



Mining Data Streams

Topic: Infinite Data

High dim. data

Locality
sensitive
hashing

Clustering

Dimensional
ity
reduction

Graph data

PageRank,
SimRank

Community
Detection

Spam
Detection

Infinite data

Filtering
data
streams

Queries on
streams

Web
advertising

Machine learning

SVM

Decision
Trees

Perceptron,
kNN

Apps

Recommen
der systems

Association
Rules

Duplicate
document
detection

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**:
 - Google queries
 - Twitter or Facebook status 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**
 - Eg: (user, query, time)
- **The system cannot store the entire stream accessibly**
- **Q: How do you make critical calculations about the stream using a limited amount of (secondary) memory?**

Stream data processing

- Stream of **tuples** arriving at a **rapid rate**
 - In contrast to **traditional DBMS** where all tuples are stored in secondary storage
- Infeasible to **use all tuples** to answer queries
 - Can not store them all in **main memory**
 - Too much **computation**
 - Query response **time critical**.

Query types

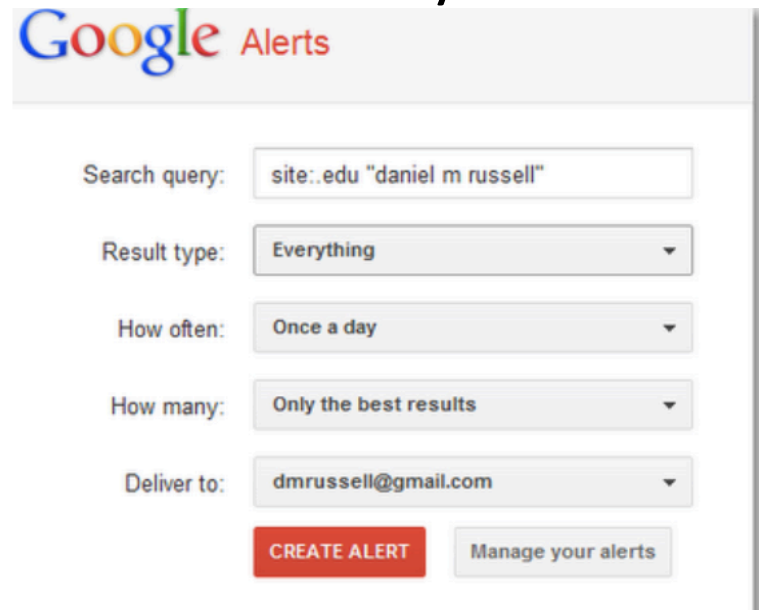
- **Standing queries**

- Executed whenever a **new tuple arrives**
- e.g., report each **new maximum value** ever seen in the stream

- **Ad-hoc queries**

Standing queries

- **Google Alerts--standing queries to monitor the world**
 - Google Alerts are basically "standing queries." You write a Google query, then decide **how often you want it to run** and over what body of content (news, web sites, etc.).



The screenshot shows the Google Alerts configuration page. At the top is the 'Google Alerts' logo. Below it are several input fields and dropdown menus: 'Search query' with the text 'site:.edu "daniel m russell"', 'Result type' set to 'Everything', 'How often' set to 'Once a day', 'How many' set to 'Only the best results', and 'Deliver to' set to 'dmrussell@gmail.com'. At the bottom are two buttons: a red 'CREATE ALERT' button and a grey 'Manage your alerts' button.

- <http://searchresearch1.blogspot.com/2012/01/google-alerts-standing-queries-to.html>

