Streaming - 1

Proportional Sampling, Reservoir Sampling, DGIM

credits:www.mmds.org

What is a stream ?

- **In many data mining scenarios, we do not know the entire data set in advance**

- **Stream Management** is important when the input rate is controlled **externally:**
  - Google queries
  - Twitter or Facebook status updates

- 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**

We can think of the data as infinite and non-stationary (the distribution changes over time)

Scenarios

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

Streaming Questions

- **How do we sample from a stream ? Given a stream of items:**
  - How do we sample a **fixed proportion** of elements in the stream (say 1 in 10) ?
  - Maintain a **random sample of fixed size** over a potentially infinite stream ?
- **How do we count elements in a stream ?**
  - How do we count the frequency of an item in the last n observed items ?

Sampling a Fixed Proportion

- **Problem 1: Sampling fixed proportion**

Sampling a Fixed Proportion

- **Problem 1: Sampling fixed proportion**

- **Scenario:** Search engine query stream
  - **Stream of tuples:** (user, query, time)
  - **Answer questions such as: How often did a user run the same query in a single days**
  - Have space to store **1/10th** of query stream

Sampling a Fixed Proportion

- **Problem 1: Sampling fixed proportion**

- **Scenario:** Search engine query stream
  - **Stream of tuples:** (user, query, time)
  - **Answer questions such as: How often did a user run the same query in a single days**
  - 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 Naïve Approach

- **Scenario: What fraction of queries by an user are duplicates ?**

Problem with Naïve Approach

- **Scenario: What fraction of queries by an user are duplicates ?**
- **Let each user issues x queries once and d queries exactly twice. (x + 2d queries in total)**
  - **Answer:** d / (x+d)
- **What is the expected number of repetitions in the sample produced by naive sampling ?**
  - **x/10 singletons, 2d/10 duplicates**

Problem with Naïve Approach

- **Scenario: What fraction of queries by an user are duplicates ?**

- **Let each user issues x queries once and d queries exactly twice. (x + 2d queries in total)**
  - **Answer:** d / (x+d)
- **What is the expected number of repetitions in the sample produced by naive sampling ?**
  - **x/10 singletons, 2d/10 duplicates**
  - **But only d/100 pairs of duplicates — (1/10).(1/10).d**
- **Of d duplicates 18d/100 appear exactly once**
  - **18d/100 = ((1/10 . 9/10) + (9/10 + 1/10)) . d**

- **Sample Answer: d/ (10x + 19d)**

  - **duplicates = d/100, overall - (d/100) + (x/10) + (18d/100)**

Solution: Sample Users

Solution:

- 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

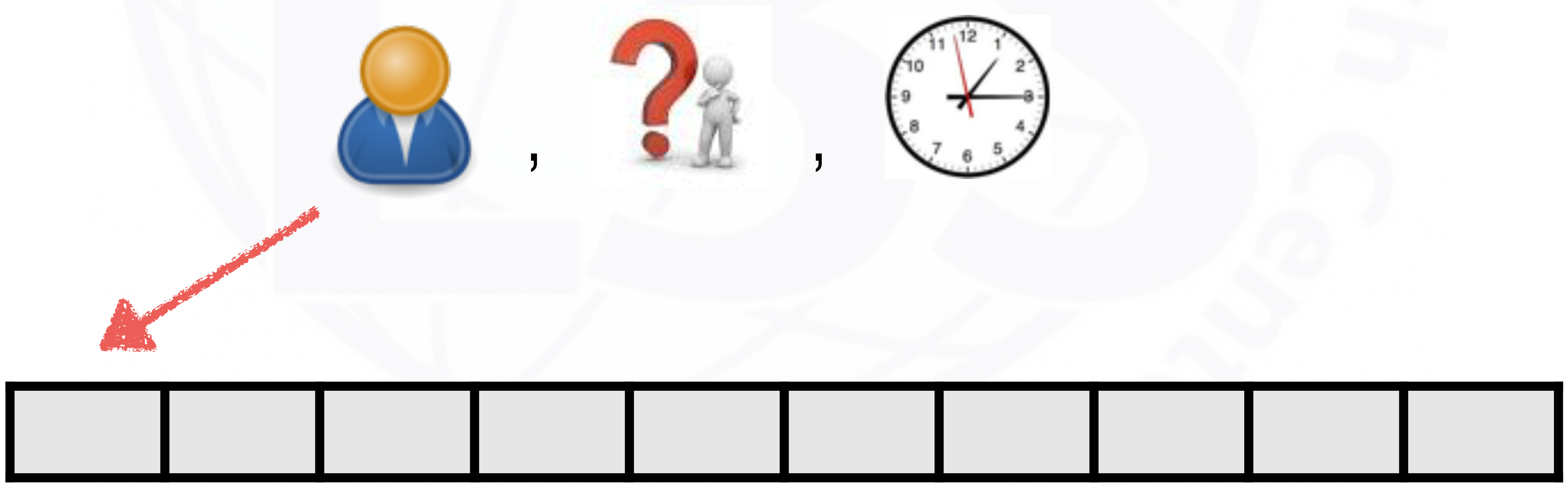
 ,  , 

Solution: Sample Users

Solution:

- 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

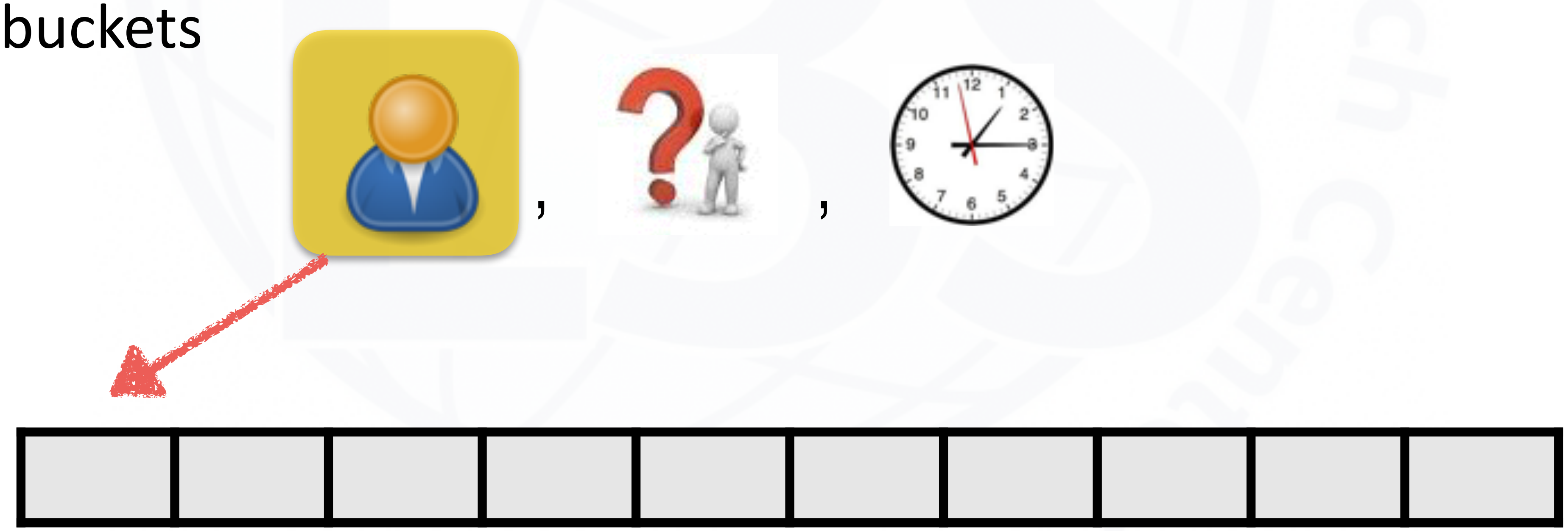


Solution: Sample Users

Solution:

- Pick **1/10th** of **users** and take all their searches in the sample

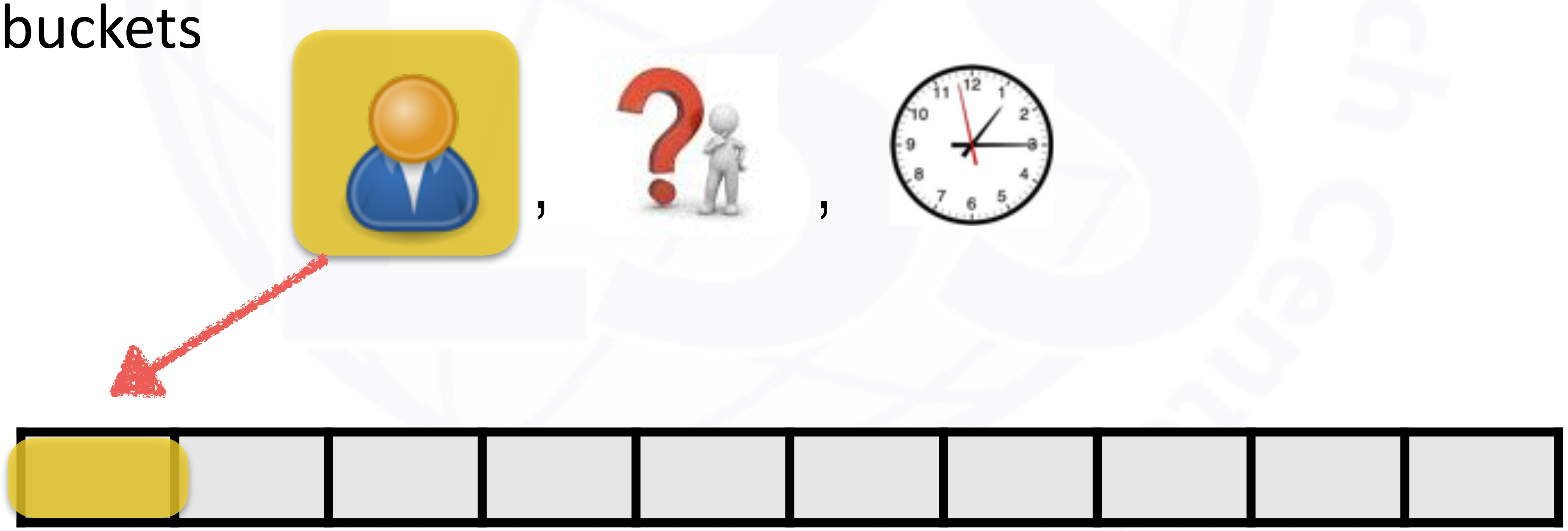
- Use a hash function that hashes the user name or user id uniformly into 10 buckets



Solution: Sample Users

Solution:

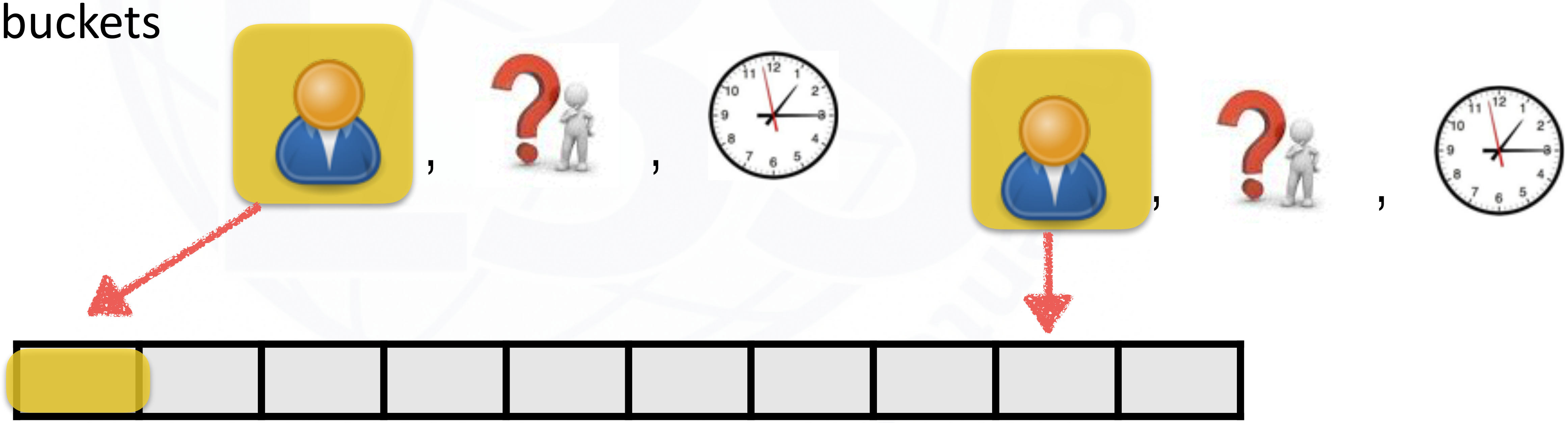
- Pick **1/10th** of **users** and take all their searches in the sample
- Use a hash function that hashes the user name or user id uniformly into 10 buckets



