EE359 Big Data Mining

Streaming Algorithms

Jiaxin Ding

John Hopcroft Center

上海交通大学
约翰·霍普克罗夫特计算机科学中心
John Hopcroft Center for Computer Science

Data Streams

- A data stream is a sequence of signals used to transmit or receive information that is in the process of being transmitted. In many situations, we do not know the entire data set in advance.
  - **Infinite**
  - **Non-stationary**

. . . 1, 5, 2, 7, 0, 9, 3

. . . a, r, v, t, y, h, b

. . . 0, 0, 1, 0, 1, 1, 0

time

Adapted from J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

The Stream Model

- Input **elements** enter at a **rapid** rate, at one or more input ports
  - We call elements of the stream **tuples**
- The system cannot store the entire stream
- Q: How do you make critical calculations about the stream using a limited amount of memory?

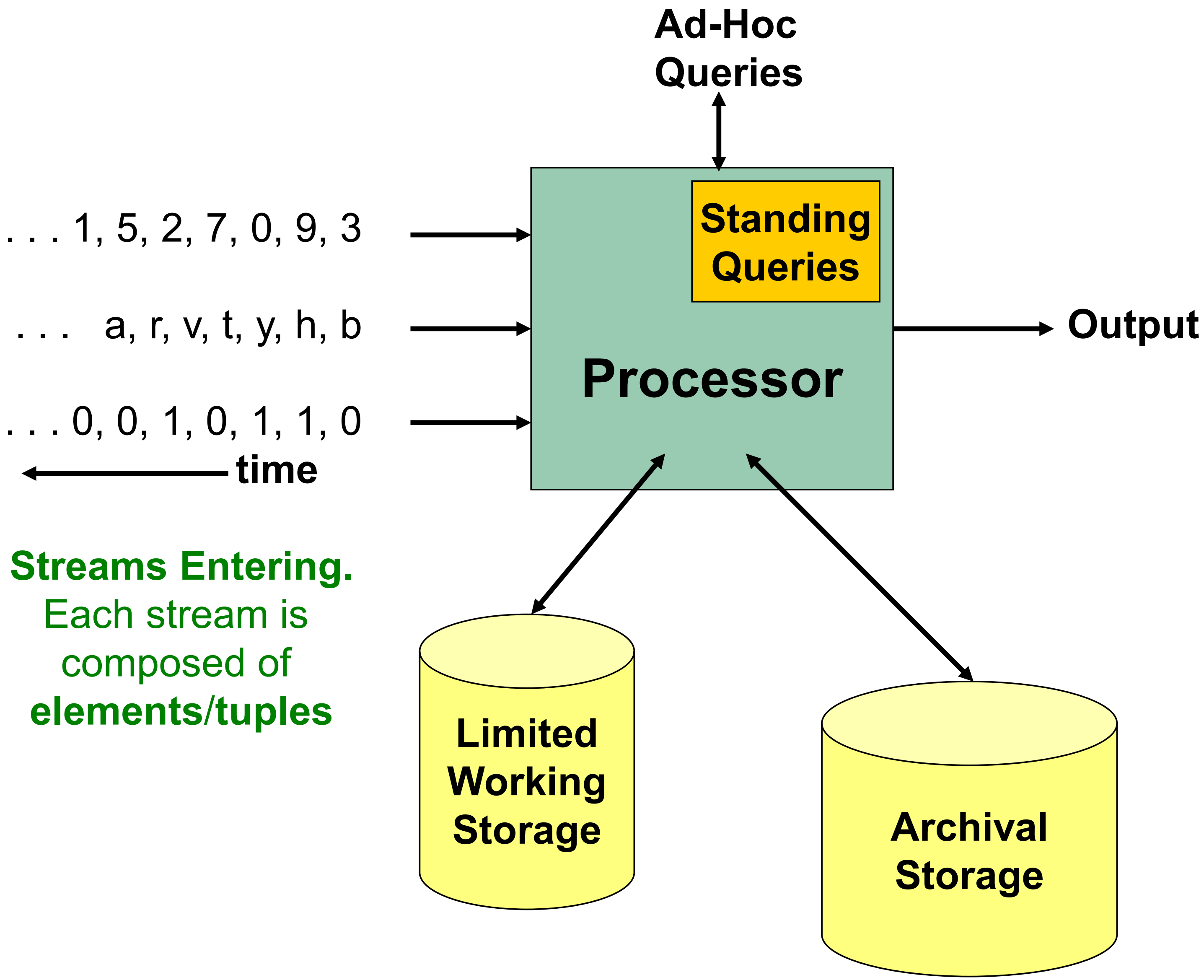
. . . 1, 5, 2, 7, 0, 9, 3

. . . a, r, v, t, y, h, b

. . . 0, 0, 1, 0, 1, 1, 0

time

General Stream Processing Model



It is better to use a crude approximation and know the truth, plus or minus 10 percent, than demand an exact solution and know nothing at all.

——Arthur Bloch, The Complete Murphy's Law

Applications: Networks

- **Mining network streams**
  - Finding abnormal patterns in sensor reading streams
  - Filtering out spam calls in phone call streams
  - Detect denial-of-service attacks in IP packet streams

Applications: Internet

- **Mining query streams**
  - Google wants to know what **queries** are more **frequent** today than yesterday
- **Mining click streams**
  - Bytedance 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 Weibo

Problems on Data Streams

- Types of queries one wants on answer on a data stream (element):
  - **Sampling data from a stream**
    - Construct a random sample
  - **Filtering a data stream**
    - Select elements with property x from the stream

Problems on Data Streams

- Types of queries one wants on answer on a data stream (statistics):
  - **Queries over sliding windows**
    - Number of items of type x in the last k elements of the stream
  - **Counting distinct elements**
    - Number of distinct elements in the last k elements of the stream
  - **Finding frequent elements**
    - Estimate the most frequent elements of the last k elements
  - **Estimating moments**
    - Estimate avg./std. dev. of last k elements

Sampling from a Data Stream: Sampling a fixed-size sample

Maintaining a fixed-size sample

- Suppose we need to maintain a random sample S of size exactly s tuples
  - E.g., main memory size constraint

- Suppose at time n we have seen n items
  - Each item is in the sample S with equal prob. 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.

