Data Streams

- The Stream Data Model
- Stream Queries
- Sampling Data in a Stream
- Filtering Streams
- Counting Distinct Elements in a Stream
- Estimating Moments
- Window Queries

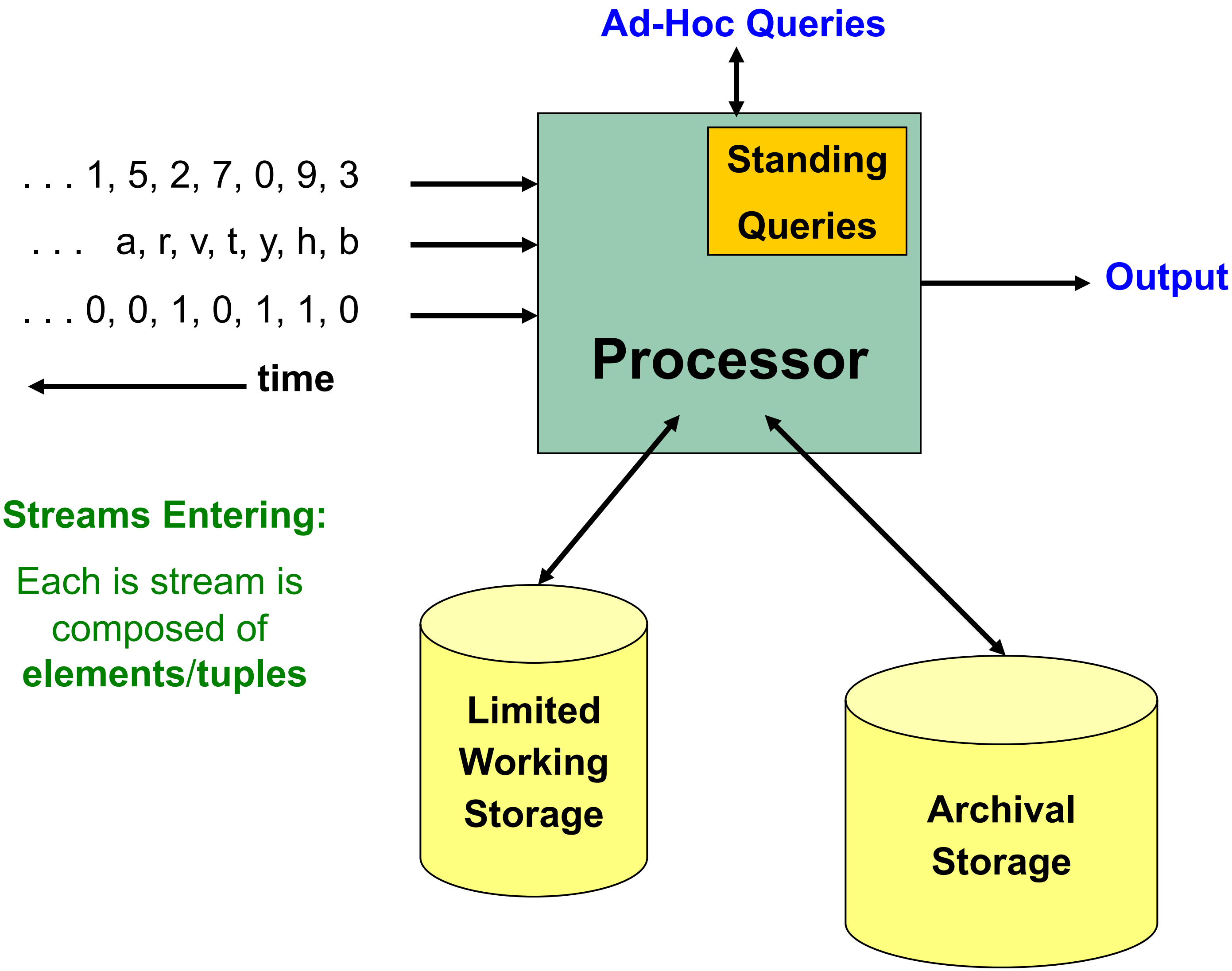
Reading: Chapter 4, Mining of Massive Datasets (Leskovec, Rajaraman, Ullman), http://www.mmds.org

The Stream Data Model

Main assumptions

- Data arrive in one or more streams
- If not processed immediately (or stored), data are lost forever
- Data arrive rapidly
  - not feasible to store data, then process them and obtain results in real time
  - data volumes may exceed storage capacity
- The input rate is controlled externally
  - Google queries
  - Twitter or Facebook status updates
- Data are infinite and non-stationary (the distribution changes over time)