Solution: Sample Users

Solution:

- 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



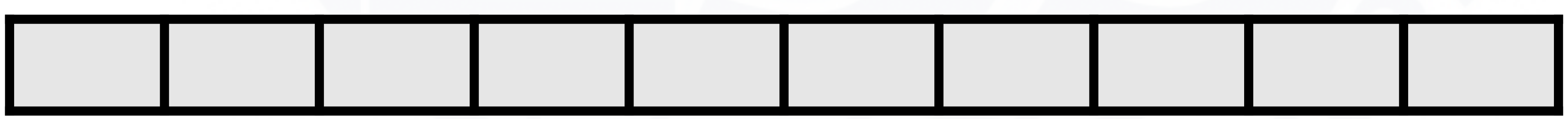
Generalized Solution

- **Stream of tuples with keys:**
  - 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

Generalized Solution

- **Stream of tuples with keys:**
  - 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

Maintaining a fixed-size sample

- **Problem 2: Fixed-size sample**

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

Maintaining a fixed-size sample

- **Problem 2: Fixed-size sample**
- **Suppose we need to maintain a random sample *S* of size exactly *s* tuples**
  - E.g., main memory size constraint
- **Why?** Don't know length of stream in advance

Maintaining a fixed-size sample

- **Problem 2: Fixed-size sample**

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

- **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 prob. *s/n***

Maintaining a fixed-size sample

- **Problem 2: Fixed-size sample**
- **Suppose we need to maintain a random sample *S* of size exactly *s* tuples**
  - E.g., main memory size constraint
- **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 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.

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

Reservoir Sampling

- **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

Reservoir Sampling

- **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:

Reservoir Sampling

- **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: 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)***

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

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:

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

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

Counting in a Stream - 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
- **Twitter example:**
    - Use-case : tracking hashtags — *#sonyps4 #ps4 #xbox360* ….

Sliding Window: 1 Stream

N = 6

Past Future

- **Sliding window on a single stream:**

Sliding Window: 1 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

← Past Future →

- **Sliding window on a single stream:**

Sliding Window: 1 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

Past Future

- **Sliding window on a single stream:**

Sliding Window: 1 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

← Past Future →

- **Sliding window on a single stream:**

Sliding Window: 1 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 →

- **Sliding window on a single stream:**

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

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+1st** bit

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**

- **Naive approach: Uniform assumption, interpolation**

Data can be non-uniform. Distribution changes over time.

DGIM method

[Datar, Gionis, Indyk, Motwani]

DGIM method

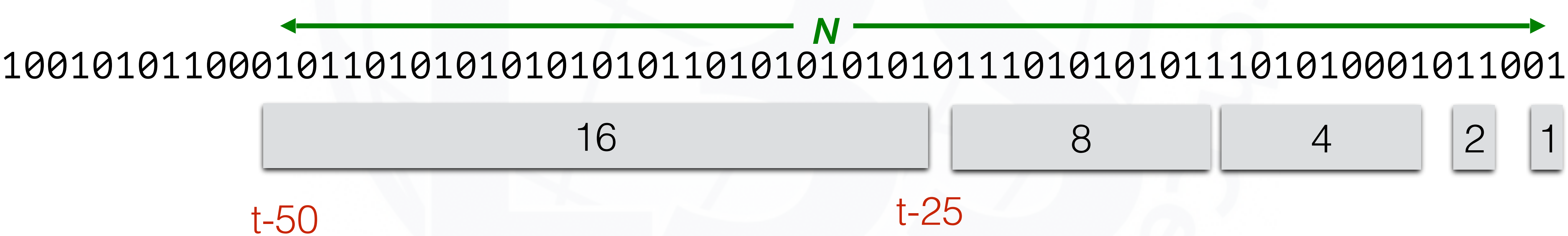
[Datar, Gionis, Indyk, Motwani]

- **Idea:** Keep non-overlapping blocks with counts.
  - **Memory:** How do we block ? How many blocks ?
  - **Accuracy:** How do we provide guarantee on the count errors ?

DGIM method

[Datar, Gionis, Indyk, Motwani]

- **Idea:** Keep non-overlapping blocks with counts.
  - **Memory:** How do we block ? How many blocks ?
  - **Accuracy:** How do we provide guarantee on the count errors ?

