Mining Data Streams

Liyao Xiang
xiangliyao.cn
Shanghai Jiao Tong University

- What is streaming data and its application?

Data Streams

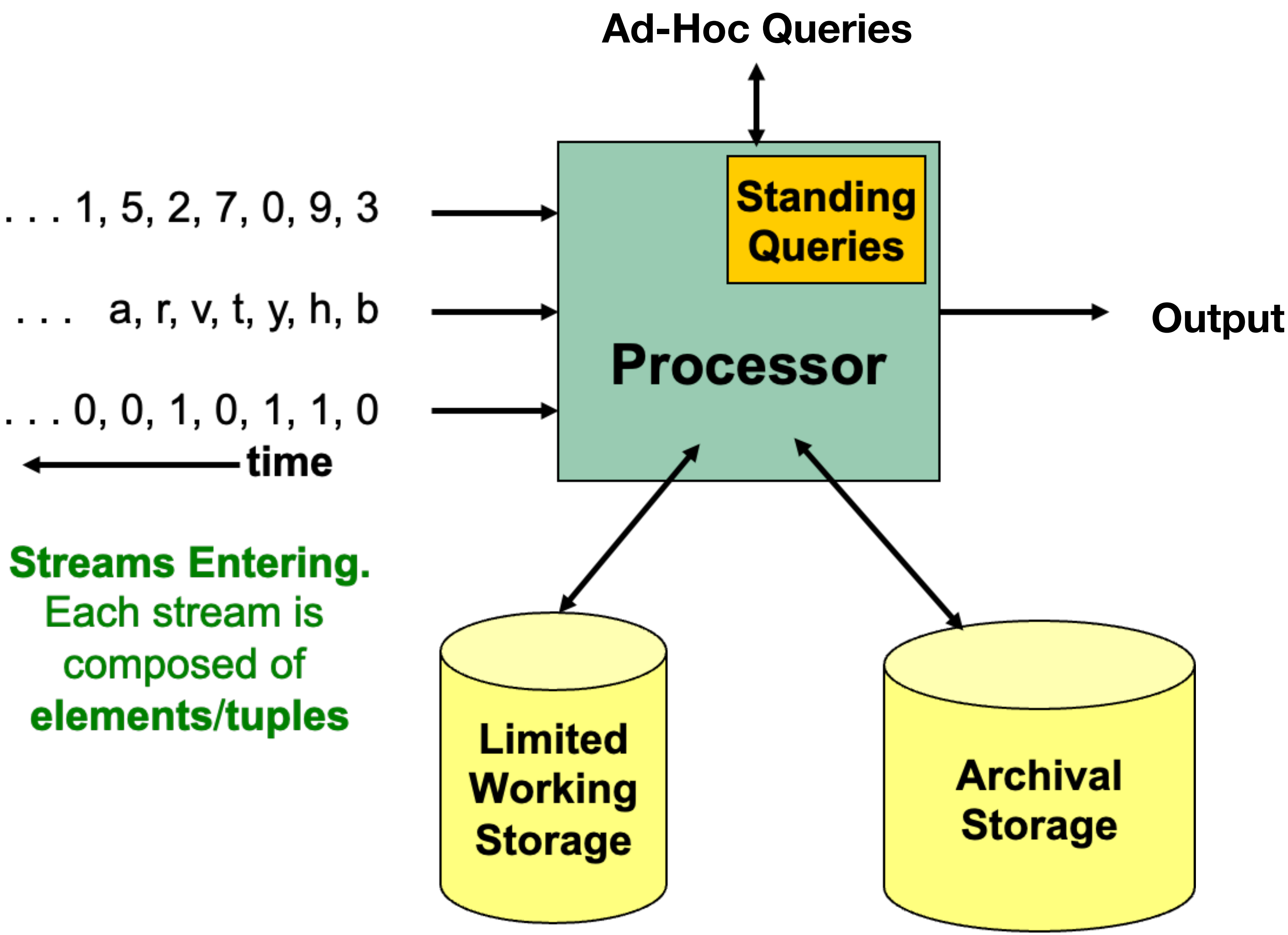
- In many data mining situations, we do not know the entire data set in advance
- **Stream Management** is important when the input rate is controlled externally:
  - Baidu queries
  - Click-through-rate on bilibili
  - Weibo status/blog updates
- We can think of the data as infinite and non-stationary (the distribution changes over time)

The Stream Model

- Input elements enter at a rapid rate, at one or more input ports (i.e., streams)
    - We call elements of the stream **tuples**
- The system cannot store the entire stream accessibly
- Problem: How do we make critical calculations about the stream using a limited amount of (secondary) memory?

Creating summary!

General Stream Processing Model



Example: SGD is a Streaming Alg.

- Stochastic Gradient Descent (SGD) is an example of a stream alg.
- In Machine Learning we call this: Online Learning
  - Allows for modeling problems where we have a **continuous stream of data**
  - We want an algorithm to learn from it and slowly adapt to the changes in data
- Idea: Do slow updates to the model
  - SGD makes small updates
  - So: First train the classifier on training data
  - Then: For every example from the stream, we slightly update the model (using small learning rate)

What are the Queries?

- What is the data distribution of the stream?
- How many items featured by XXX in the last K elements of the stream?
- What are the items with property YY from the stream?
- How many distinctive items in the last K elements of the stream?
- What are the statistics of the last K elements?
- What is the most frequently appearing items in the stream?
- …

Applications

- Mining query streams
  - Baidu wants to know what queries are more frequent today than yesterday
- Mining click streams
  - Bilibili wants to know which of the videos clips are getting an unusual high number of hits in the past hour
- Mining social network news feeds
  - Weibo looks for trending topics

Applications

- Sensor networks where many sensors' feeding data into a central controller
  - Want to detect unusual events
- IP packets monitored at a switch
  - Gather information for optimal routing
  - Detect denial-of-service attacks

- How do we sample data from a stream?

Sampling from a Data Stream

- Since we **can not store the entire stream**, one obvious approach is to store **a sample**
- How do we sample from the stream just **like we have seen the entire stream**?
  - At any "time" **k** we would like a random sample of **s** elements
  - **What is the property of the sample we want to maintain?**
    - For all time steps **k**, each of **k** elements seen so far has equal prob. of being sampled
- Two different problems:
  1. Sample a fixed proportion
  2. Sample a fixed-sized set

Sampling a Fixed Proportion

- Problem 1: Sample a fixed proportion of elements in the stream (say 1 in 10)
- Scenario: Search engine query stream
    - Stream of tuples: (user, query, time)
    - Answer questions such as: How often did a user run the same query through the search engine in a single day?
    - Have space to store 1/10th of query stream
- Naïve solution:
    - Generate a random integer in [0..9] for each query
- Store the query if the integer is 0, otherwise discard

Problem with Naive Approach

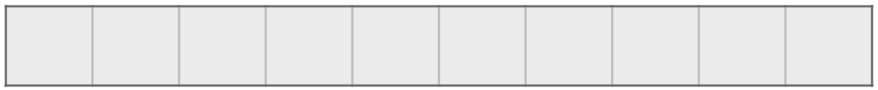
- Simple question: What fraction of queries by an average search engine user are duplicates?
  - Suppose each user issues **x** queries once and **d** queries twice (total of **x+2d** queries)
- Proposed solution: We keep 10% of the queries
  - Sample will contain **x/10** of the singleton queries and **2d/10** of the duplicate queries at least once
  - But only **d/100** pairs of duplicates
    - $d/100 = 1/10 \cdot 1/10 \cdot d$
  - Of **d** "duplicates" **18d/100** appear exactly once
    - $18d/100 = ((1/10 \cdot 9/10)+(9/10 \cdot 1/10)) \cdot d$
- What fraction of queries by an average search engine user are duplicates by the sample-based approach?

Sampling Users

- How about sampling users?
  - Pick 1/10th of **users** and take all their searches in the sample
  - Use a hash function that hashes the user name or user id uniformly into 10 buckets
- What fraction of queries by an average search engine user are duplicates by the user-based approach?

Generalized Solution

- Stream of tuples with keys:
  - Key is some subset of each tuple's components
    - e.g., tuple is (user, search query, time); key is **user**
  - Choice of key depends on application
- 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

Sampling a fixed-size sample

- Problem 2: Fixed-size sample
  - As the stream grows, the sample is of fixed size
- Suppose we need to maintain a random sample S of size exactly s tuples
  - E.g., main memory size constraint is s
- Why? Don't know length of stream in advance
- Suppose at time n we have seen n items
  - Each item is in the sample S with equal probability s/n

How to think about the problem: say s = 2
Stream: a x c y z k c d e g…

At **n= 5,** each of the first 5 tuples is included in the sample **S** with equal prob.
At **n= 7,** each of the first 7 tuples is included in the sample **S** with equal prob.