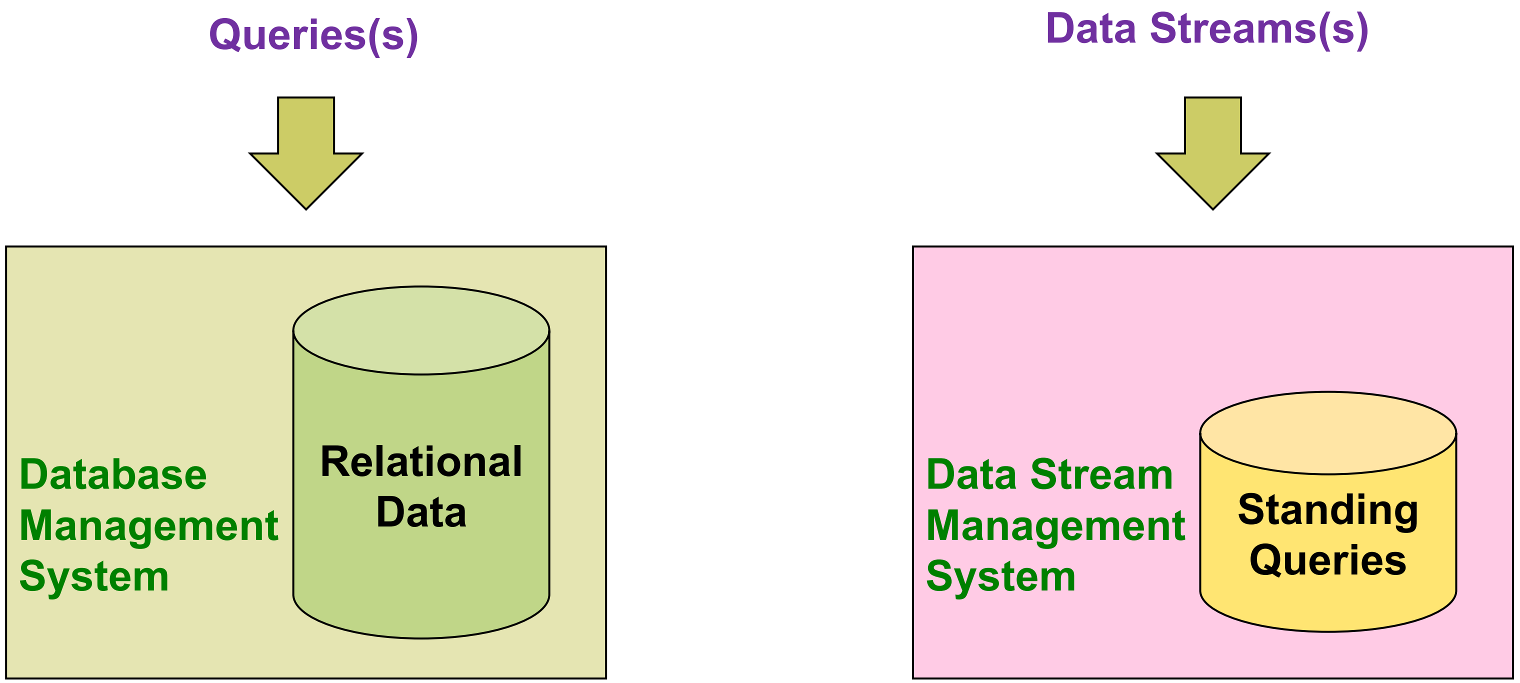
General Stream Processing Model



Streams Entering:

Each is stream is composed of **elements/tuples**

Ad-hoc queries on a DBMS vs. Standing queries on a DSMS



Problems on Data Streams

- Types of queries one wants on answer on a data stream:
  - Filtering a data stream
    - Select elements with property ***x*** from the 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

Applications (1)

- Analyzing query streams
  - Google wants to know what queries are more frequent today than yesterday
- Analyzing click streams
  - Yahoo wants to know which of its pages are getting an unusual number of hits in the past hour
- Analyzing social network news feeds
  - E.g., look for trending topics on Twitter, Facebook

Applications (2)

- Sensor Networks
  - Many sensors feeding into a central controller
- Telephone call records
  - Data feeds into customer bills as well as settlements between telephone companies
- IP packets monitored at a switch
  - Gather information for optimal routing
  - Detect denial-of-service attacks
- Image data
  - Satellites transmit streams of imagery data
  - Surveillance cameras produce image streams

Approaches

- Maintain and use data summaries while processing the stream
- Process standing queries continuously and filter out useless data
- Apply queries and analysis tasks on a sliding window with the last ***k*** elements

- *Many stream processing techniques give approximate result by nature!*

Sampling from a Stream

Sampling from a Stream

- Two different problems:
  1. Sample a fixed proportion of elements in the stream (say 1 in 10)
  2. Maintain a random sample of fixed size s over a potentially infinite stream
     - At each time k, each of the k elements seen so far has equal probability of being sampled
- Objective:
  - ask queries on the selected subset and have the answers be statistically representative of the stream as a whole

Example

- Application: Search engine
- Stream of tuples: (user, query, time)
- Answer questions such as: How often did a user run the same query in a single day
- Space for sample: **$1/10^{th}$** of query stream
- Naïve solution:
  - Generate a random integer in **[0..9]** for each query
  - Keep the query if the integer is **0**, otherwise discard

Problem with Naïve Approach

- Query: 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*+2*d* queries)
  - Correct answer: *d*/(*x*+*d*)
- Proposed solution: We keep 10% of the queries
  - Sample will contain *x*/10 of the singleton queries and 2*d*/10 of the duplicate queries at least once
  - But only *d*/100 pairs of duplicates
    - d/100 = 1/10 · 1/10 · d
  - Of *d* "duplicates" *18d/100* appear exactly once in sample
    - 18d/100 = ((1/10 · 9/10)+(9/10 · 1/10)) · d
- So the sample-based answer is $\frac{\frac{d}{100}}{\frac{x}{10}+\frac{d}{100}+\frac{18d}{100}} = \frac{d}{10x+19d}$