Solution: Fixed Size Sample

- **Algorithm**

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

- 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: By Induction

- **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: By Induction

- **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:

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

- 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* = $\boldsymbol{\frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1}}$

Filtering Data Streams

Applications

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

- Publish-subscribe systems
  - You are collecting lots of messages
  - People express interest in certain sets of keywords
  - Determine whether each message matches user’s interest

Filtering Data Streams

- Each element of data stream is a tuple
- Given a list of keys $S = [\boldsymbol{key_1}, \boldsymbol{key_2}, \cdots]$
- **Determine which tuples of stream are in *S***

- Obvious solution: store and compare
  - But suppose we **do not have enough memory** to store all of *S*
  - The **complexity** is *O(S)*, which can be big.

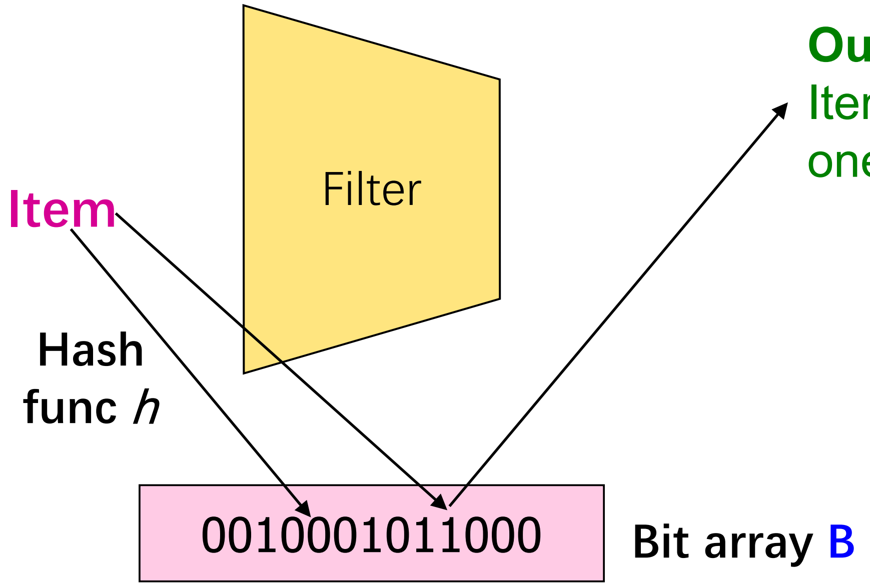
Adapted from J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

First Cut Solution

- **Given a set of keys *S* that we want to filter**
- Create a **bit array** *B* of *n* bits, initially all *0*s
- 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 Cut Solution

Output the item since it may be in *S*.
Item hashes to a bucket that at least one of the items in **S** hashed to.



Drop the item.
It hashes to a bucket set to **0** so it is surely not in ***S***.

- 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 Cut Solution

- $|S|$ = 1 billion email addresses
  $|B|$= 1GB = 8 billion bits, for the hash values
- If the email address is in ***S***, then it surely hashes to a bucket that has the big set to **1**, so it always gets through (*no false negatives*)
- Approximately **1/8** of the bits are set to **1**, so about **1/8** of the addresses not in ***S*** get through to the output (*false positives*)

Analysis: Throwing Darts

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

Analysis: Throwing Darts

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

Equals **1/e** as **n** $\rightarrow\infty$

Equivalent

$$\boxed{1 - \underline{\boxed{(1 - 1/n)}^{\,n(m/n)}}}$$

$$\mathbf{1 - e^{-m/n}}$$

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

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

Analysis: Throwing Darts

- Fraction of 1s in the array B

= probability of false positive = $1 – e^{-m/n}$

- **Example:** $\mathbf{10^9}$ darts, $\mathbf{8 * 10^9}$ targets
  - Fraction of **1s** in **B = $\mathbf{1 – e^{-1/8}}$ = 0.1175**
    - Compare with our earlier estimate: **1/8 = 0.125**

- How to further **improve** this false positive probability?
- Similar to LSH: Bloom Filter.

Bloom Filter

- Consider: **|S| =** m**, |B| =** n
- Use k independent hash functions $h_1, \cdots, h_k$
- **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 $i = 1, \dots, k$)
- **Run-time:**
  - When a stream element with key x arrives
    - If **B[**$h_i(x)$**] = 1** **for all** i **= 1,…,** 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