An impractical solution would be to store all the n tuples seen so far and out of them pick s at random

Solution: Fixed Size Sample

- **Algorithm (a.k.a. Reservoir Sampling)**
  - Store all the first **s** elements of the stream to **S**
  - Suppose we have seen **n-1** elements, and now the **n**th element arrives (**n > s**)
    - With probability **s/n**, keep the **n**th element, else discard it
    - If we picked the **n**th element, then it replaces one of the **s** elements in the sample **S**, picked uniformly at random
- Claim: This algorithm maintains a sample **S** with the desired property:
  - After **n** elements, the sample contains each element seen so far with probability **s/n**

Proof

- We prove this by induction:
  - Assume that after **n** elements, the sample contains each element seen so far with probability **s/n**
  - We need to show that after seeing element **n+1** the sample maintains the property
    - Sample contains each element seen so far with probability **s/(n+1)**
- Base case:
  - After we see **n=s** elements the sample **S** has the desired property
    - Each out of **n=s** elements is in the sample with probability **s/s = 1**

Proof

- Inductive hypothesis: After **n** elements, the sample **S** contains each element seen so far with prob. **s/n**
- Now element **n+1** arrives
- Inductive step: For elements already in **S**, probability that the algorithm keeps it in **S** is:

$$\underbrace{\left(1-\frac{s}{n+1}\right)}_{\text{Element n+1 discarded}}+\underbrace{\left(\frac{s}{n+1}\right)}_{\text{Element n+1 not discarded}}\underbrace{\left(\frac{s-1}{s}\right)}_{\text{Element in the sample not picked}}=\frac{n}{n+1}$$

- So, at time **n**, tuples in **S** were there with prob. **s/n**
- Time **n -> n+1**, tuple stayed in **S** with prob. **n/(n+1)**
- So prob. tuple is in **S** at time **n+1 =** $\frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1}$

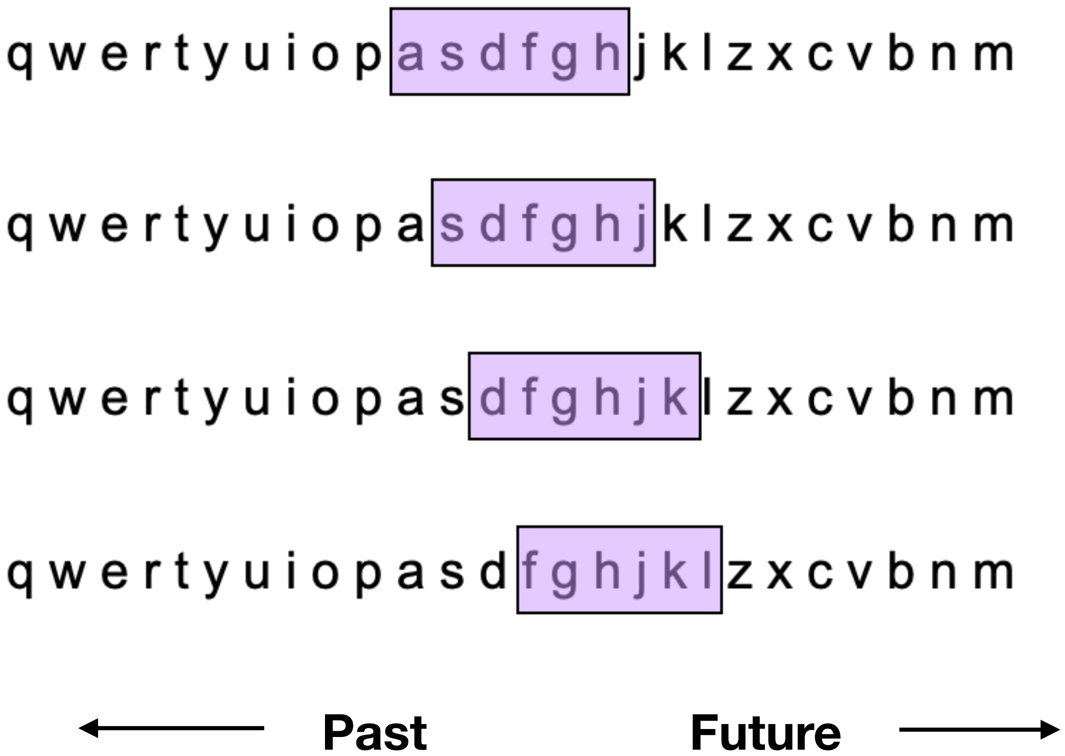
- How do we query over a sliding window?

Sliding Windows

- A useful model of stream processing is that queries are about a window of length **N** – the **N** most recent elements received
- Interesting case: **N** is so large that the data cannot be stored in memory, or even on disk
  - Or, there are so many streams that windows for all cannot be stored
- Example:
  - For every product **X** we keep 0/1 stream of whether that product was sold in **each** transaction
  - We want to answer queries: how many times have we sold **X** in the last **k** sales (cannot know **k** in advance)

Sliding Window: 1 Stream

- Sliding window on a single stream: **N = 6**



Counting Bits

- Problem:
    - Given a stream of **0**s and **1**s
    - Be prepared to answer queries of the form:
        - How many 1s are in the last **k** bits? where **k ≤ N**
- Obvious solution:
- Store the most recent **N** bits
    - When new bit comes in, discard the **N+1th** bit

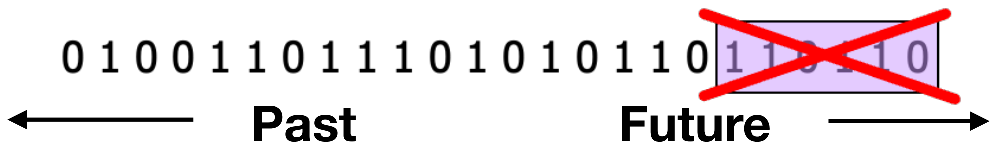
0 1 0 0 1 1 0 1 1 1 0 1 0 1 0 1 1 0 1 1 0 1 1 0

Suppose N = 6

Past **Future**

Counting Bits

- You can not get an **exact** answer without storing the **entire** window
- Real Problem: **What if we cannot afford to store N bits**?
  - E.g., we're processing 1 billion streams and N = 1 billion



- But we are happy with an approximate answer

An Attempt: Simple Solution

- Q: How many 1s are in the last **k** bits?
- A simple solution that does not really solve our problem:

  Uniformity assumption

  N

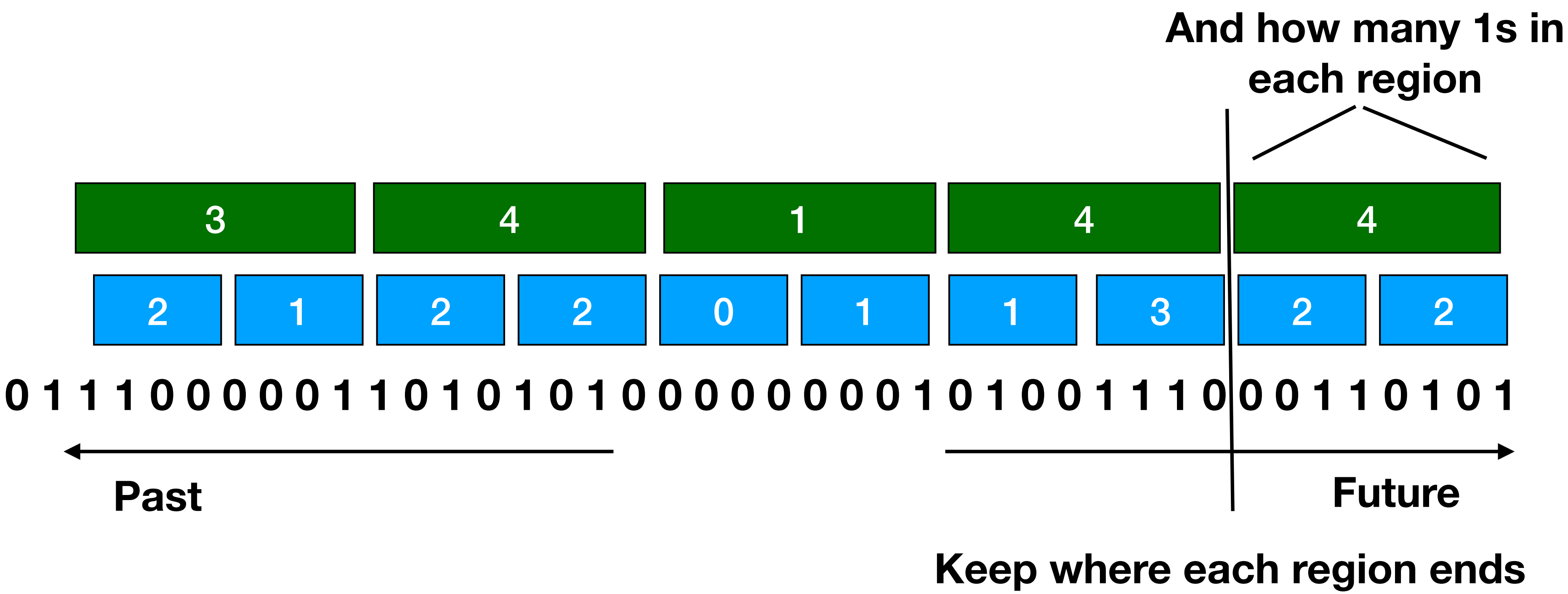
  0 1 0 0 1 1 1 0 0 0 1 0 1 0 0 1 0 0 0 1 0 1 1 0 1 1 0 1 1 1 0 0 1 0 1 0 1 1 0 0 1 1 0 1 0

  Past Future

- Maintain 2 counters:
  - S: number of 1s from the beginning of the stream
  - Z: number of 0s from the beginning of the stream
- How many 1s are in the last **k** bits? $k \cdot S/(S+Z)$
- But, what if stream is **non-uniform**?
  - What if distribution changes over time?