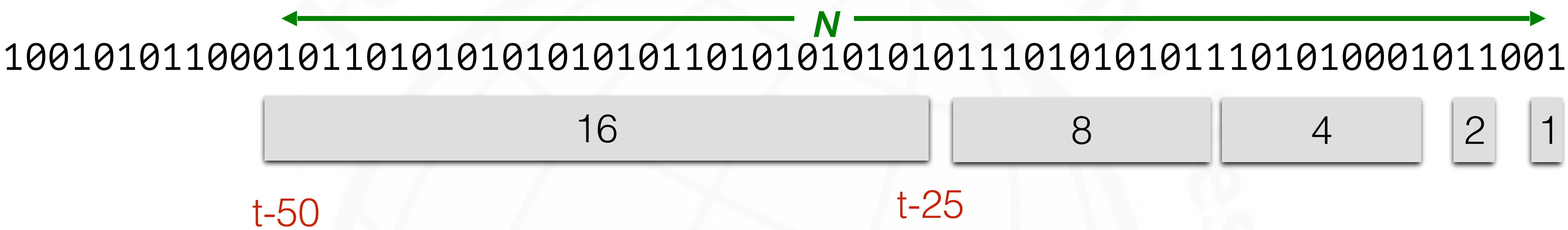


Each block contains a count = power of 2

Each block belongs to a interval

DGIM method

[Datar, Gionis, Indyk, Motwani]

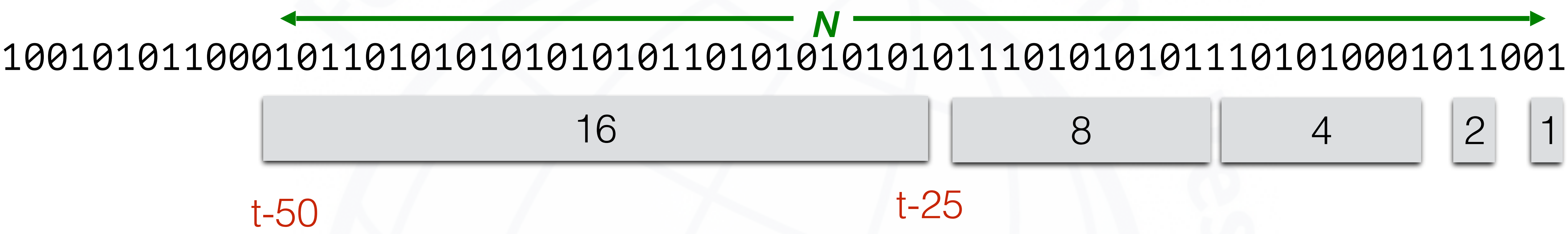


Each block encodes an interval

Each block contains a count = power of 2

DGIM method

[Datar, Gionis, Indyk, Motwani]