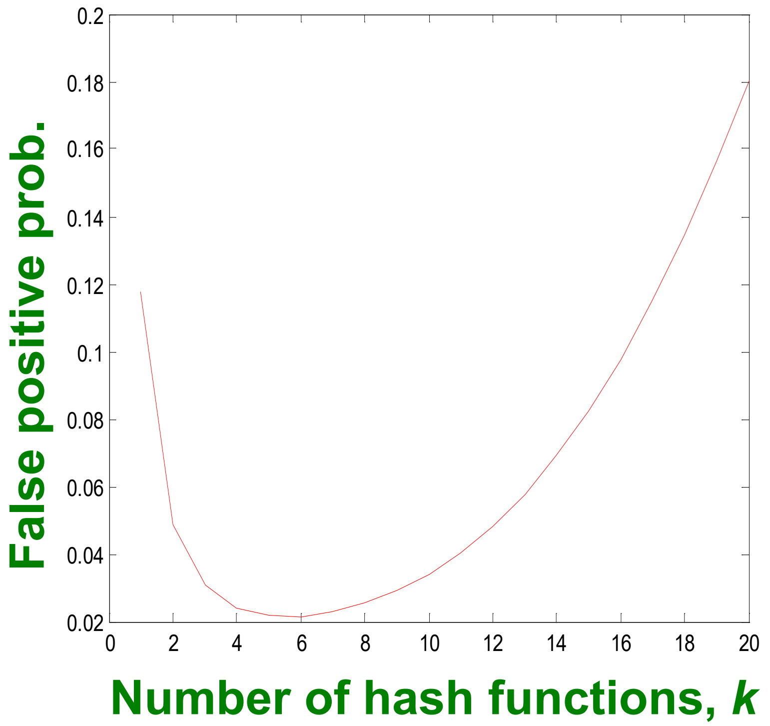
- **What fraction of the bit vector B are 1s?**
  - Throwing $k \cdot m$ darts at n targets
  - So fraction of **1**s is $(1 - e^{-km/n})$ **(false positive of 1 hash function)**

- 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**: $(1 - e^{-1/6})^2$ = **0.0235**

Bloom Filter: Wrap-up

- 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

- Disadvantage: only insertion, no deletion from Bloom Filter.

Count-Min Sketch

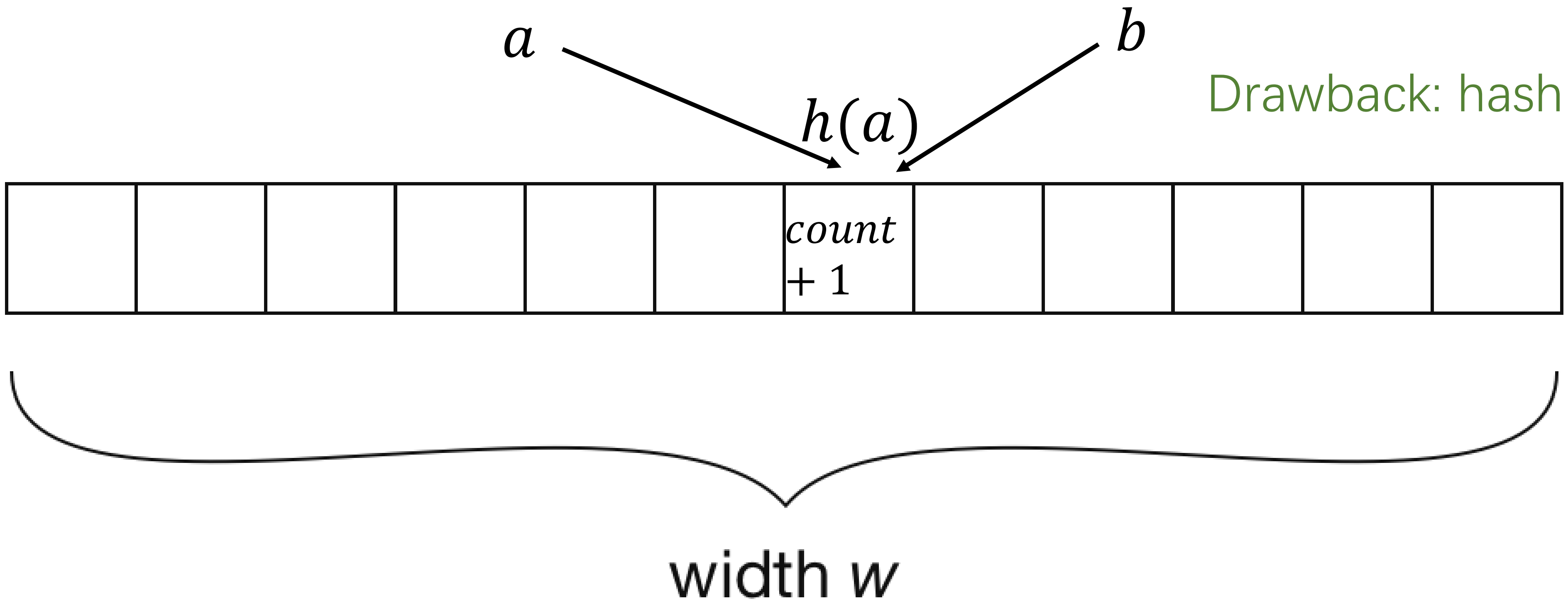
Count Element Frequency

- Faced with big data streams, storing all elements and corresponding frequencies is **impossible**.

- **Approximate** counts are acceptable.
- We can use **hashing** again.