Idea: Uniform Windows

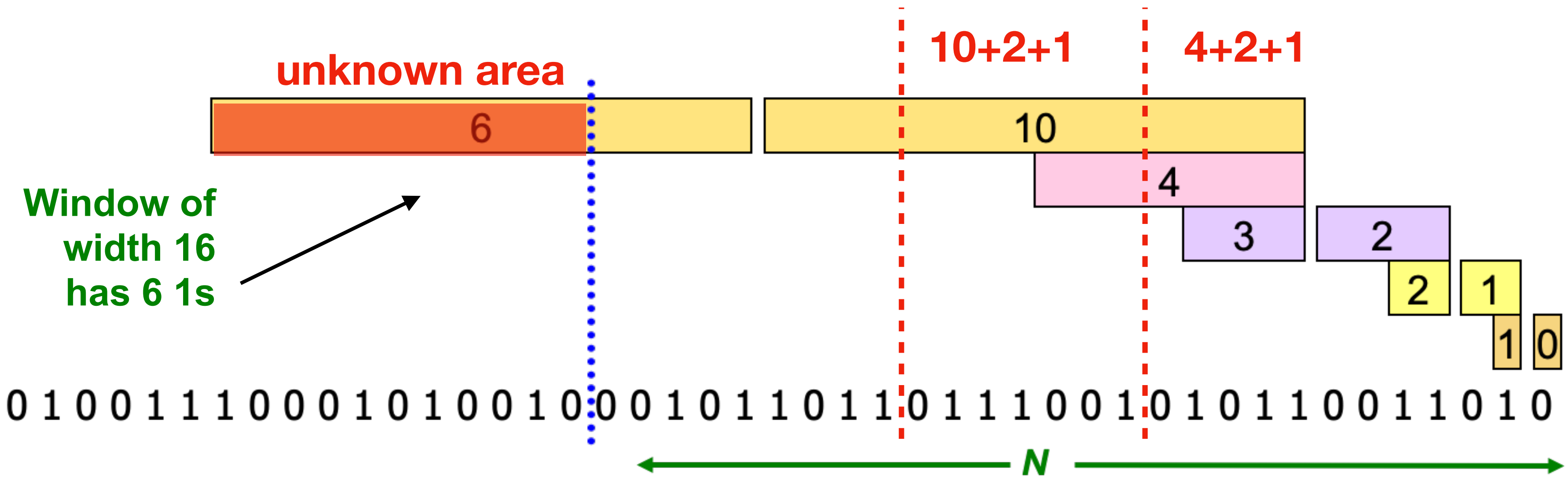
- Summarize **uniformly-sized** regions of the stream, looking backward



- But we usually weigh more on the recent data

Idea: Exponential Windows

- Summarize **exponentially increasing** regions of the stream, looking backward
  - Drop small regions if they begin at the same point as a larger region



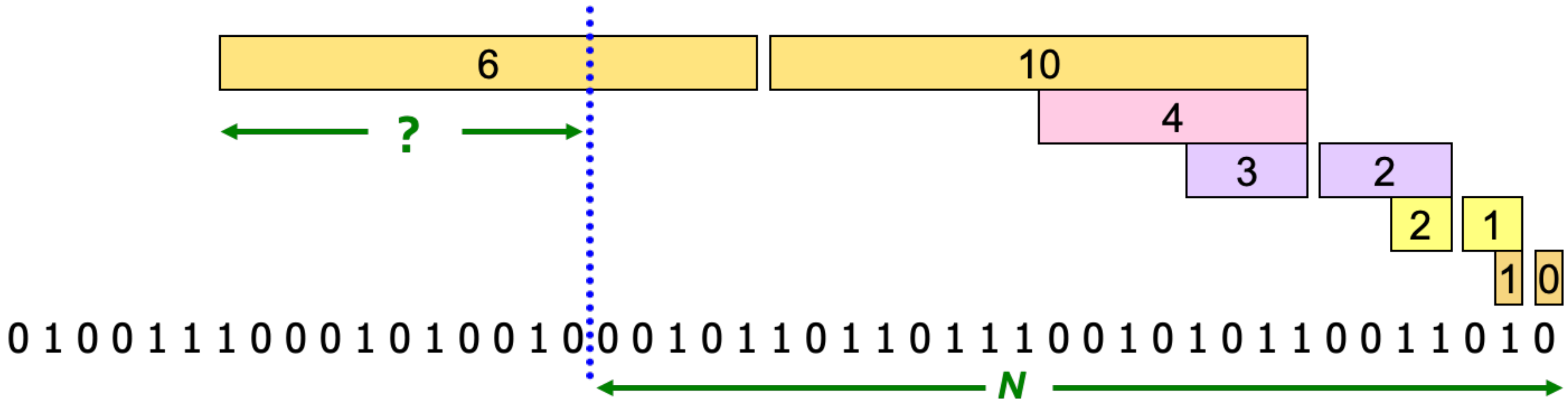
We can reconstruct the count of the last ***N*** bits, except we are not sure how many of the last **6 1s** are included in the ***N***

Advantages

- Stores only $O(\log^2 N)$ bits
  - $\boldsymbol{O}(\log \boldsymbol{N})$ counts of $\log_2 \boldsymbol{N}$ bits each
- Easy update as more bits enter
- Error in count no greater than the number of 1s in the "**unknown**" area

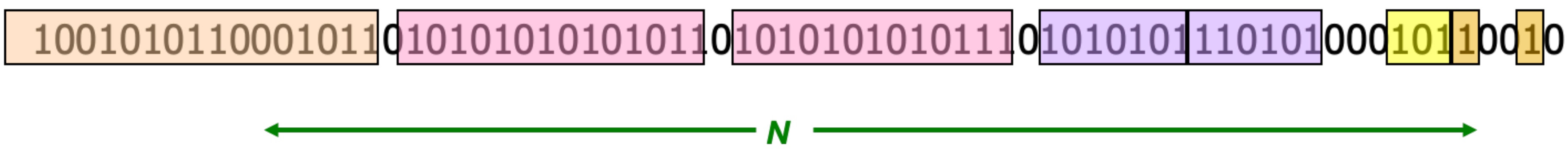
Drawbacks

- As long as the 1s are fairly evenly distributed, the error due to the unknown region is small – no more than 50%
- But it could be that all the 1s are in the unknown area at the end
- In that case, the error is unbounded!



Fixup: DGIM Method

- Idea: Instead of summarizing fixed-length blocks, summarize blocks with specific number of **1**s:
  - Let the block **sizes** (number of **1**s) increase exponentially
- When there are few **1**s in the window, block sizes stay small, so errors are small

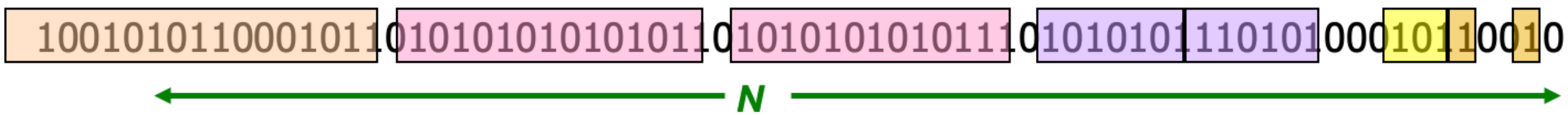


DGIM: Timestamps

- Each bit in the stream has a timestamp, starting 1, 2, …
- Record timestamps modulo N (the window size), so we can represent any relevant timestamp in $O(log_2 N)$ bits

DGIM: Buckets

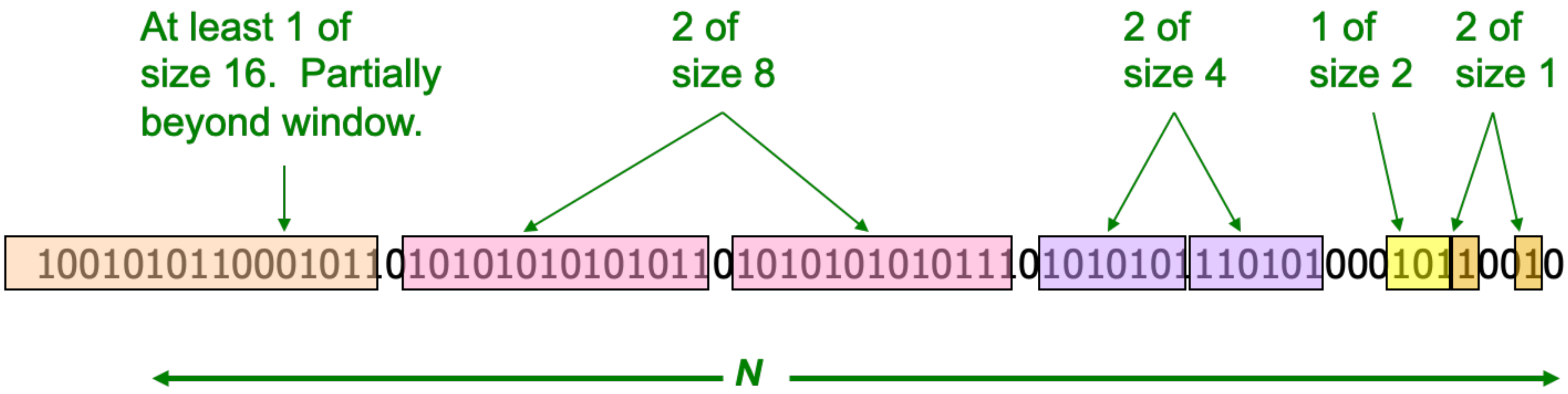
- A bucket in the DGIM method is a record consisting of:
    A. The timestamp of its end expressed in **[O(logN) bits]**
    B. The number of 1s between its beginning and end expressed in **[O(logN - 1) = O(logN)bits]**
- Constraint on buckets: Number of **1**s must be a power of **2**