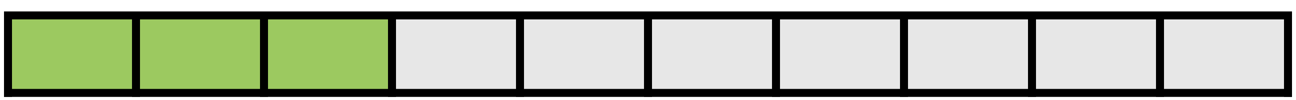
Solution: Sample Users

Solution:

- Pick 1/10th of users and take all their searches in the sample
- Use a hash function that maps the user name or user id uniformly to 10 numbers: 0..9
  - Keep data only from users that hash to number 0
  - Using a hash function avoids explicitly keep users we have chosen before (no lookups)
- In general, we can obtain a sample fraction a/b of the users by hashing to numbers 0 through b−1 and adding to the sample data for users that hash to any value less than a.

Generalized Solution

- Determine the hash key:
  - Key is some subset of each tuple's components
    - e.g., tuple is (user, search, 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

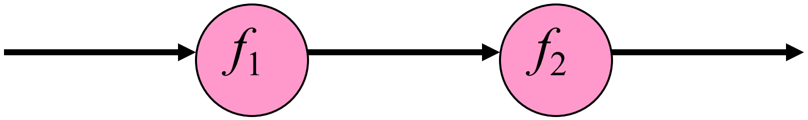
- Varying the sample size:
  - If the sample size grows too large for our memory, reduce a and drop sampled data with hash value >***a***

Stream Filtering

Stream filtering

Problem:

- Accept tuples that meet a filtering criterion and reject all other tuples
- Accepted tuples may be passed to another filter



- If the criterion is a selection based on the attribute values of a tuple then the solution is straightforward
  - Example: accept tuples where query = "cat*"
- Challenge: select based on membership in a set ***S***

Filtering Data Streams

- Given a list of *keys* ***S***, determine which tuples of stream have their *key* attribute in ***S***

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

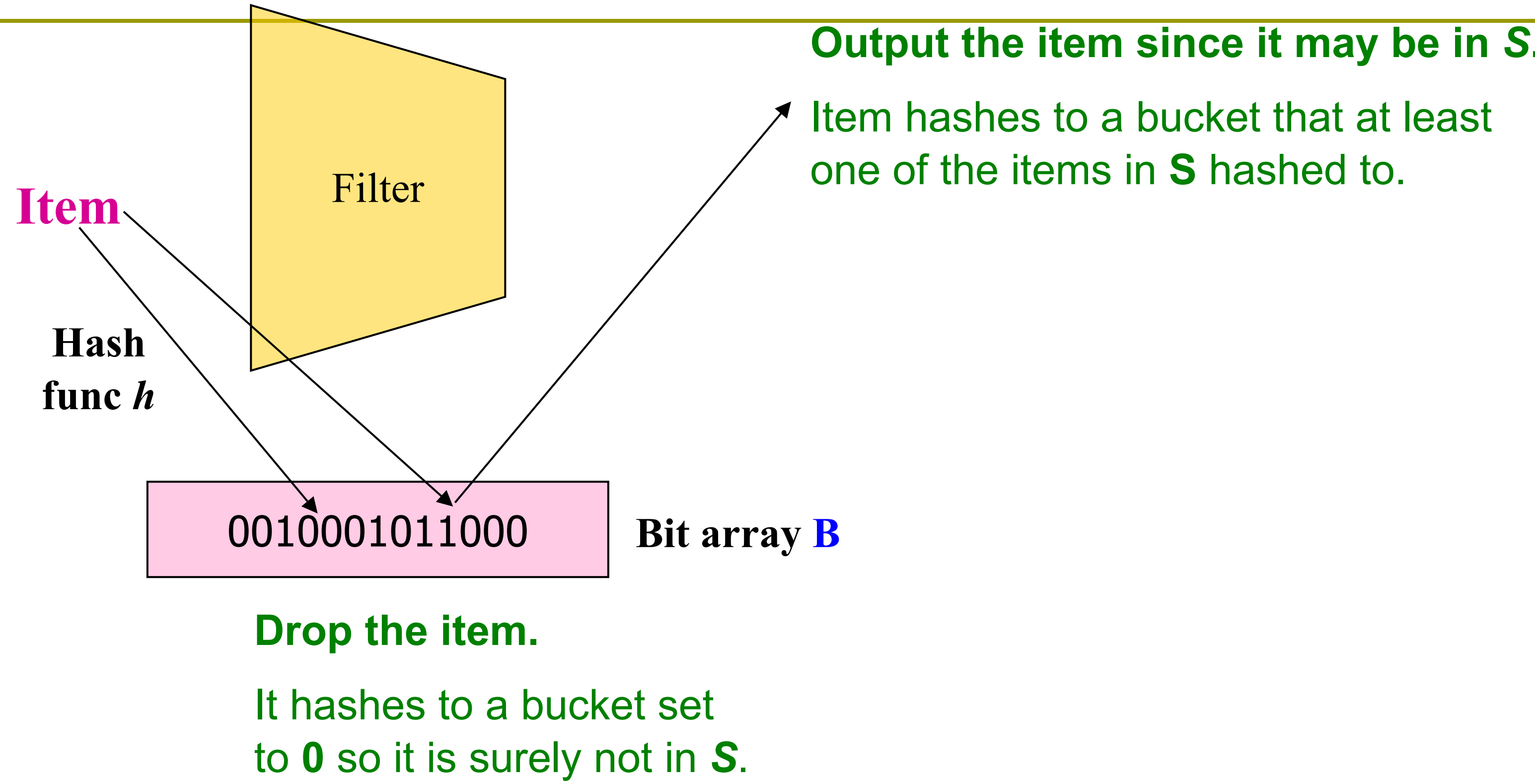
- Publish-subscribe systems
  - You are collecting lots of messages (news articles)
  - People express interest in certain sets of keywords
  - Determine whether each message matches user's interest