Approximate Counts with Hashing

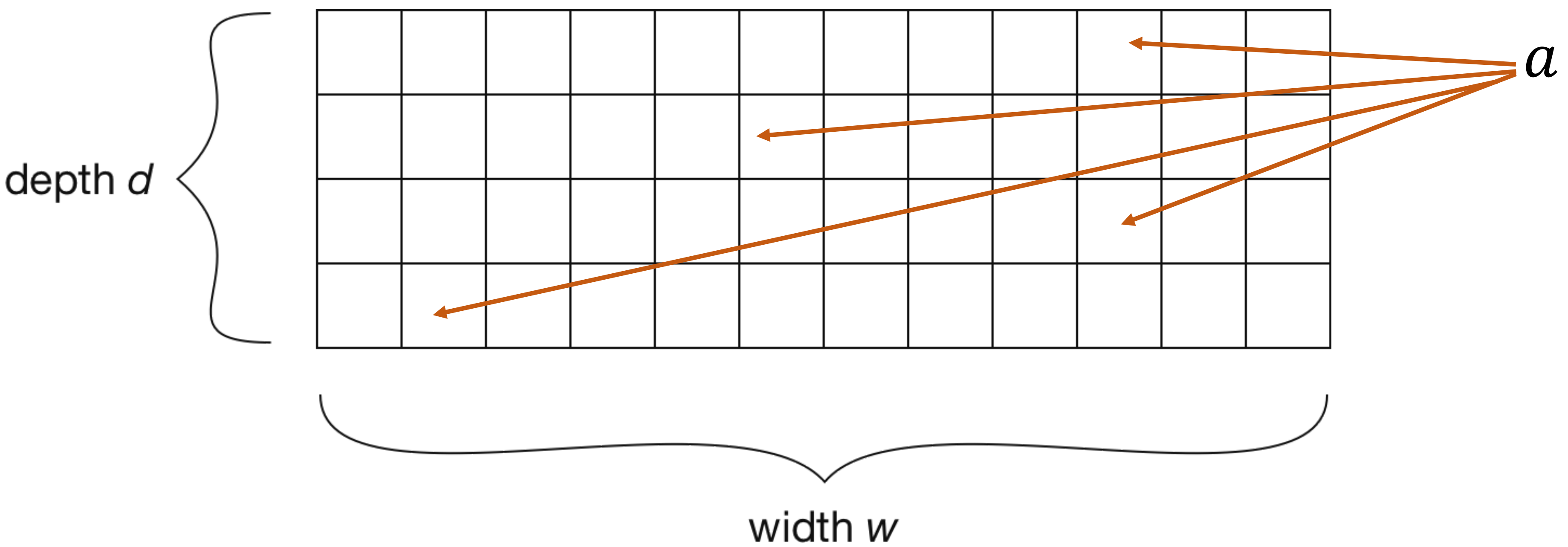
- **Initialization**: $count[i] = 0$, for $i \in [1, w]$
- **Increment** count of element a: $count[h(a)] += 1$
- **Retrieve** count of element a: $count[h(a)]$



Drawback: hash conflicts

Improvement: More Hash Functions

- We use *d* pairwise independent hash functions
- **Increment** count of element a: $count[i, h_i(a)] += 1$ for $i \in [1, d]$
- **Retrieve** count of element a: $\min_{i \in [1,d]} count[i, h_i(a)]$



Guarantees

- Theorem[1]: with probability $1-\delta$, the error is at most $\varepsilon * count$. Concrete values for these error bounds can be chosen by setting $w = \left\lceil \frac{e}{\varepsilon} \right\rceil$ and $\mathrm{d} = \left\lceil \ln(\frac{1}{\delta}) \right\rceil$, e ≈2.718.
  - Adding another **hash** function **exponentially** decreases the chance of hash conflicts
  - Increasing the **width** helps spread up the counts with a **linear** effect

[1]Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. Journal of Algorithms 55, 1 (2005), 58–75

Queries over a Sliding Window

Sliding Windows

- A useful model of stream processing is that queries are within a *window* of length N – the N most recent elements received
  - **Amazon example:** For every product **X** we keep 0/1 stream of whether that product was sold in the **n**-th transaction.

    We want answer queries, how many times we sold **X** in the last **k** sales.

Item sold
in transactions:

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

← Past   Future →

Suppose we keep a window with length N=6,
we can query on the last k transactions, for $k \leq N$.

Counting Bits over a Sliding Window

- **Problem:**
  - Given a stream of **0**s and **1**s
  - **How many 1s are in the last *k* bits?** where $k \leq N$

- **Obvious solution:** Store the most recent N bits
  - When new bit comes in, discard the N**+1**st bit
  - **Not feasible** when N is so **large** that the data cannot be stored in memory and cannot answer in short time

- **Approximation solution?**
  - **Without any assumptions on the data distribution**

0 1 0 0 1 1 0 1 1 1 0 1 0 1 0 1 1 0 $\boxed{1\ 1\ 0\ 1\ 1\ 0}$

← Past   Future →

DGIM Method

DGIM(Datar-Gionis-Indyk-Motwani**)** Algorithm

- Does **not** have assumptions on data distribution

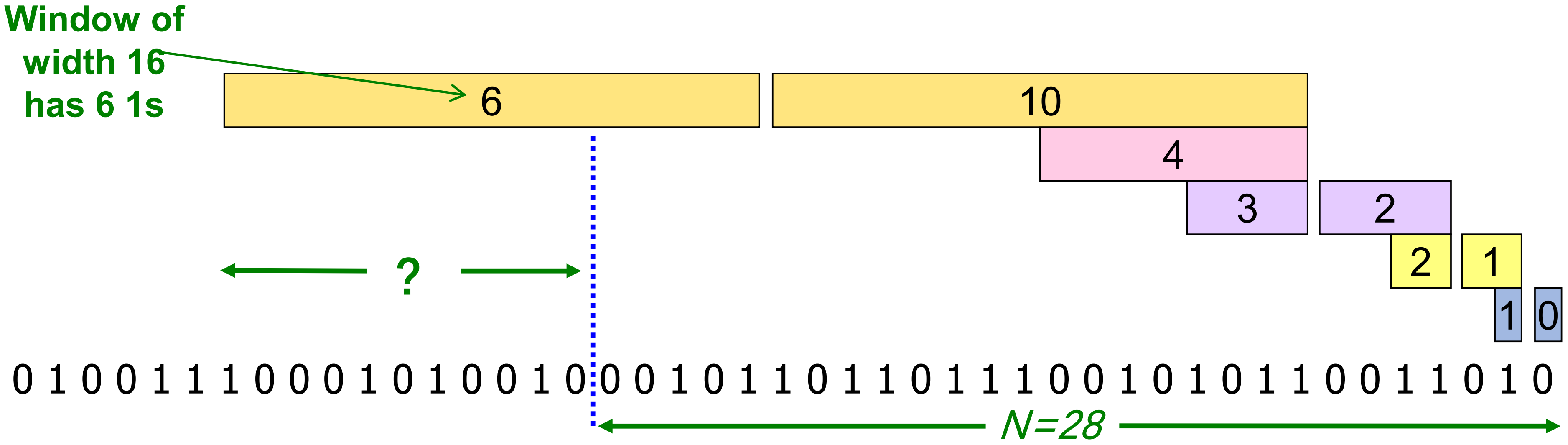
- Only stores $\boldsymbol{O(log^2 N)}$ bits per stream