Representing a Stream by Buckets

- Either one or two buckets with the same **power-of-2 number** of **1**s
- Buckets do not overlap in timestamps
- Buckets are sorted by size
  - Earlier buckets are not smaller than later buckets
- Buckets disappear when their end-time is **> N** time units in the past

Example: Bucketized Stream



- Three properties of buckets that are maintained:
  - Either one or two buckets with the same **power-of-2 number** of 1s
  - Buckets do not overlap in timestamps
  - Buckets are sorted by size

Updating Buckets

- When a new bit comes in, drop the last (oldest) bucket if its end-time is prior to **N** time units before the current time
- **2 cases**: Current bit is **0** or **1**
- If the current bit is 0: no other changes are needed
- If the current bit is 1:

  1. Create a new bucket of size 1, for just this bit
     - End timestamp = current time
  2. If there are now **three buckets of size 1**, combine the oldest two into **a bucket of size 2**
  3. If there are now **three buckets of size 2**, combine the oldest two into **a bucket of size 4**
  4. And so on …

Example: Updating Buckets

Current state of the stream:

1001010110001011 0 1010101010101011 0 101010101010111 0 1010101 110101 000 101 1 00 1 0

Bit of value 1 arrives

0010101100010110 1010101010101011 0 101010101010111 0 1010101 110101 000 101 1 00 1 0 1

Two orange buckets get merged into a yellow bucket

0010101100010110 1010101010101011 0 101010101010111 0 1010101 110101 000 101 1001 0 1

Next bit 1 arrives, new orange bucket is created, then 0 comes, then 1:

0101100010110 1010101010101011 0 101010101010111 0 1010101 110101 000 101 1001 0 1 1 0 1

Buckets get merged…

0101100010110 1010101010101011 0 101010101010111 0 1010101 110101 000 101 1001 0 11 0 1

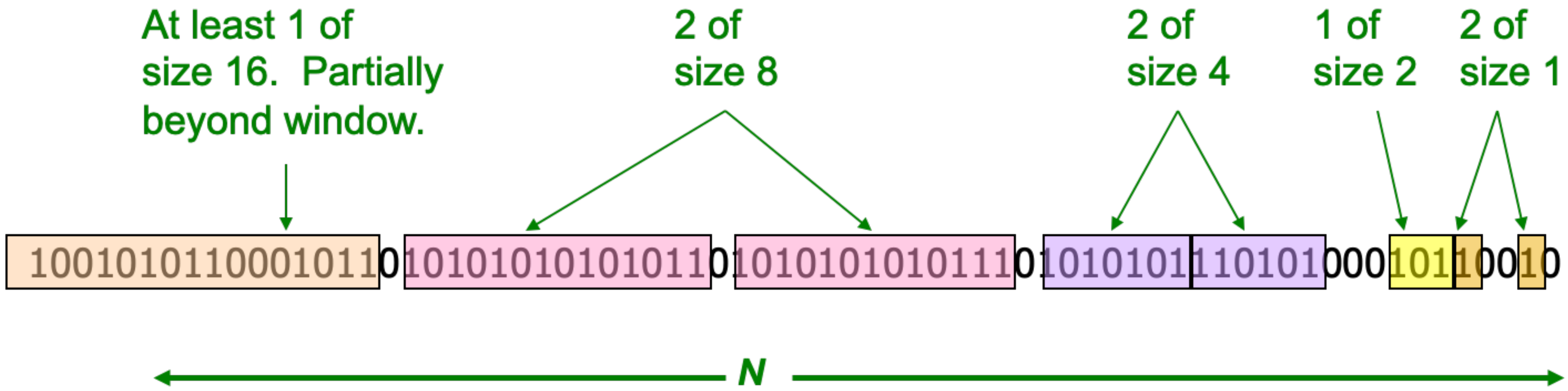
State of the buckets after merging

0101100010110 1010101010101011101010101010111 0 1010101110101 000 1011001 0 11 0 1

How to Query?

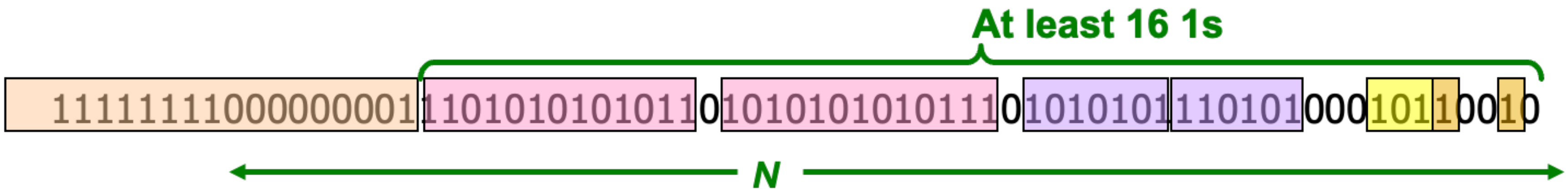
- To estimate the number of **1**s in the most recent **N** bits:
    1. Sum the sizes of all buckets but the last (note “size” means the number of 1s in the bucket)
    2. Add half the size of the last bucket
- Remember: We do not know how many 1s of the last bucket are still within the wanted window

Example: Bucketized Stream



Error Bound: Proof

- Why is error 50%? Let's prove it!
- Suppose the last bucket has size $\mathbf{2^r}$
- Then by assuming $\mathbf{2^{r-1}}$ (i.e., half) of its **1**s are still within the window, we make an error of at most $\mathbf{2^{r-1}}$
- Since there is at least one bucket of each of the sizes less than $\mathbf{2^r}$, the true sum is at least $\mathbf{1 + 2 + 4 + .. + 2^{r-1} = 2^r - 1}$
- Thus error is at most 50%



Further Reducing the Error

- Instead of maintaining **1** or **2** of each size bucket, we allow either **r-1** or **r** buckets (**r > 2**) except for the largest-sized buckets
    - We can have any number between **1** and **r** of the largest-sized buckets
- Error is at most **O(1/r)**
- By picking **r** appropriately, we can tradeoff between number of bits we store and the error

Can We Handle Stream of Integers?

- Stream of positive integers
- We want the sum of the last **k** elements
  - E.g.: Avg. price of last **k** sales
- A naive solution:

  1. If you know all have at most **m** bits
     - Treat **m** bits of each integer as a separate stream
     - Use DGIM to count **1**s in each integer
     - The sum = $\sum_{i=0}^{m-1} c_i 2^i$ c_i …estimated count for i-th bit

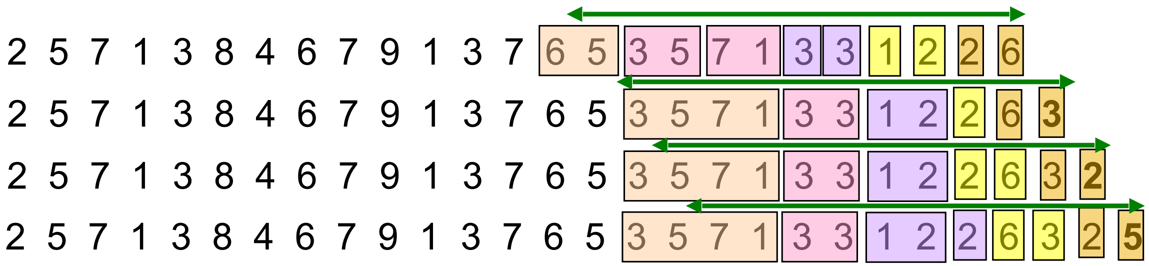
2^i: weigh each count differently

Can We Handle Stream of Integers?