- Challenge: ***S*** could be very large

First Cut Solution (1)

- 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 Cut Solution (2)



- Creates false positives but no false negatives
  - If the item is in ***S***, we surely output it
  - If the item is not in ***S***, we may output it (false positive)
  - 100% recall, less than 100% precision

First Cut Solution (3)

- |S| = 1 billion email addresses (=*m*)
  |B|= 1GB = 8 billion bits (=*n*)

- 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 -th of the addresses not in *S* get through to the output (*false positives*)
  - Actually, less than 1/8th, because more than one address in S might hash to the same bit

Analysis: Throwing Darts (1)

- 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 = *n* bits/buckets
  - Darts = *m* hash values of items (emails in S)

Analysis: Throwing Darts (2)

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

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

Equivalent

$$1 - (1 - 1/n)^{n(m/n)}$$

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

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

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

Analysis: Throwing Darts (3)

- Fraction of 1s in the array B =
  = probability of false positive = $1 - e^{-m/n}$

- Example: $m = 10^9$ darts, $n = 8 \cdot 10^9$ targets
  - Fraction of 1s in B = $1 - e^{-1/8} = 0.1175$

- Can we do better?

Bloom Filter

- Consider: $|S| = m$, $|B| = n$
  - *Goal: accept all keys in S, reject most keys not in S*
- 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 $i = 1, \ldots, k$)

(**note:** we have a single array B!)

- 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

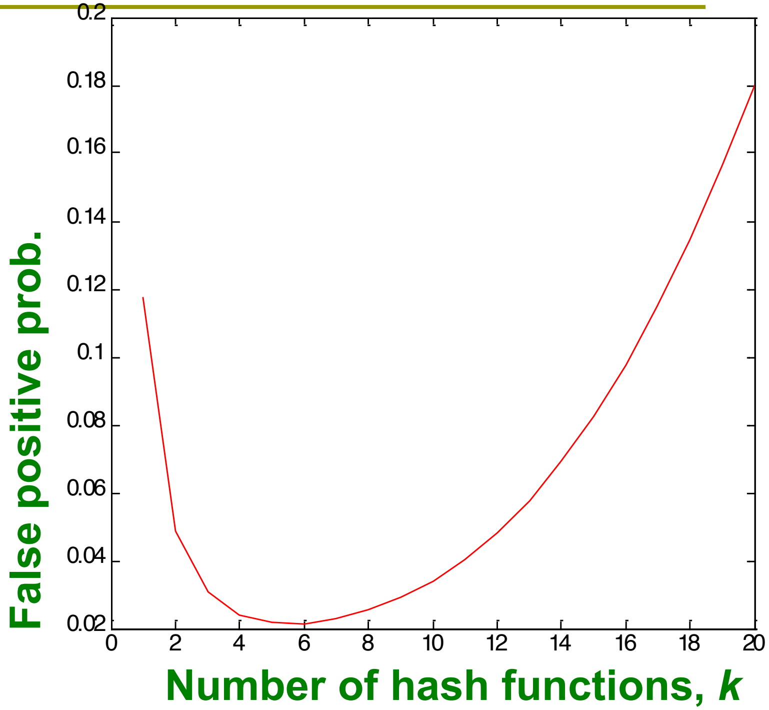
- What fraction of the bit vector B are 1s?
  - Throwing $k \cdot m$ darts at n targets
  - So fraction of 1s is $(1 - e^{-km/n})$

- 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 (2)

- 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 Bs?
  - It is the same: $(1 - e^{-km/n})^k$ vs. $(1 - e^{-m/(n/k)})^k$
  - But keeping 1 big B is simpler

Counting from 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 unique users have visited a given web page each month?
- 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 ***h*** that maps each of the ***N*** elements to at least $\log_2 N$ bits
  - Ensure there are more possible mapped values (hash function results) than elements
- For each stream element ***a***, let ***r*(*a*)** be the number of trailing **0s** in ***h*(*a*)**'s binary representation
  - ***r(a)*** = position of first 1 counting from the right
    - E.g., say ***h(a) = 12***, that is ***1100*** binary, so ***r(a) = 2***
- Record ***R*** = the maximum ***r*(*a*)** seen
  - $\mathbf{R = \max_a r(a)}$, over all the items ***a*** seen so far
- Estimated number of distinct elements = 2^R