- Solution gives **approximate** answer, **never off** by more than **50%**
  - Error factor can be reduced to any fraction > 0, with more complicated algorithm and proportionally more stored bits

Idea: Exponential Windows

1. when a bit comes in, create a bucket of length 1 with the proper count (0 or 1).
2. If any level has 3 buckets:
   a) add the rightmost two and create a bucket at the next higher level (twice the length) with that sum.
   b) delete the leftmost two buckets, keeping only the rightmost of the three.
3. Repeat (2) recursively for progressively higher levels.

- **First trial:**
  - Summarize **exponentially increasing** regions of the stream, looking backward, to answer queries over last k items $(k \leq N)$.
  - Drop small regions if there are more than two on the same level(keep the leftmost)



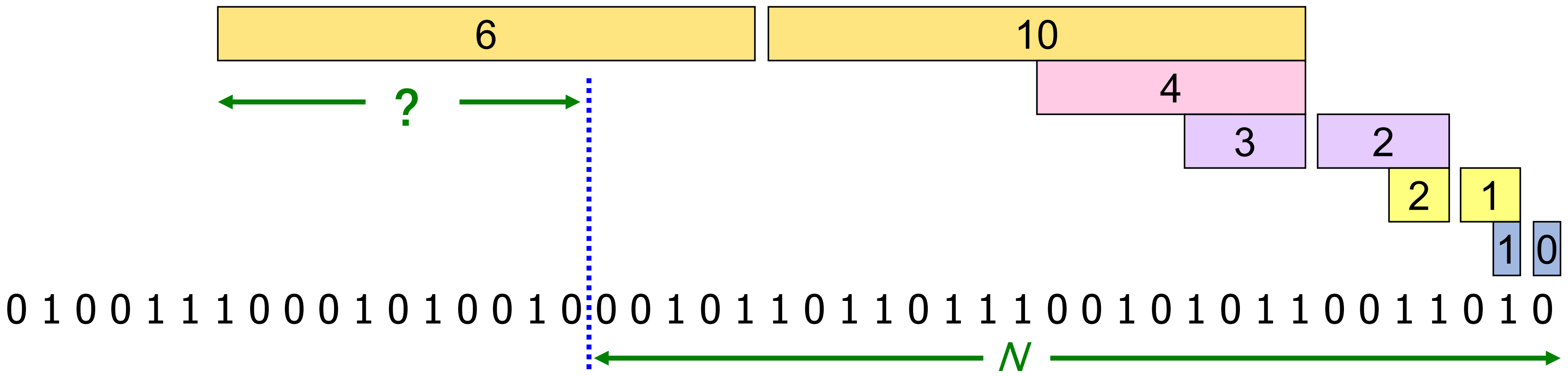
We can reconstruct the count of the last ***N*** bits, except we are not sure how many of the last **6 1s** are included in the ***N***

What's Good?

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

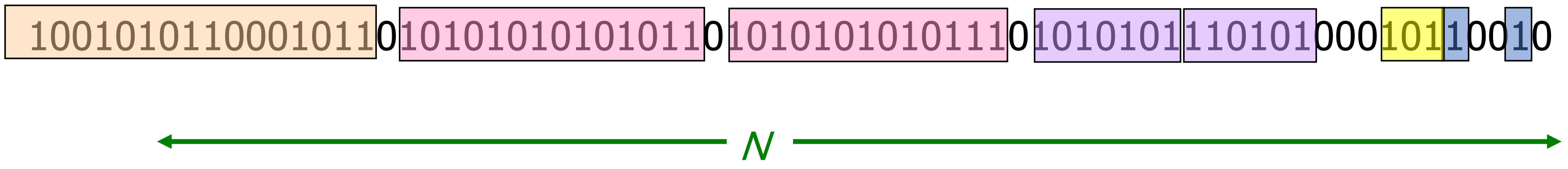
What's Not So Good?

- **The relative error could be unbounded!**
  - **Relative error=error/true count**
  - Consider the case that all the **1s** are in the unknown area(**?** part) and the rest are all 0s. Here the relative error is infinite.



Fixup: DGIM method

- **Idea:** Instead of summarizing fixed-length blocks, summarize blocks with specific number of **1s**:
  - Let the block **sizes** (number of **1s**) increase exponentially
  - Data dependent
- When there are few 1s 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 $\boldsymbol{O(log_2 N)}$ bits

E.g. Real Timestamp： 105,
N=40, we only store 105 mod 40=25.