- Stream of positive integers
- We want the sum of the last **k** elements
  - E.g.: Avg. price of last **k** sales
- Another solution:
  2. Use buckets to keep partial sums
     - Sum of elements in size **b** bucket is at most $\mathbf{2^b}$

Idea: Sum in each bucket is at most 2^b (unless bucket has only 1 integer)
Bucket sizes:

16 | 8 | 4 | 2 | 1

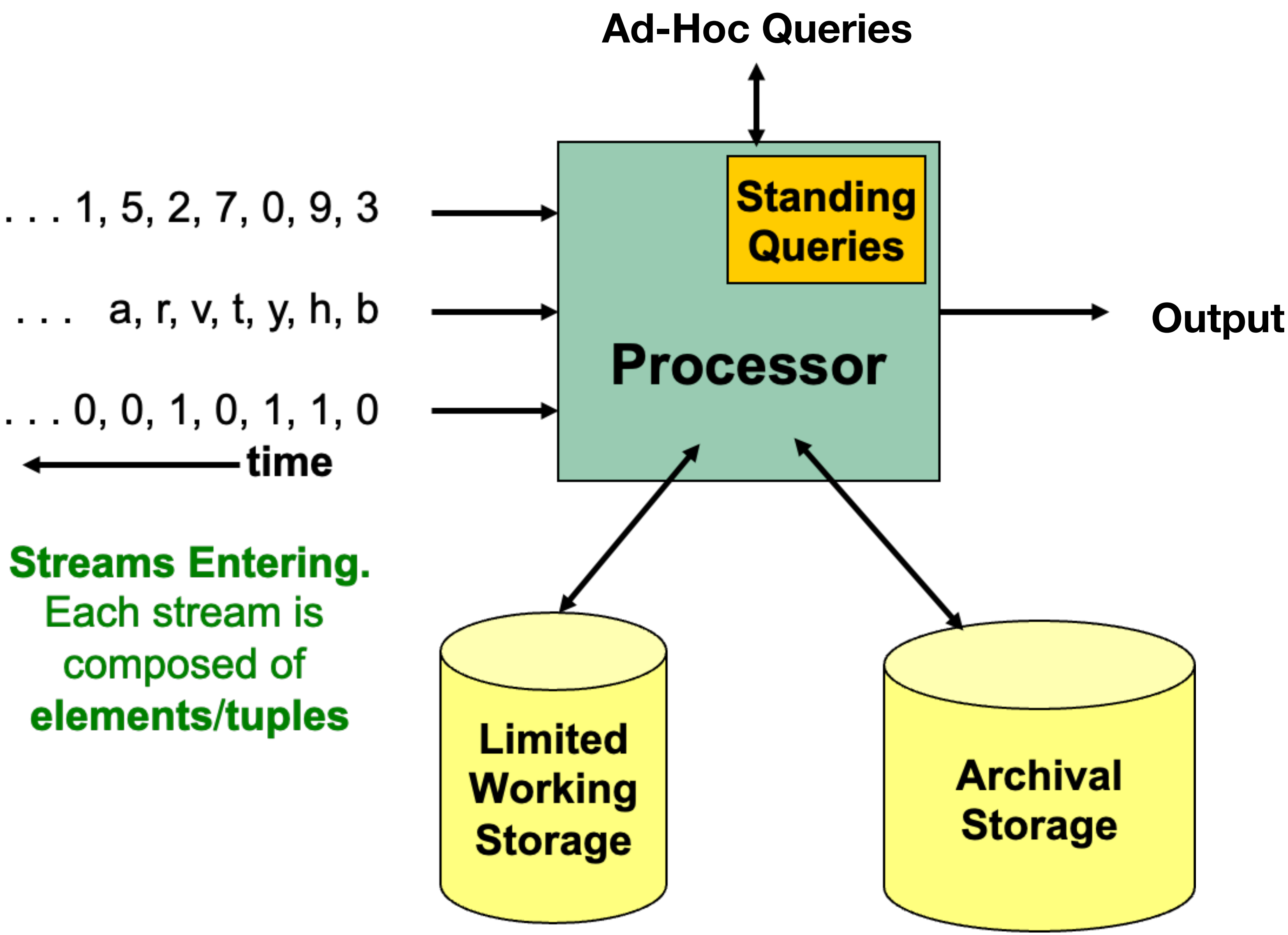


Each time, we only merge buckets of the same color

Summary

- Sampling a fixed proportion of a stream
  - Sample size grows as the stream grows
- Sampling a fixed-size sample
  - Reservoir sampling
- Counting the number of 1s in the last N elements
  - Exponentially increasing windows
  - Extensions:
    - Sums of integers in the last N elements

Review—General Stream Processing Model



Streams Entering. Each stream is composed of **elements/tuples**

Review—Sampling from a Data Stream

- Since we **can not store the entire stream**, one obvious approach is to store **a sample**
- How do we sample from the stream just **like we have seen the entire stream**?
  - At any "time" **k** we would like a random sample of **s** elements
  - **What is the property of the sample we want to maintain?**
    - For all time steps **k**, each of **k** elements seen so far has equal prob. of being sampled
- Two different problems:
  1. Sample a fixed proportion
  2. Sample a fixed-sized set

Review—Fixed Size Sample

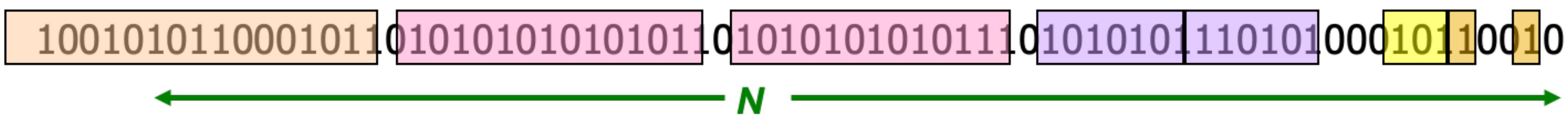
- **Algorithm (a.k.a. Reservoir Sampling)**
  - Store all the first **s** elements of the stream to **S**
  - Suppose we have seen **n-1** elements, and now the **n**$^{\text{th}}$ element arrives (**n > s**)
    - With probability **s/n**, keep the **n**$^{\text{th}}$ element, else discard it
    - If we picked the **n**$^{\text{th}}$ element, then it replaces one of the **s** elements in the sample **S**, picked uniformly at random
- Claim: This algorithm maintains a sample **S** with the desired property:
  - After **n** elements, the sample contains each element seen so far with probability **s/n**

Review—Sliding Windows

- A useful model of stream processing is that queries are about a window of length **N** – the **N** most recent elements received
- Interesting case: **N** is so large that the data cannot be stored in memory, or even on disk
  - Or, there are so many streams that windows for all cannot be stored
- Example:
  - For every product **X** we keep 0/1 stream of whether that product was sold in **each** transaction
  - We want to answer queries: how many times have we sold **X** in the last **k** sales (cannot know **k** in advance)

Review—DGIM: Buckets

- A bucket in the DGIM method is a record consisting of:
    A. The timestamp of its end expressed in **[O(logN) bits]**
    B. The number of 1s between its beginning and end expressed in **[O(logN - 1) = O(logN)bits]**
- Constraint on buckets: Number of **1**s must be a power of **2**



- How do we filter data streams?

Filtering Data Streams

- Each element of data stream is a tuple
- Given a list of keys **S**
- Determine which tuples of stream are in **S**
- Obvious solution: Hash table
    - But suppose we do not have enough memory to store all of **S** in a hash table
        - E.g., we might be processing millions of filters on the same stream

Applications

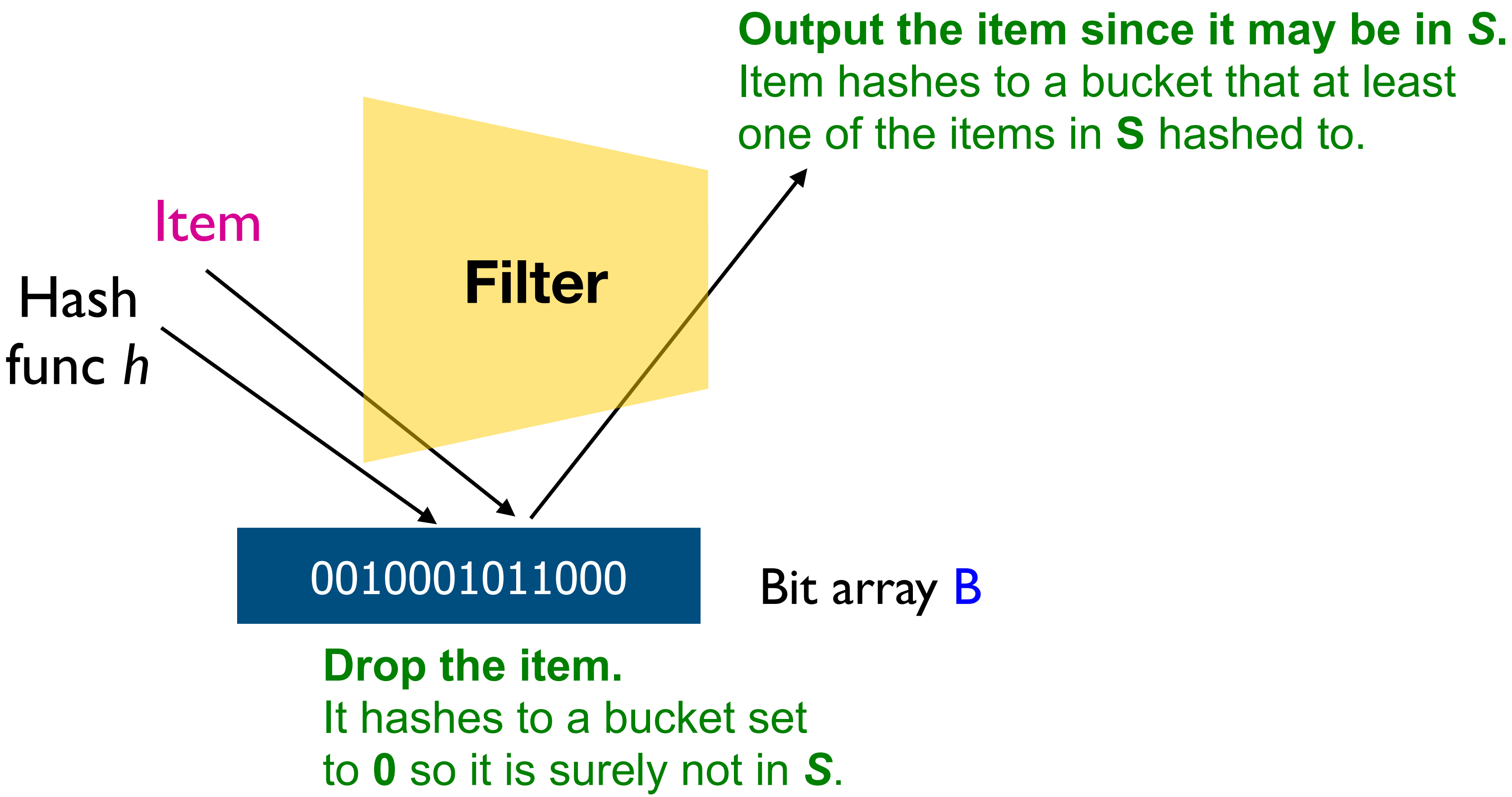
- Example: **Email spam filtering**
  - We know 1 billion “good” email addresses
  - If an email comes from one of these, it is **NOT** spam