Why It Works: Intuition

- Very 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 $\mathbf{log_2 N}$ 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 $m \gg 2^r$
  - Goes to 0 if $m \ll 2^r$

  where 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
  - **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:

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

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

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

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***

Estimating Moments

Generalization: Moments

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

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

- **0th moment =** 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

- Assumption: not enough memory to explicitly count all $\boldsymbol{m_i}$'s
- 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 an 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*** (we relax this later)
  - 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***. *Set* ***X.el=i***
  - Then maintain count ***c*** (***X.val = c***) of the number of ***i***s in the stream starting from the chosen time ***t***
- Then the estimate of the 2nd moment ($\sum_i m_i^2$) is:

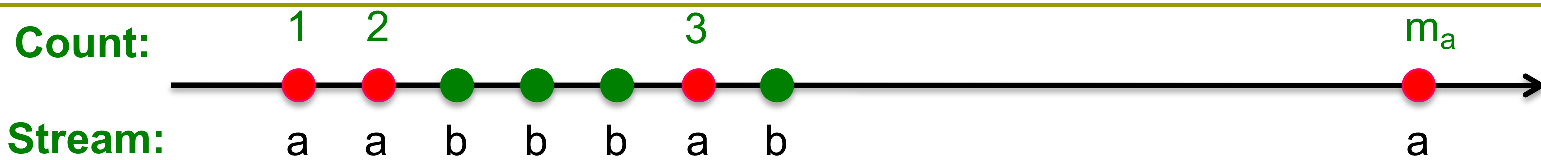
$$\boldsymbol{S = f(X) = n\,(2 \cdot c - 1)}$$

  - Note, we will keep track of multiple **X_i**'s, (**X_1, X_2,… X_k**) and our final estimate will be $\boldsymbol{S = 1/k \sum_j^k f(X_j)}$

Example

- Stream: a,b,c,b,d,a,c,d,a,b,d,c,a,a,b (n=15)
- Real 2nd moment is:
  - $m_a^2+m_b^2+m_c^2+m_d^2 = 5^2 + 4^2 + 3^2 + 3^2 = 59$
- Set X_1, X_2, X_3 at positions 3, 8, 13
  - X_1.el = c, X_2.el = d, X_3.el = a
- From X_1: $n(2X_1.val-1) = 15\times(2\times3-1) = 75$
- From X_2: $n(2X_2.val-1) = 15\times(2\times2-1) = 45$
- From X_3: $n(2X_3.val-1) = 15\times(2\times2-1) = 45$
- Final estimate: average(75,45,45) = 55

Expectation Analysis



- 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_a-2)
- $E[f(X)] = \frac{1}{n}\sum_{t=1}^{n} n(2c_t - 1)$

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

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

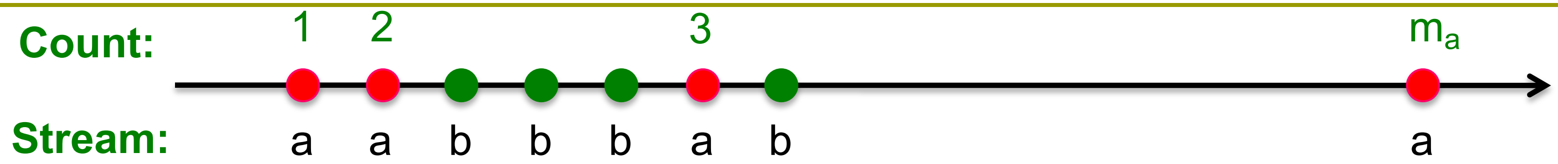
Group times by the value seen

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



- $E[f(X)] = \frac{1}{n}\sum_i n\,(1 + 3 + 5 + \cdots + 2m_i - 1)$
  - Little side 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 $\boldsymbol{E[f(X)] = \frac{1}{n}\sum_i n\,(m_i)^2}$
- So, $\mathbf{E[f(X)]} = \sum_i (\boldsymbol{m_i})^2 = \boldsymbol{S}$
- 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 $n\ (2 \cdot c - 1)$
  - For k=3 we use: $n\ (3 \cdot c^2 - 3c + 1)$ (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 m^2
    - $\sum_{c=1}^{m} 2c - 1 = \sum_{c=1}^{m} c^2 - \sum_{c=1}^{m} (c-1)^2 = m^2$
    - So: $2c - 1 = c^2 - (c-1)^2$
  - For k=3: $c^3 - (c-1)^3 = 3c^2 - 3c + 1$
- Generally: Estimate $= n\ (c^k - (c-1)^k)$

Streams Never End: Problem

- The number ***n*** of positions is infinite
- Assumption: times ***t*** picked at random
- If we pick most times ***t*** early, we will overestimate moment
- If we delay picking, we will under-estimate moment first, maybe overestimate later

Streams Never End: Fixup

- 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 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

General problem: 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
  - 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 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: 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*** of size ***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*** $= \frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1}$