10010101100010110101010101010110101010101011101010101110101000101100 10

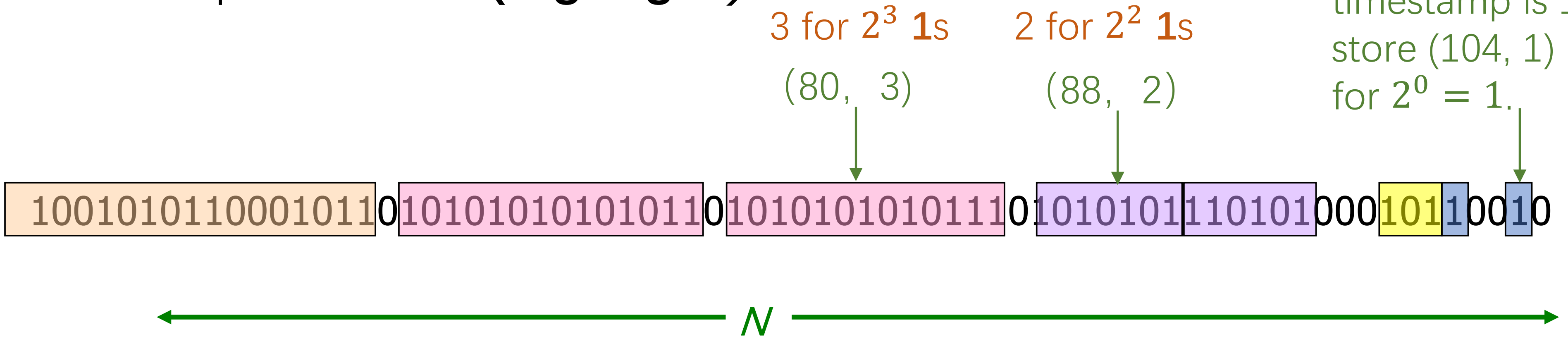
N

DGIM: Buckets

- A *bucket* in the DGIM method is a record consisting of:
  - **The timestamp of its end [O(log *N*) bits]**
  - **The number of 1s between its beginning and end [O(log log *N*) bits]**

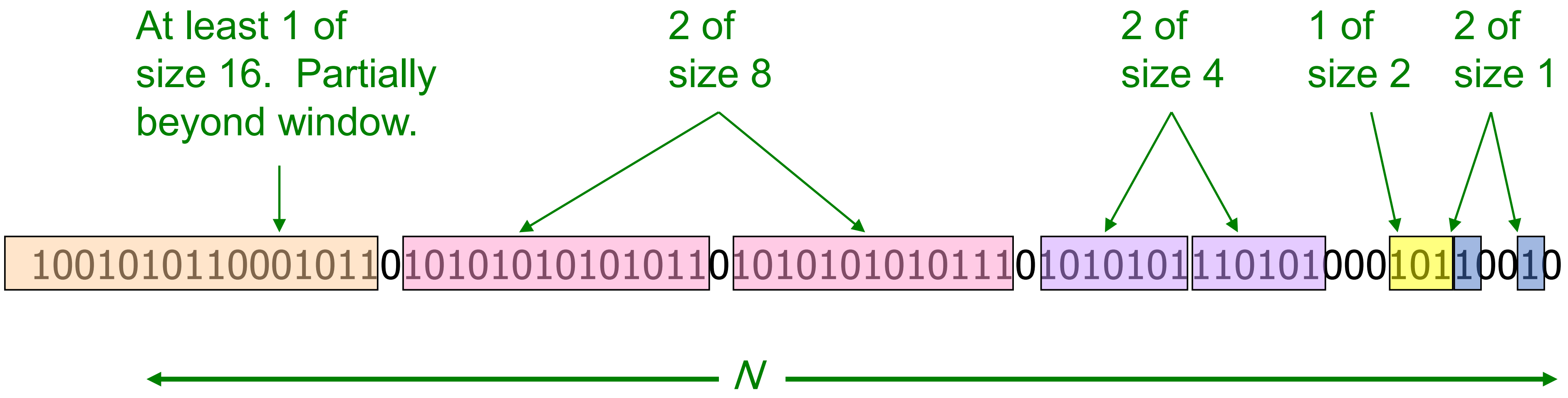
- **Constraint on buckets:**
  Number of **1s** must be a power of **2**
  - That explains the **O(log log *N*)**

E.g. In this window, if the timestamp of the last timestamp is 105, we actually store (104, 1) here, or (104, 0) for $2^0 = 1$.



Representing a Stream by Buckets

- 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**
    - Earlier buckets are not smaller than later buckets
- Buckets **disappear** when their end-time is **> *N*** time units in the past



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

Updating Buckets

- **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) Continue until there are at most two buckets of size 2^i

Example: Updating Buckets

Current state of the stream:

1001010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1 00 1 0

Bit of value 1 arrives

001010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1 00 1 0 **1**

Two smallest buckets get merged into a size-2 bucket

001010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1001 0 **1**

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

010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1001 0 1 **1** **0** **1**