- Example: **Publish-subscribe systems**
  - You are collecting lots of messages (news articles)
  - People express interest in certain sets of keywords
  - Determine whether each message matches user’s interest

First Attempt

- Given a set of keys **S** that we want to filter
  - Create a bit array B of **n** bits, initially all **0s**
  - Choose a hash function h with range [0,n)
  - Hash each member of $s \in S$ to one of **n** buckets, and set that bit to **1**, i.e., **B[h(s)]=1**
  - Hash each element a of the stream and output only those that hash to bit that was set to **1**
  - Output a if **B[h(a)] == 1**

First Attempt



- Creates false positives but no false negatives
    - If the item is in S we surely output it, if not we may still output it

First Attempt

- **|S| = 1 billion email addresses**

  |B|= 1GB = 8 billion bits

- If the email address is in **S**, then it surely hashes to a bucket that has the bit set to 1, so it always gets through (no false negatives)

- Approximately **1/8** of the bits are set to 1, so about **1/8th** of the addresses not in **S** get through to the output (landing randomly, false positives)

  - Actually, less than **1/8th**, because more than one address might hash to the same bit

Analysis

- More accurate analysis for the number of false positives
- Consider: If we throw **m** darts into **n** equally likely targets, <u>what is the probability that a target gets at least one dart</u>?
- In our case:
    - Targets = bits or buckets
    - Darts = hash values of items

Analysis

- We have **m** darts, **n** targets
- <u>What is the probability that a target gets at least one dart?</u>

Equals **1/e** as **n** →∞

Equivalent

$$\underbrace{1 - \underbrace{(1 - 1/n)^{n}}_{\text{Probability some target X not hit by a dart}}{}^{(m/n)}}_{\text{Probability at least one dart hits target X}} \qquad \mathbf{1 - e^{-m/n}}$$

Probability some target **X** not hit by a dart

Probability at least one dart hits target **X**

Analysis

- Fraction of 1s in the array B (by random throw)= = probability of false positive = $1 - e^{-m/n}$
- Example: 10^9 darts, $8 \cdot 10^9$ targets
  - Fraction of **1s** in $\mathbf{B = 1 - e^{-1/8} = 0.1175}$
    - Compare with our earlier estimate: **1/8 = 0.125**

Bloom Filter

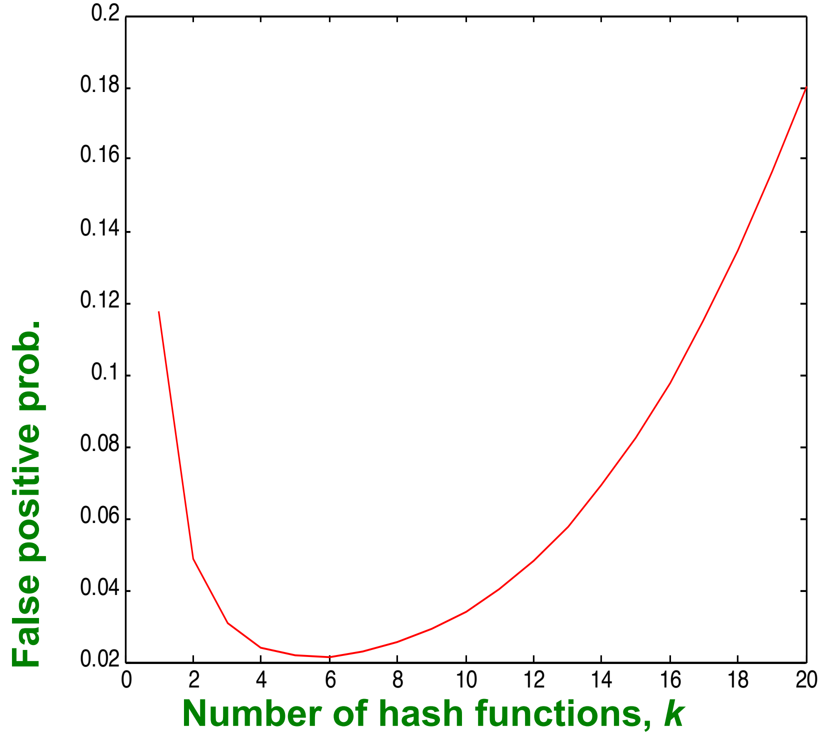
- Consider: $|S| = m$, $|B| = n$
- Use k independent hash functions $h_1, \ldots, h_k$
- Initialization:
  - Set B to all **0s**
  - Hash each element $s \in S$ using each hash function h_i, set $B[h_i(s)] = 1$ (for each $i = 1, \ldots, k$)
- Run-time:
  - When a stream element with key x arrives
    - If $B[h_i(x)] = 1$ for all $i = 1, \ldots, k$ then declare that x is in S
    - That is, x hashes to a bucket set to **1** for every hash function $h_i(x)$
  - Otherwise discard the element x

Bloom Filter — Analysis

- What fraction of the bit vector B are 1s (by random throw)?
  - Throwing $k \cdot m$ darts at n targets
  - So fraction of **1**s is $(1 - e^{-km/n})$
- But we have k independent hash functions
  and we only let the element x through **if all** k hash element x to a bucket of value **1**
- So, false positive probability $= (1 - e^{-km/n})^k$

Bloom Filter — Analysis

- m = 1 billion, n = 8 billion
  - k = 1: $(1 - e^{-1/8}) = 0.1175$
  - k = 2: $(1 - e^{-1/4})^2 = 0.0493$
- What happens as we keep increasing k?
- "Optimal" value of k: n/m ln(2)
  - In our case: Optimal k = 8 ln(2) = 5.54 ≈ 6
    - Error at k = 6: $(1 - e^{-6/8})^6 = 0.0216$



Summary

- Bloom filters guarantee no false negatives, and use limited memory
  - Great for pre-processing before more expensive checks
- Suitable for hardware implementation
  - Hash function computations can be parallelized
- Is it better to have **1** big **B** or ***k*** small **B**s?
  - It is the same: $(1 - e^{-km/n})^k$ vs. $(1 - e^{-m/(n/k)})^k$
  - But keeping 1 big B is simpler

Problems on Data Streams

Types of queries one wants to answer on a data stream:

- **Sampling data from a stream**
  - Construct a random sample
- **Queries over sliding windows**
  - Number of items of type **x** in the last **k** elements of the stream
- **Filtering a data stream**
  - Select elements with property **x** from the stream

Problems on Data Streams

Types of queries one wants to answer on a data stream:

- **Counting distinct elements**
  - Number of distinct elements in the last **k** elements of the stream
- **Estimating moments**
  - Estimate avg./std. dev. of last **k** elements
- **Finding frequent elements**

- How do we count distinctive items in a stream?

Counting Distinct Elements

- Problem:
  - Data stream consists of a universe of elements chosen from a set of size ***N***
  - Maintain a count of the number of distinct elements seen so far
- Obvious approach:
  Maintain the set of elements seen so far
  - That is, keep a hash table of all the distinct elements seen so far

Applications

- How many different words are found among the Web pages being crawled at a site?
    - Unusually low or high numbers could indicate artificial pages (spam?)
- How many different web pages does each customer request in a week?
- How many distinct products have we sold in the last week?

Using Small Storage

- Real problem: What if we do not have space to maintain the set of elements seen so far?