Sliding Window Queries

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
- 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 have we sold **X** in the last ***k*** sales, where $k \le N$

Sliding Window: 1 Stream

- Sliding window on a single stream:

N = 6

q w e r t y u i o p a s d f g h j k l z x c v b n m

q w e r t y u i o p a s d f g h j k l z x c v b n m

q w e r t y u i o p a s d f g h j k l z x c v b n m

q w e r t y u i o p a s d f g h j k l z x c v b n m

← Past Future →

Counting Bits (1)

- **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 $k \leq N$

- **Obvious solution:** Store the most recent N bits
  - When new bit comes in, discard the $N+1^{st}$ bit

0 1 0 0 1 1 0 1 1 1 0 1 0 1 0 1 1 0 $\boxed{1\ 1\ 0\ 1\ 1\ 0}$ Suppose N=6

← Past Future →

Counting Bits (2)

- 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   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 →

- But we are happy with an approximate answer

An attempt: Simple solution

- Q: How many 1s are in the last *N* bits?
- A simple solution that does not really solve our problem: Uniformity assumption
- 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 N bits? $N \cdot \frac{S}{S+Z}$
- But, what if stream is non-uniform?
  - What if distribution changes over time?

DGIM Method

- DGIM solution that does <u>not</u> assume uniformity
- We store $O(log^2 N)$ 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

DGIM method

- Idea: Summarize blocks with specific number of **1s**:
  - Let the block *sizes* (number of **1s**) increase exponentially
    - The number of 1s in a block is a power of 2
- When there are few 1s in the window, block sizes stay small, so errors are small

10010101100010110101010101010110101010101011101010111010100010110010

← N →

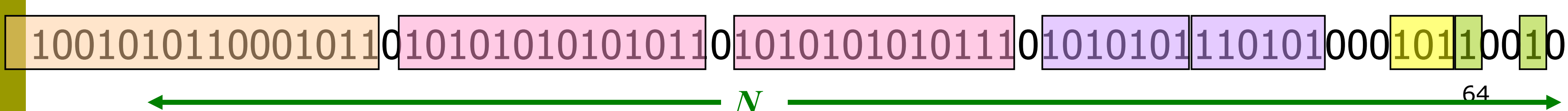
DGIM: Timestamps

- 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

DGIM: Buckets

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

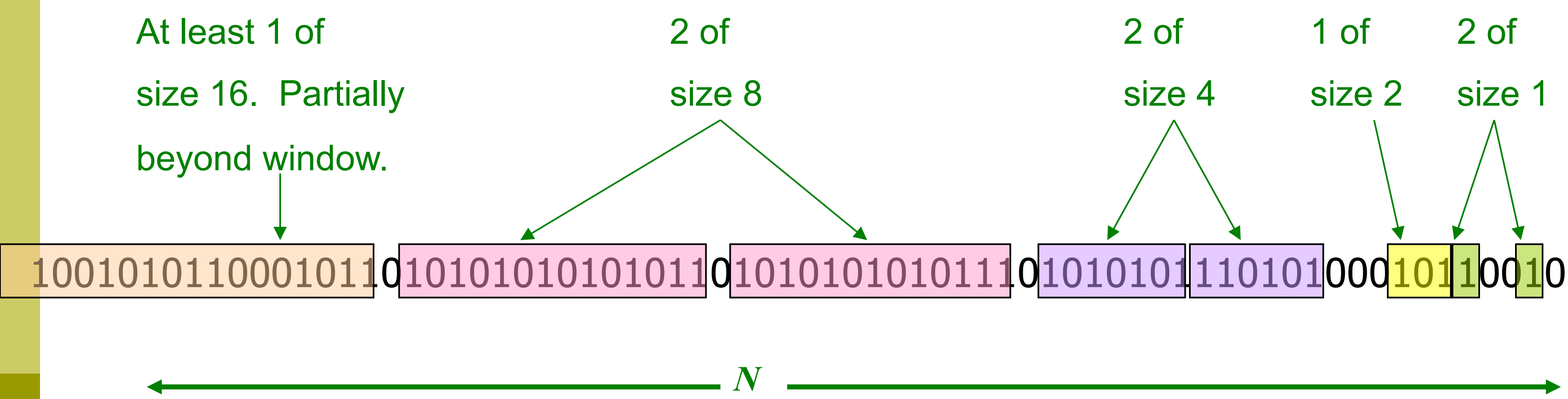
- Constraint on buckets:
  Number of 1s must be a power of 2
  - That explains the O(log log N) in (B) above



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

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 (1)

- 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 (2)

- 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

001010110001011 0 1010101010101011 0 101010101010111 0 1010101 110101 000 101 1 00 1 0 **1**

Two orange buckets get merged into a yellow bucket

001010110001011 0 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:

010110001011 0 1010101010101011 0 101010101010111 0 1010101 110101 000 101 1001 0 1 **1** **0** **1**

Buckets get merged…

010110001011 0 1010101010101011 0 101010101010111 0 1010101 110101 000 101 1001 0 1 **1** **0** **1**

State of the buckets after merging

010110001011 0 1010101010101011010101010101110 0 1010101110101 000 1011001 0 11 0 1