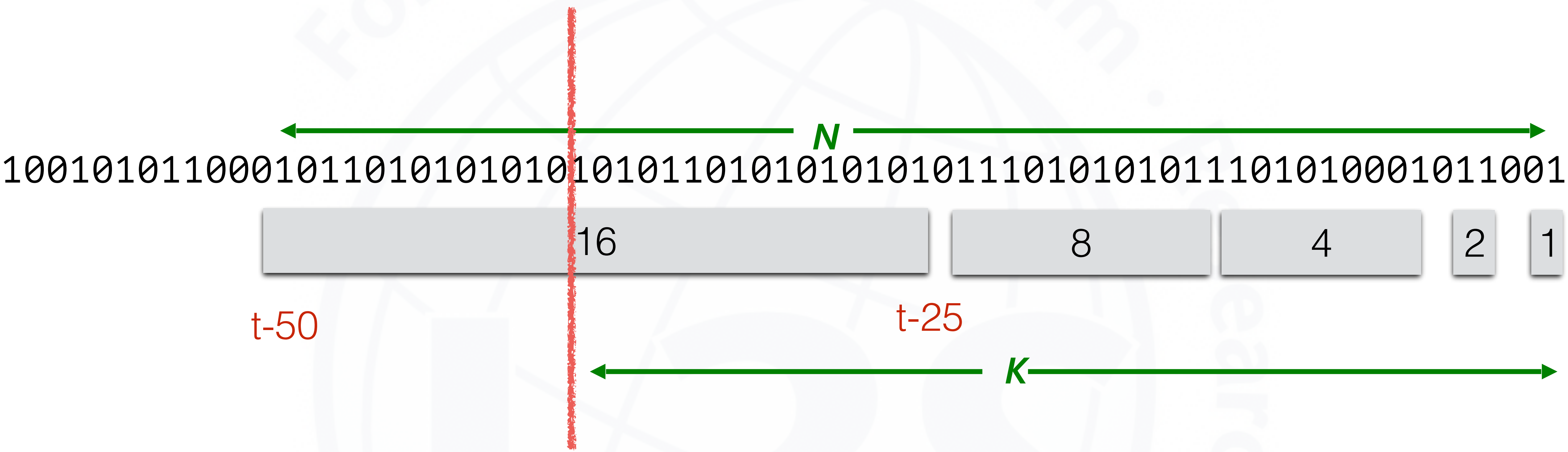


Each block encodes an interval

Each block contains a count = power of 2

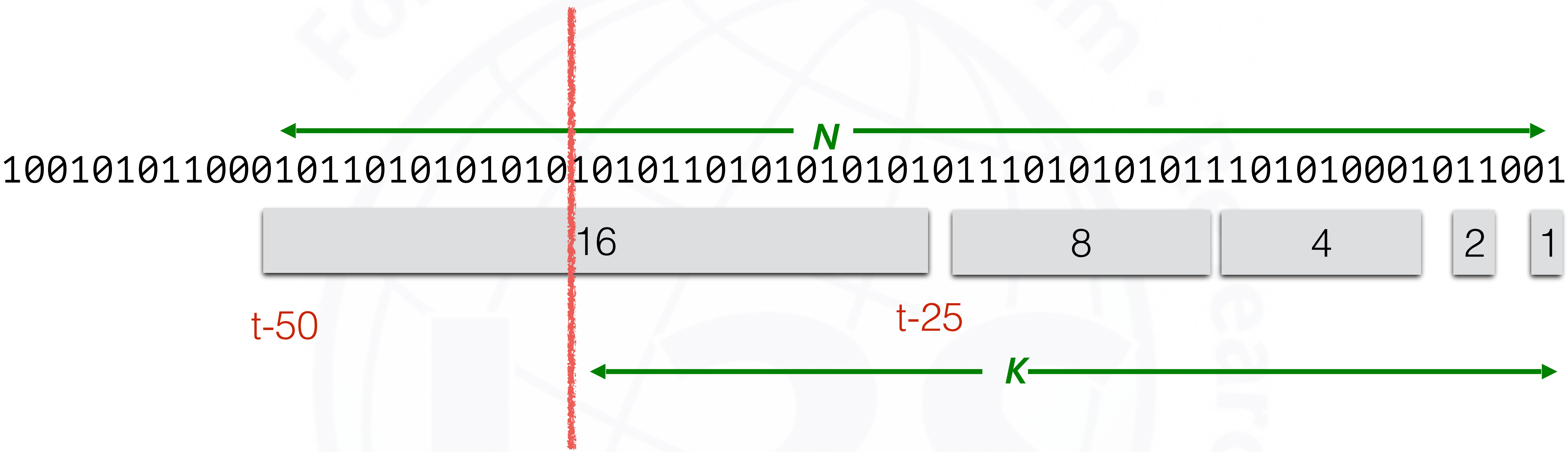
- **Space overhead: $O(\log^2 N)$**
  - **Number of buckets: log N**
  - **Max bits reqd. per bucket: log N (time stamps, count)**