  - 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

Flajolet-Martin Approach

- Pick a hash function $\boldsymbol{h}$ that maps each of the $\boldsymbol{N}$ elements to at least $\log_2 N$ bits
- For each stream element $\boldsymbol{a}$, let $\boldsymbol{r(a)}$ be the number of trailing **0s** in $\boldsymbol{h(a)}$
  - $\boldsymbol{r(a)}$ = position of first 1 counting from the right
    - E.g., say $\boldsymbol{h(a) = 12}$, then ***12*** is ***1100*** in binary, so $\boldsymbol{r(a) = 2}$
- Record R = the maximum $\boldsymbol{r(a)}$ seen
  - $\mathbf{R = max_a\ r(a)}$, over all the items *a* seen so far
- Estimated number of distinct elements = $\mathbf{2}^{R}$

Intuition

- Very rough and heuristic intuition why Flajolet-Martin works:
  - h(a) hashes a with **equal prob.** to any of N values
  - Then ***h(a)*** is a sequence of **$\log_2$ N** bits, where ***2^{-r}*** fraction of all ***a***s have a tail of ***r*** zeros
    - About 50% of ***a***s hash to ***0
    - About 25% of ***a***s hash to **00
    - So, if we saw the longest tail of ***r=2*** (i.e., item hash ending ***100**) then we have probably seen **about *4*** distinct items so far
  - So, it takes to hash about *2^r* items before we see one with zero-suffix of length *r*

Formal Derivation

- Now we show why Flajolet-Martin works
- Formally, letting m be the number of distinct elements seen so far in the stream, we will show that probability of finding a tail of **r** zeros:
  - Goes to 1 if $m \gg 2^r$
  - Goes to 0 if $m \ll 2^r$
- Thus, 2^R will almost always be around m

Formal Derivation

- The probability that a given *h*(*a*) ends in at least *r* zeros is 2^{-r}
  - h(a) hashes elements uniformly at random
  - Probability that a random number ends in at least *r* zeros is 2^{-r}
- Then, the probability of **NOT** seeing a tail of length **r** among **m** elements:

$$\boxed{\left(\boxed{1-2^{-r}}\right)^{m}}$$

Prob. all end in fewer than ***r*** zeros.

Prob. that given **h(a)** ends in fewer than ***r*** zeros

Formal Derivation

- Note: $(1-2^{-r})^m = (1-2^{-r})^{2^r(m2^{-r})} \approx e^{-m2^{-r}}$
- Prob. of NOT finding a tail of length r is:
  - If ***m << 2^r^***, then prob. tends to 1
    - $(1-2^{-r})^m \approx e^{-m2^{-r}} = 1$ as **$m/2^r \to 0$**
    - So, the probability of finding a tail of length r tends to 0
  - If ***m >> 2^r^***, then prob. tends to 0
    - $(1-2^{-r})^m \approx e^{-m2^{-r}} = 0$ as **$m/2^r \to \infty$**
    - So, the probability of finding a tail of length r tends to 1
- Thus, 2^R will almost always be around m

Fixups

- $E[2^R]$ is actually infinite
  - Probability of seeking an item with trailing 0s halves when $R \rightarrow R+1$, but estimated number of distinct items doubles
- Workaround involves using many hash functions h_i and getting many samples of R_i
- How are samples R_i combined?
  - Average? What if one very large value 2^{R_i}?
  - Median? All estimates are a power of **2**
- Solution:
  - Partition your samples into small groups
  - Take the average of groups
  - Then take the median of the averages

- How do we calculate statistics for the last k items in a stream?

Generalization: Moments

- Suppose a stream has elements chosen from a set **A** of **N** values
- Let $\mathbf{m_i}$ be the number of times value **i** occurs in the stream
- The k^{th} moment is

$$\sum_{i \in A} (m_i)^k$$

Cases

$$\sum_{i \in A} (m_i)^k$$

- 0thmoment = number of distinct elements
  - The problem just considered
- 1st moment = count of the numbers of elements = length of the stream
  - Easy to compute
- 2nd moment = **surprise number S** = a measure of how uneven the distribution is

Example: Surprise Number

- Stream of length 100
- 11 distinct values
- Item counts: 10, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9
  - **Surprise S** = 910
- Item counts: 90, 1, 1, 1, 1, 1, 1, 1 ,1, 1, 1
  - **Surprise S** = 8110

AMS Method

- AMS method works for all moments
- Gives an unbiased estimate
- We will just concentrate on the **2nd moment S**
- We pick and keep track of many variables **X**:
  - For each variable **X** we store **X.el** and **X.val**
    - **X.el** corresponds to the item **i**
    - **X.val** corresponds to the **count** of item **i**
  - Note this requires a count in main memory, so number of **X**s is limited
- Our goal is to compute $$S = \sum_i m_i^2$$

One Random Variable (X)

- How to set **X.val** and **X.el**?
  - Assume stream has length **n** (we relax this later)
  - Pick some random time **t (t<n)** to start, so that any time is equally likely, and we pick k of it
  - Let at time **t** the stream have item **i. We set X.el = i**
  - Then we maintain count **c(i)** (**X.val = c**) of the number of item is in the stream starting from the chosen time **t**
- Then the estimate of the **2nd moment** $S = \sum_i m_i^2$ is:

$$f(X_i) = n(2c(i) - 1), \qquad S = \frac{1}{k}\sum_i^k f(X_i)$$

An Example

- Suppose the stream is a, b, c, b, d, a, c, d, a, b, d, c, a, a, b. The length of the stream is $n = 15$
- Since a appears 5 times, b appears 4 times, c appears 3 times, and d appears 3 times, the second moment for the stream is $5^2 + 4^2 + 3^2 + 3^2 = 59$
- We keep three variables X1, X2, and X3 and pick at random the 3rd, 8th, and 13th positions to define them

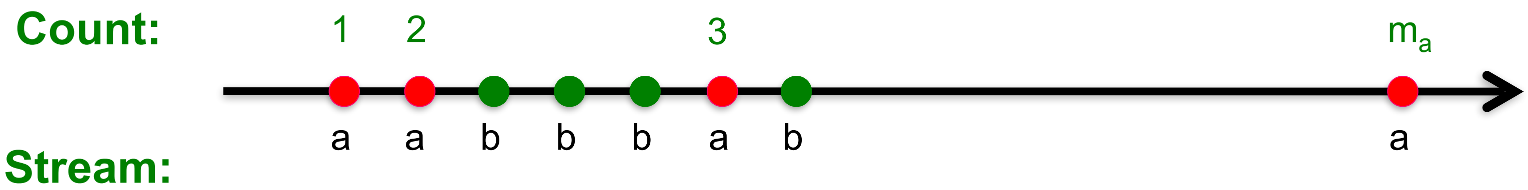
X1.element = c, X1.value = 3 From X1, we derive n(2X1.value - 1) = 15 x(2 x 3 -1) = 75

X2.element = d, X2.value = 2 From X2, we derive n(2X2.value - 1) = 15 x(2 x 2 -1) = 45

X3.element = a, X3.value = 2 From X3, we derive n(2X3.value - 1) = 15 x(2 x 2 -1) = 45

- The average of three estimates is 55, which is close to 59

Expectation Analysis



m_i … total count of item ***i*** in the stream (we are assuming stream has length **n**)

- 2nd moment is $S = \sum_i m_i^2$
- **c_t** … number of times item at time **t** appears from time **t** onwards (**c_1=m_a , c_2=m_a-1, c_3=m_b**)
- The expected value of n(2c(i) - 1) is the average over all positions i between 1 and n of n(2c(i) - 1), that is
- $$E[f(X)] = \frac{1}{n}\sum_{i=1}^{n} n(2c(i)-1) = \frac{1}{n}\sum_a n\,(1+3+5+\ldots+2m_a-1)$$

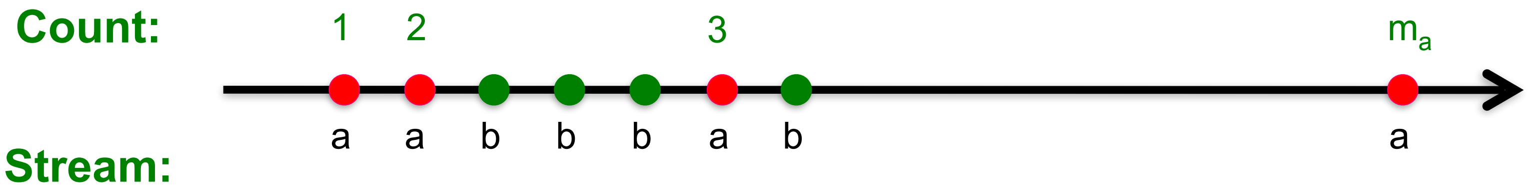
Group times by the value seen

Time t when the last X is seen (c(i)=*1*)

Time *t* when the penultimate X is seen (*c(i)=2*)

Time *t* when the first X is seen (*c(i)=m_a*)

Expectation Analysis



- $$E\left[f(X)\right] = \frac{1}{n}\sum_i n\,(1 + 3 + 5 + \ldots + 2m_i - 1)$$
  - $$\left(1 + 3 + 5 + \ldots + 2m_i - 1\right) = \sum_{i=1}^{m_i}(2i - 1) = 2\frac{m_i\left(m_i + 1\right)}{2} - m_i = (m_i)^2$$
- Then $$\boldsymbol{E\left[f(X)\right] = \frac{1}{n}\sum_i n\,\left(m_i\right)^2}$$
- So, $$\mathbf{E\left[f(X)\right]} = \sum_i \left(m_i\right)^2 = S$$
- We have the second moment (in expectation)!