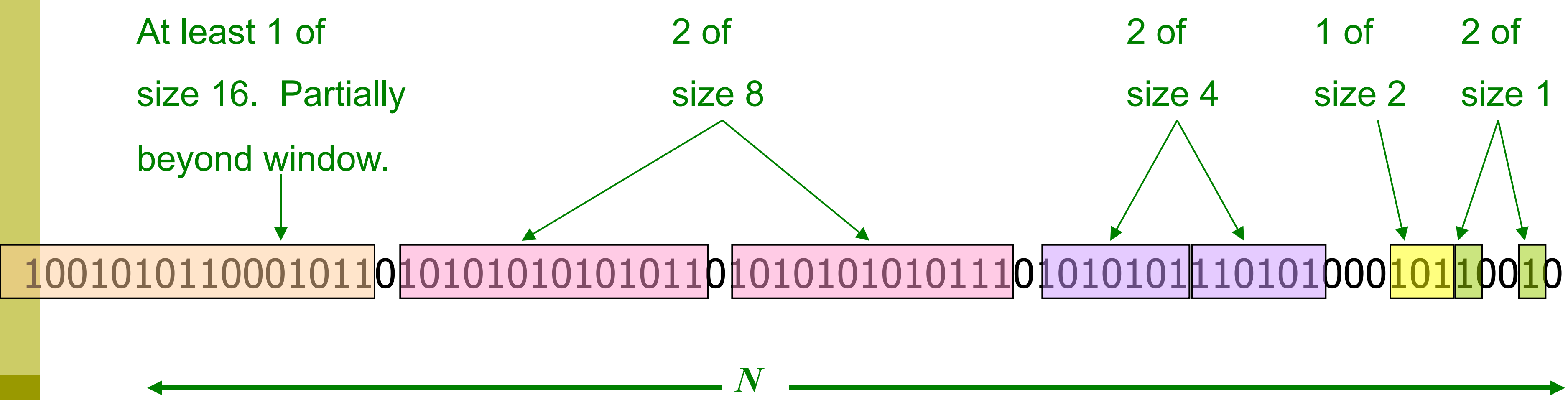
How to Query?

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

     (note "size" means the number of 1s in the bucket)
  2. Add half the size of the last bucket

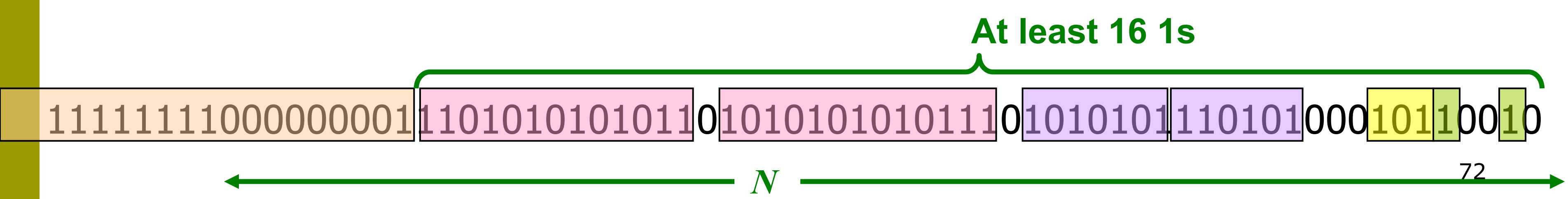
- Remember: We do not know how many 1s of the last bucket are still within the wanted window

Example: Bucketized Stream



Error Bound: Proof

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



Further Reducing the Error

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

Extensions

- Can we use the same trick to answer queries How many 1's in the last *k*? where $k < N$?
  - A: Find earliest bucket B that at overlaps with *k*. Number of 1s is the sum of sizes of more recent buckets + ½ size of B

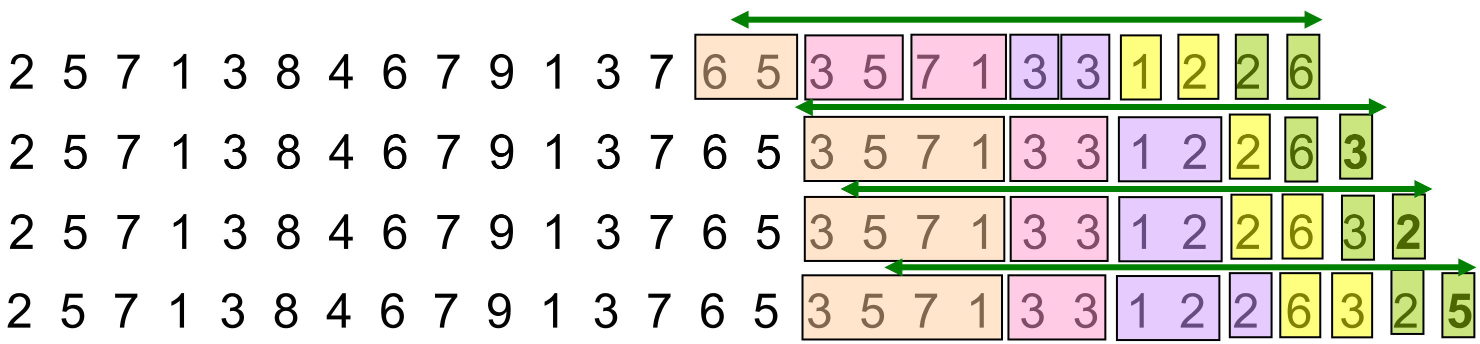
1001010110001011 0 101010101010111 0 101010101010111 0 1010101 110101 000 101 1 00 1 0

$\xleftarrow{\quad\quad} k \xrightarrow{\quad\quad\quad\quad\quad\quad}$

- Can we handle the case where the stream is not bits, but integers, and we want the sum of the last *k* elements?