DGIM method- Counting



1/2 size of last bucket

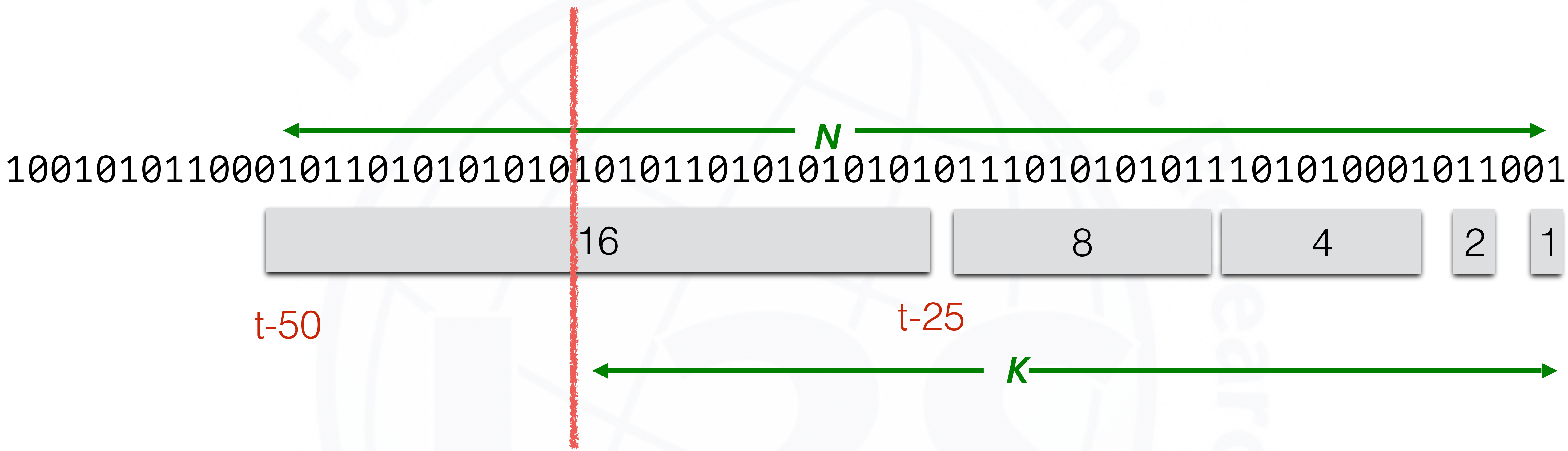
DGIM method- Counting



- Find the number of 1's in the last k entries
- Find the affected buckets in the timespan of the query using timestamps
    - At most 1 bucket with inexact counts
    - Count all affected buckets (exact) + estimate the count in the last bucket (approximate)