Higher-Order Moments

- For estimating **k**th moment we essentially use the same algorithm but change the estimate:
  - For **k=2** we used ***n* (2·c - 1)**
  - For **k=3** we use: ***n* (3·c^2 - 3c + 1)**   (where **c=X.val**)
- Why?
  - For **k=2**: Remember we had (1 + 3 + 5 + … + 2m_a - 1) and we showed terms **2c-1** (for **c=1,…,m**) sum to m^2
    - $\sum_{c=1}^{m} 2c - 1 = \sum_{c=1}^{m} c^2 - \sum_{c=1}^{m} (c-1)^2 = m^2$
    - So: $\mathbf{2c - 1 = c^2 - (c-1)^2}$
- For **k=3**: c^3 - (c-1)^3 = 3c^2 - 3c + 1
- Generally: Estimate $= n\,(c^k - (c-1)^k)$

Combining Samples

- In practice:
  - Compute $f(X) = n(2c - 1)$ for as many variables **X** as you can fit in memory
  - Average them in groups
  - Take median of averages
- Problem: Streams never end
  - We assumed there was a number **n**, the number of positions in the stream
  - But real streams go on forever, so **n** is a variable – the number of inputs seen so far

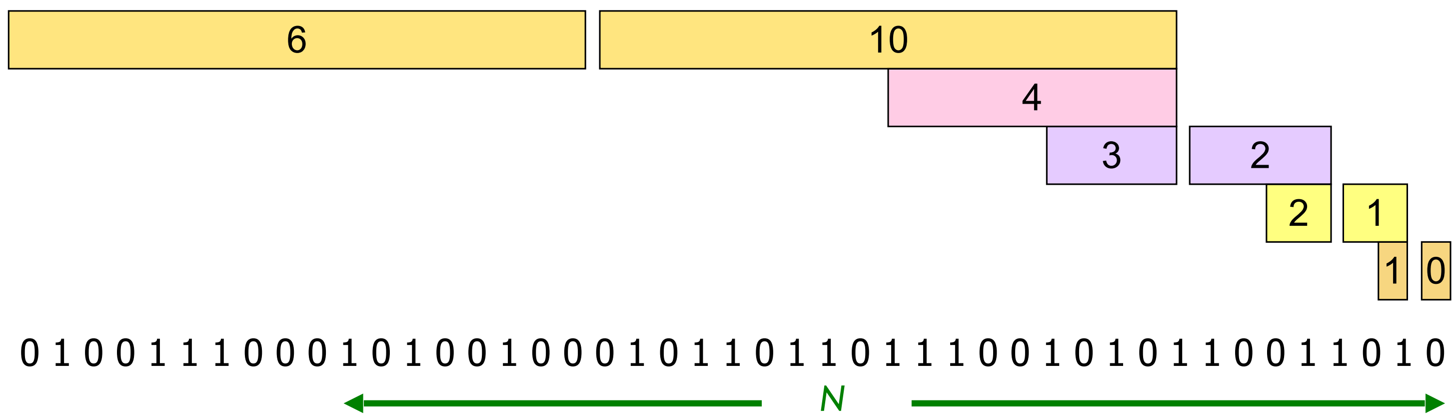
Streams Never End: Fixups

- (1) The variables **X** have **n** as a factor – keep **n** separately; just hold the count in **X**
- (2) Suppose we can only store **k** counts. We must throw some **X**s out as time goes on:
  - Objective: Each starting time **t** is selected with probability **k/n**
  - Solution: (fixed-size sampling!)
    - Choose the first **k** times for **k** variables
    - When the n^{th} element arrives **(n > k)**, choose it with probability **k/n**
    - If you choose it, throw one of the previously stored variables **X** out, with equal probability

- How do we count frequently-appearing items in a stream?

Counting Itemsets

- New Problem: Given a stream, which items appear more than **s** times in the window?
- Possible solution: Tear the stream up into multiple binary streams; one binary stream per item
  - 1 = item present; 0 = not present
  - Use **DGIM** to estimate counts of 1s for all items



Extensions

- In principle, you could count frequent pairs or even larger sets the same way
    - One stream per itemset
- Drawbacks:
    - Only approximate
    - **Number of itemsets** is way too big

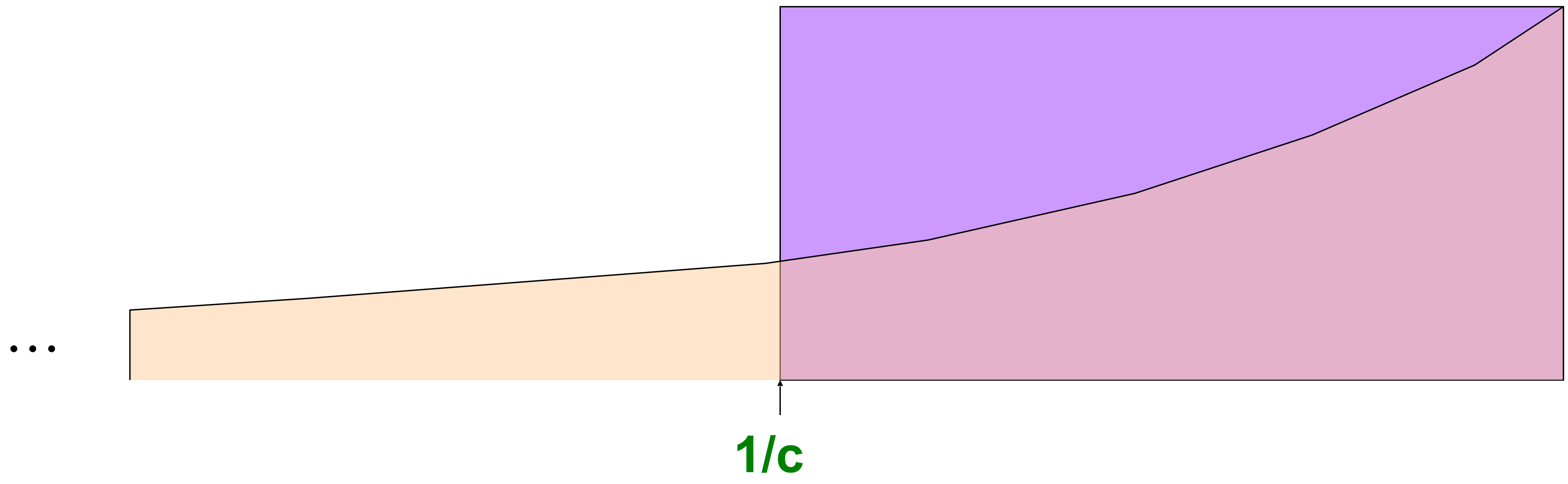
Exponentially Decaying Windows

- Exponentially decaying windows: A heuristic for selecting likely frequent item(sets)
  - What are "currently" most popular movies?
    - Instead of computing the raw count in last **N** elements
    - Compute a smooth aggregation over the whole stream
- If stream is $a_1, a_2, \ldots$ and we are taking the sum of the stream, take the answer at time **t** to be: $= \sum_{i=1}^{t} a_i (1-c)^{t-i}$
  - **c** is a constant, presumably tiny, like 10^{-6} or 10^{-9}
- When **new** $\mathbf{a_{t+1}}$ arrives:
  Multiply current sum by **(1-c)** and add $\mathbf{a_{t+1}}$

Example: Counting Items

- If each $\boldsymbol{a_i}$ is an "item" we can compute the characteristic function of each possible item $\boldsymbol{x}$ as an Exponentially Decaying Window
  - That is: $\sum_{i=1}^{t} \delta_i \cdot (1-c)^{t-i}$ where $\boldsymbol{\delta_i=1}$ if $\boldsymbol{a_i=x}$, and **0** otherwise
  - Imagine that for each item $\boldsymbol{x}$ we have a binary stream (**1** if $\boldsymbol{x}$ appears, **0** if $\boldsymbol{x}$ does not appear)
  - New item $\boldsymbol{x}$ arrives:
    - Multiply all counts by **(1-c)**
    - Add **+1** to count for element $\boldsymbol{x}$
- Call this sum the "weight" of item $\boldsymbol{x}$

Sliding Versus Decaying Windows



- Important property: Sum over all weights $\sum_t (1-c)^t$ is

1/[1 – (1 – c)] = 1/c

$$\sum_{k=0}^{n} z^k = \frac{1 - z^{n+1}}{1 - z}$$

Example: Counting Items

- What are “currently” most popular movies?
- Suppose we want to find movies of weight **> ½**
    - Important property: Sum over all weights $\sum_t (1-c)^t$ is **1/[1 – (1 – c)] = 1/c**
- Thus:
    - There cannot be more than **2/c** movies with weight of ½ or more
- So, **2/c** is a limit on the **number of movies** being counted at any time

Reading

- Jure Leskovec, Anand Raj, Jeff Ullman, “Mining of Massive Datasets,” Cambridge University Press, Chapter 4