Extensions

- Stream of positive integers
- We want the sum of the last *k* elements
  - Amazon: Avg. price of last k sales
- Solution:
  - Use buckets to keep partial sums
    - Sum of elements in size *b* bucket is at most 2^b



Idea: Sum in each bucket is at most $\mathbf{2^b}$ (unless bucket has only **1** integer)

Bucket sizes:

16 8 4 2 1

Exponentially Decaying Windows

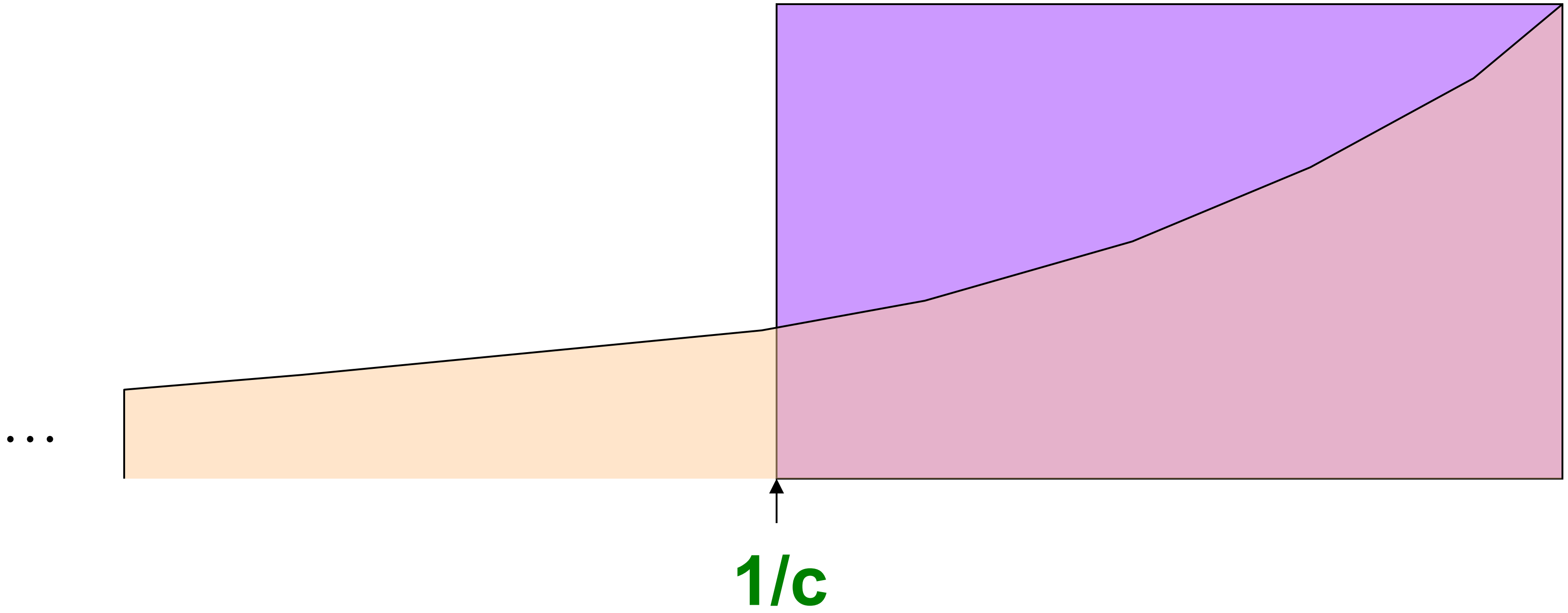
Exponentially Decaying Windows

- Exponentially decaying windows: A heuristic for selecting likely frequent items
  - 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 a_{t+1} arrives:
  - Multiply current sum by (1-c) and add a_{t+1}

Example: Counting Items

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

Sliding Versus Decaying Windows



- Important property: Sum over all weights $\sum_t (1-c)^t$ is 1/[1 – (1 – c)] = 1/c

Example: Counting Items

- What are "currently" most popular movies?
- When a new ticket arrives on the stream, do the following:
  - For each movie whose score we are currently maintaining, multiply its score by (1 − c)
  - Suppose the new ticket is for movie M. If there is currently a score for M, add 1 to that score. If there is no score for M, create one and initialize it to 1.
  - If any score is below the threshold 1/2, drop movie from counting (drop score)
- 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

Summary

- Streaming data are ubiquitous in applications
- Objective: read the stream just once and compute some statistics, given limited storage
- Specialized objectives:
  - Sampling from a stream
  - Filtering streaming data
  - Counting data from a stream
  - Estimating moments
  - Sliding window queries
  - Exponentially decaying windows