (d, 2)
- When the third c is encountered, first variable is set to X_1 = (c, 3)
- At 13th position, third variable is set to X_3 = (a, 1)
- When the second a is encountered, third variable is set to X_3 = (a, 2)

AMS algorithm for 2nd order moments (Cont.)

- Estimate of the 2nd order moment from any X = (element, value)
  - n * (2 * X.value – 1)

- Applied to our example:
  - Estimate from X_1 : 15 * (2 * 3 -1) = 75
  - Estimate from X_2 : 15 * (2 * 2 -1) = 45
  - Estimate from X_3 : 15 * (2 * 2 -1) = 45

  - AVG(X_1, X_2, X_3) = 55

Approximate solution… very close to actual value of 59

Sampling / Sketches Libraries

- Implementation of Sampling and Sketches in Big Data tools is still elementary, but there is limited support.
  - DataSketches – Java Library
    - https://datasketches.github.io/
  - Spark Sketch
    - https://github.com/apache/spark/tree/master/common/sketch

Additional References

- An Improved Data Stream Summary: The Count-Min Sketch and its Applications - Cormode & Muthukrishnan : The paper that introduced count min sketch
- Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches by Graham Cormode, Minos Garofalakis, Peter J. Haas and Chris Jermaine . Describes basic principles and recent developments in approximate query processing. It focuses on four key synopses: random samples, histograms, wavelets, and sketches. It considers issues such as accuracy, space and time efficiency, optimality, practicality, range of applicability, error bounds on query answers, and incremental maintenance. It also discusses the trade-offs between the different synopsis types.
- http://theory.stanford.edu/~tim/s15/l/l2.pdf
- http://www.cs.dartmouth.edu/~ac/Teach/CS49-Fall11/Notes/lecnotes.pdf