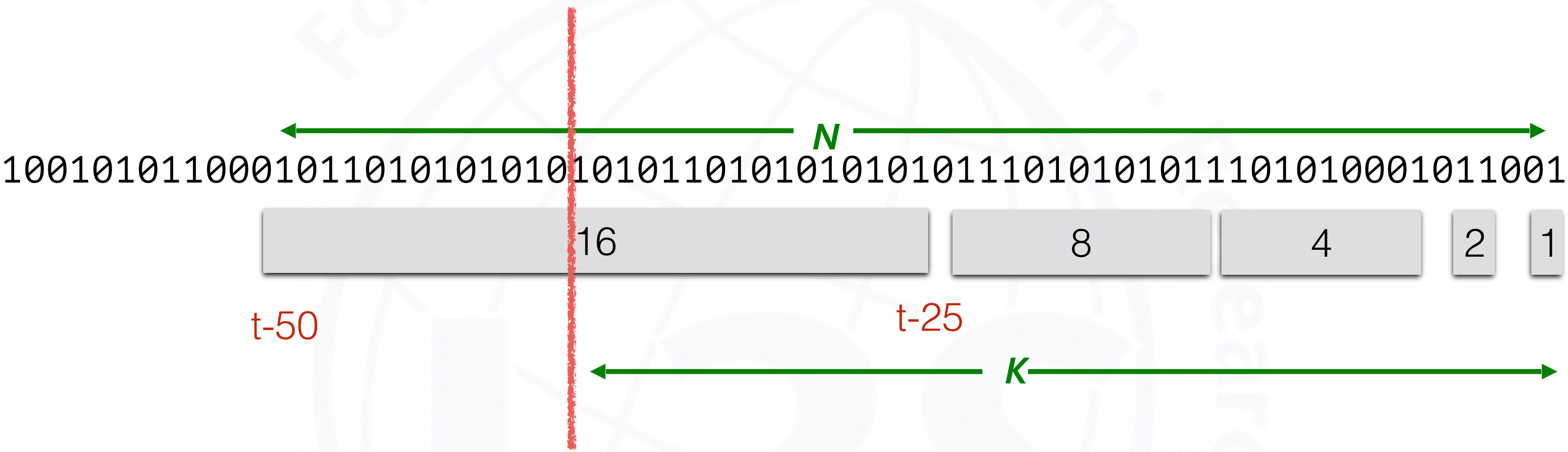
1/2 size of last bucket

DGIM method- Accuracy



$$\sum_i 2^i = 2^{i+1} - 1$$

DGIM method- Accuracy

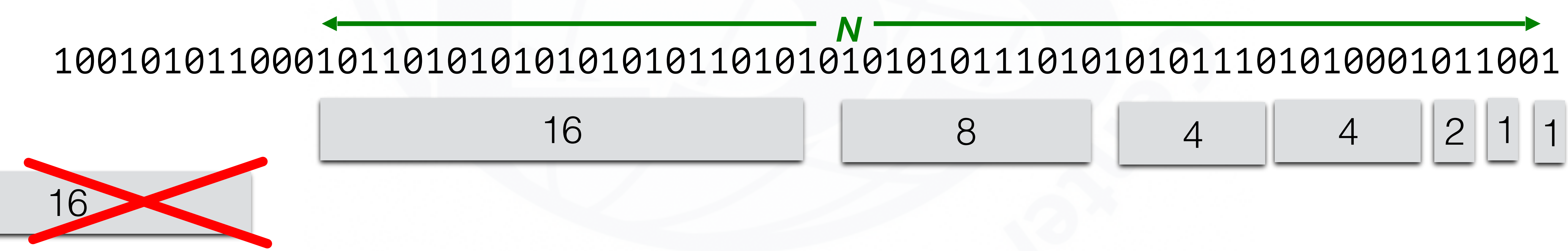


- Error rate of 50% in the worst case
- **Proof insight:** earliest affected bucket does not contribute more than 50% of the actual answer

$$\sum_i 2^i = 2^{i+1} - 1$$

DGIM method- Maintaining 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 (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

Example: Updating Buckets

Current state of the stream:

100101011000101101010101010101101010101010111010101011101011000101110 01

Example: Updating Buckets

Current state of the stream:

100101011000101101010101010101011010101010101110101010111010100010110011

Bit of value 0 arrives do nothing,1 arrives

10010101100010110101010101010101101010101010111010101011101010001011001011

Example: Updating Buckets

Current state of the stream:

1001010110001011010101010101011010101010101110101010111010100010110011

Bit of value 0 arrives do nothing,1 arrives

100101011000101101010101010101101010101010111010101011101010001011001011

Two orange buckets get merged into a yellow bucket

100101011000101101010101010101101010101010111010101011101010001011001011

Example: Updating Buckets

Current state of the stream:

1001010110001011010101010101011010101010101110101010111010100010110011

Bit of value 0 arrives do nothing,1 arrives

10010101100010110101010101010110101010101011101010101110101000101100101

Two orange buckets get merged into a yellow bucket

10010101100010110101010101010110101010101011101010101110101000101100101

Next bit 1 arrives

100101011000101101010101010101101010101010111010101011101010001011001011

Example: Updating Buckets

Current state of the stream:

100101011000101101010101010101101010101010111010101011101010001011001

Bit of value 0 arrives do nothing,1 arrives

1001010110001011010101010101011010101010101110101010111010100010110010 1

Two orange buckets get merged into a yellow bucket

1001010110001011010101010101011010101010101110101010111010100010110010 1

Next bit 1 arrives

1001010110001011010101010101011010101010101110101010111010100010110010 1 1

Next bit 1 arrives Buckets get merged…

1001010110001011010101010101011010101010101110101010111010100010110010 1 1 1

Example: Updating Buckets

Current state of the stream:

10010101100010110101010101010110101010101011101010101110101000101100 1

Bit of value 0 arrives do nothing,1 arrives

100101011000101101010101010101101010101010111010101011101010001011001 01

Two orange buckets get merged into a yellow bucket

1001010110001011010101010101011010101010101110101010111010100010110010 1

Next bit 1 arrives

1001010110001011010101010101011010101010101110101010111010100010110010 1 1

Next bit 1 arrives Buckets get merged…

1001010110001011010101010101011010101010101110101010111010100010110010 1 1 1

State of the buckets after merging

1001010110001011010101010101011010101010101110101010111010100010110010 1 1 1

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 size buckets; we can have any number between **1** and ***r*** of those

- What is the space requirement for r buckets ?

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 size buckets; we can have any number between **1** and ***r*** of those

- What is the space requirement for r buckets ?

- **Error is at most *O*(1/*r*)**

- By picking ***r*** appropriately, we can tradeoff between number of bits we store and the error

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:
    - Number of 1s in any last k (k < N) elements
    - Sums of integers in the last N elements

Appendix Slides