Buckets get merged…

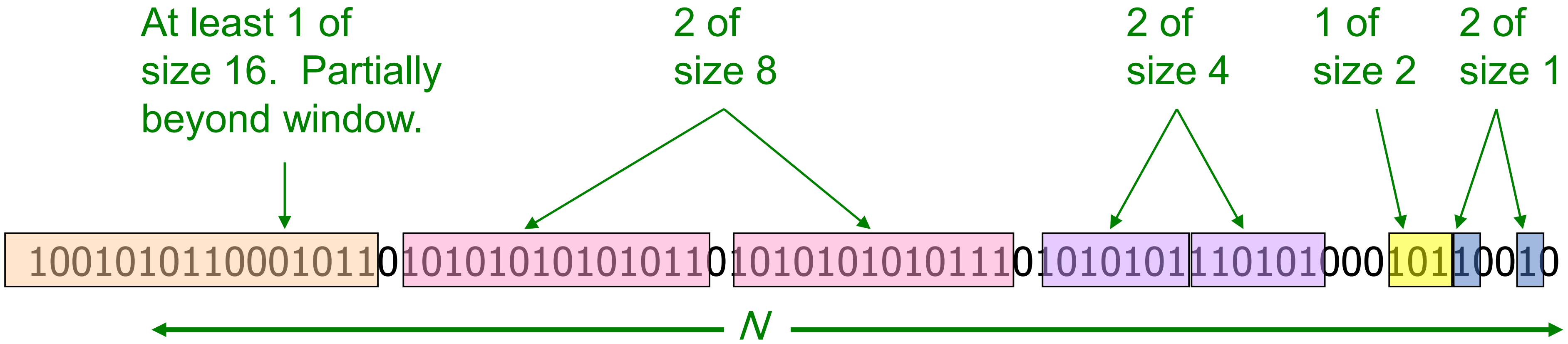
010110001011 0 10101010101011 0 1010101010111 0 1010101 110101 000 101 1001 0 1**1** **0** **1**

State of the buckets after merging

010110001011 0 1010101010101101010101010111 0 1010101110101 000 1011001 0 11 0 1

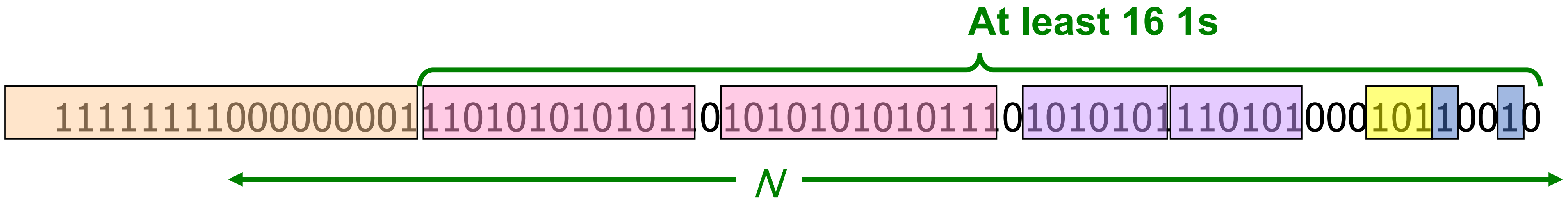
How to Query?

- To estimate the number of 1s in the most recent N bits:
  1. Sum the sizes of **all** buckets **but the last partially overlapping with the query**
  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



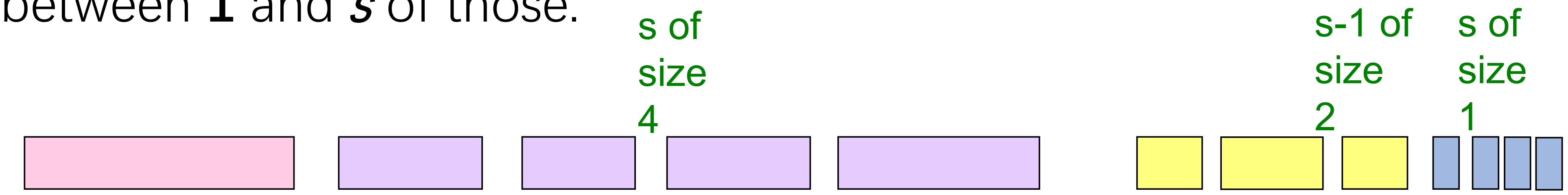
Error Bound: Proof

- **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 **1s** 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, relative error at most **50%**



Further Reducing the Error

- Instead of maintaining **1** or **2** of each size bucket, we allow either *s*-**1** or *s* buckets (*s* > **2**)
  - Except for the largest size buckets, where we can have any number between **1** and *s* of those.



- **Error is at most $\frac{2^{r-1}}{(s-1)(2^r-1)}$ =O(1/s)**
- By picking *s* appropriately, we can tradeoff between number of bits we store and the error

Extensions

- **Can we handle the case where the stream is not bits, but integers, and we want the sum of the last *k* elements?**
- **We want the sum of the last *k* elements**
  - **Amazon:** Avg. price of last **k** sales
- **Solution:**
  - **If you know all have at most *m* bits**
    - Treat *m* bits of each integer as a separate stream
    - Use DGIM to count **1s** in each integer
    - The sum is $= \sum_{i=0}^{m-1} c_i 2^i$

Two streams represent 1

11111110000000011101010101011010101010101110101010111010100010110010

11111110000000011101010101011010101000101110101010111010100011110011

Counting Distinct Elements

Counting Distinct Elements

- **We often ask questions like:**
  - How many distinct people visit the website?
  - How many distinct products have we sold in the last week?

- **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

Using Small Storage

- **Obvious approach:**
  Maintain the set of elements seen so far

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

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

Why It Works: Intuition

- **Rough and heuristic intuition:**
  - *h(a)* hashes *a* with **equal prob.** to any of *N* values
  - Then *h(a)* is a sequence of $\mathbf{\log_2 N}$ 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***

Why It Works: More formally

- Now we show why Flajolet-Martin works

- Formally, we will show that **probability of finding a tail of *r* zeros**:
  - **Goes to 1 if $\boldsymbol{m \gg 2^r}$**
  - **Goes to 0 if $\boldsymbol{m \ll 2^r}$**

  where $\boldsymbol{m}$ is the number of distinct elements seen so far in the stream

- Thus, 2^R will almost always be **around** m!

Why It Works: More formally

- **What is 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 $\mathbf{2^{-r}}$
- Then, the probability of **NOT** seeing a tail of length *r* among *m* elements:

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

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

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

Why It Works: More formally

- **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 → 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 → ∞**
    - So, the probability of finding a tail of length *r* tends to **1**

- **Thus, 2^R will almost always be around *m*!**

Issues to fix

- $E[2^R]$ is actually infinite
  - Probability halves when $R \rightarrow R+1$, but value doubles
  - Limit the bits of hashing values to L
- The estimation is biased
  - Estimated with $\mathbf{2^R/\Phi}$, where $\Phi = 0.77351$ is a correction factor.
- Problems of high variance. Improve accuracy.
  - Use many hash functions with samples of R
  - Partition your samples into small groups
  - Take the median of groups
  - Then take the average of the medians

Computing Moments

Generalization: Moments

- Suppose a stream has elements chosen from a set *A* of *N* values (say 1 to N)
- Let m_i be the number of times value *i* occurs in the stream
- The k^{th} *moment* is

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

Special Cases

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

- **0^{th}moment =** number of **distinct** elements(Flajolet-Martin Approach)
- **1^{st} moment =** count of the **numbers** of elements = length of the stream
- **2^{nd} moment = surprise number S** = a measure of how uneven the distribution is

E.g. **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 = 8,110

AMS(Alon-Matias-Szegedy) 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 some 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 Xs is limited
- **Our goal is to compute $S = \sum_i m_i^2$**

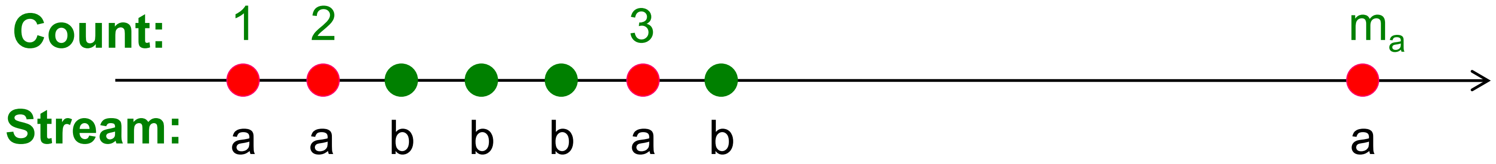
One Random Variable (X)

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

$$S = f(X) = n\,(2 \cdot c - 1)$$

  - Note, we will keep track of multiple Xs, ($X_1, X_2, \cdots X_k$) and our final estimate will be $S = 1/k \sum_j^k f(X_j)$

Expectation Analysis



- 2^{nd} moment is $S = \sum_i m_i^2$
- c_t … number of times item at time t appears from time **t** onwards ($c_1=m_a$, $c_2=m_a-1$, $c_3=m_b$)
- $\boldsymbol{E[f(X)] = \frac{1}{n}\sum_{t=1}^{n} n(2c_t - 1)}$

$$\boldsymbol{= \frac{1}{n}\sum_i n\,(1 + 3 + 5 + \cdots + 2m_i - 1)}$$

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

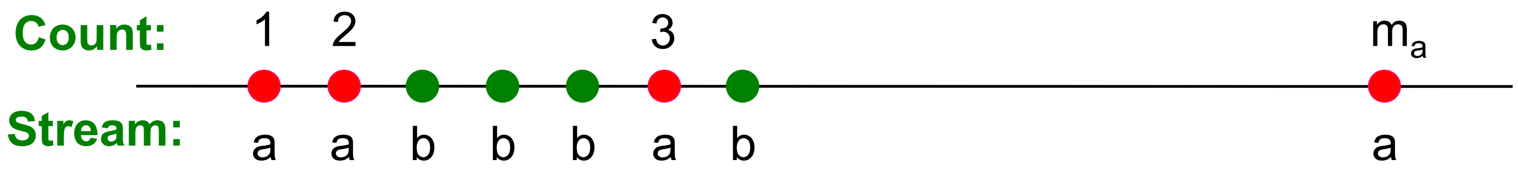
Group times by the value seen

Time t when the last i is seen **($c_t=1$)**

Time t when the penultimate i is seen ($c_t=2$)

Time t when the first i is seen ($c_t=m_i$)

Expectation Analysis



- $E[f(X)] = \frac{1}{n}\sum_i n\,(1 + 3 + 5 + \cdots + 2m_i - 1)$
  - calculation: $(1 + 3 + 5 + \cdots + 2m_i - 1) = \sum_{i=1}^{m_i}(2i - 1) = 2\frac{m_i(m_i+1)}{2} - m_i = (m_i)^2$
- **Then $\mathbf{E[f(X)]} = \frac{1}{n}\sum_i n\,(m_i)^2$**
- **So, $\mathbf{E[f(X)]} = \sum_i (m_i)^2 = S$**
- **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·c² – 3c + 1)** (where **c=X.val**)
- **Why?**
  - **For k=2:** Remember we had $(1 + 3 + 5 + \cdots + 2m_i - 1)$ and we showed terms *2c-1* (for **c=1,⋯,m**) sum to *m²*
    - $\sum_{c=1}^{m} 2c - 1 = \sum_{c=1}^{m} c^2 - \sum_{c=1}^{m} (c-1)^2 = m^2$
    - **So:** $\mathbf{2c - 1 = c^2 - (c-1)^2}$
  - **For k=3:** **c³ - (c-1)³** = **3c² - 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 Xs out as time goes on:

- **Objective:** Each starting time t is selected with probability k/n
- **Solution: (fixed-size (Reservoir) 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

Summary of Streaming Algorithms

- Queries
  - Filtering a data stream
  - Queries over a sliding window
  - Counting distinct elements
  - Estimating moments
- Key techniques
  - Hashing functions
  - Approximation with sketch/summarization
  - Theoretical analysis