Query types

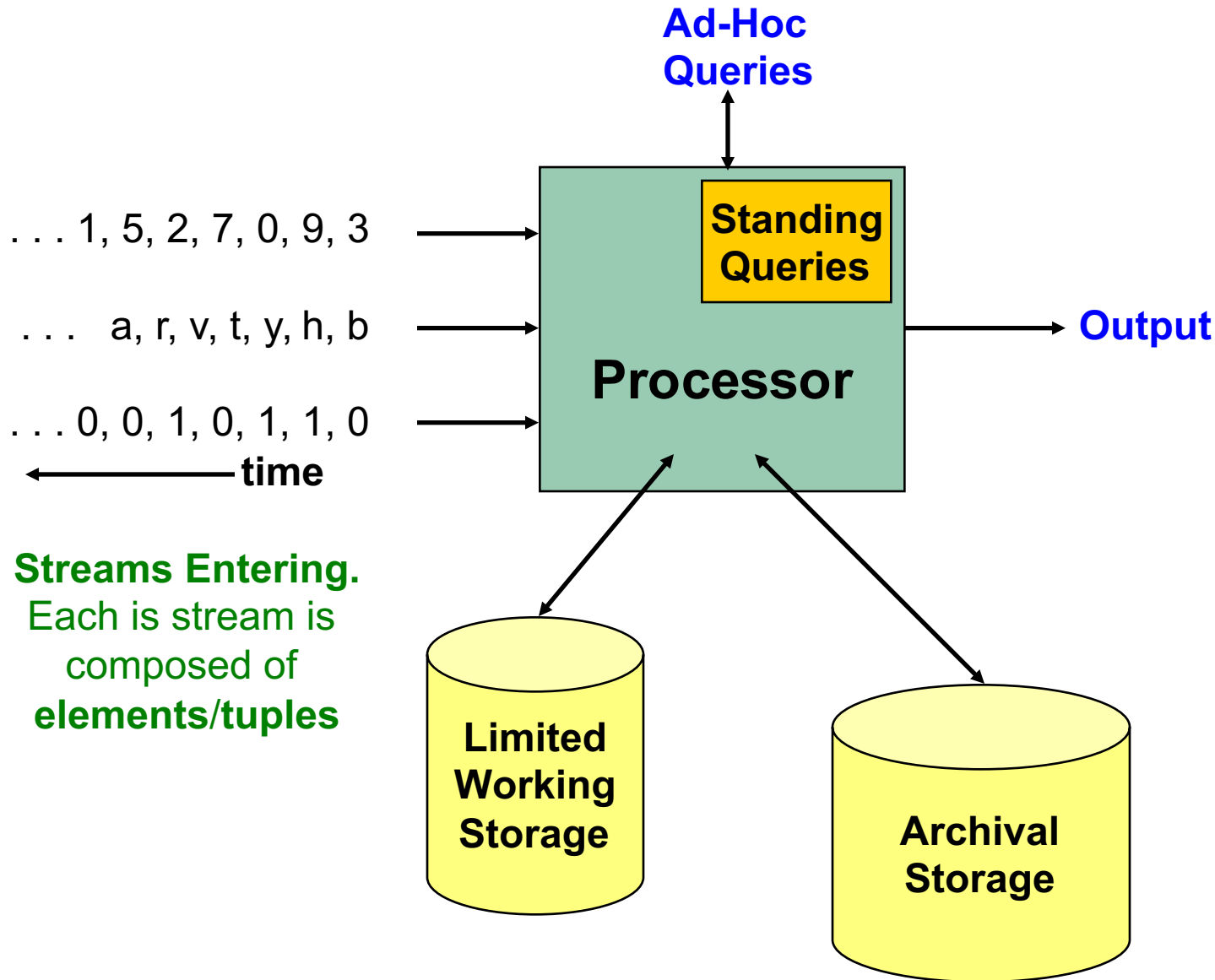
- **Standing queries**

- Executed whenever a **new tuple arrives**
- e.g., report each **new maximum value** ever seen in the stream

- **Ad-hoc queries**

- Normal queries asked **one time**
- **Ad hoc** comes from Latin which means "for the purpose"
- E.g., what is the **maximum value so far?**

General Stream Processing Model



Example: Running averages

- Given a window of size N
 - report the average of values in the window whenever **a value arrives**
 - **N is so large** that we can not store all tuples in the window
- How to do this?

Example: running averages

- First N inputs, accumulate sum and count
 - $\text{Avg} = \text{sum}/\text{count}$
- A new element i
 - Change the average by adding $(i-j)/N$
 - j is the oldest element in the window.

Problems on Data Streams

- **Types of queries one wants on answer on a data stream:**
 - **Queries over sliding windows**
 - Number of items of type x in the last k elements of the stream
 - **Sampling data from a stream**
 - Construct a random sample.

Problems on Data Streams (2)

- **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

- **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.

Queries over a (long) 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
- **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.**

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

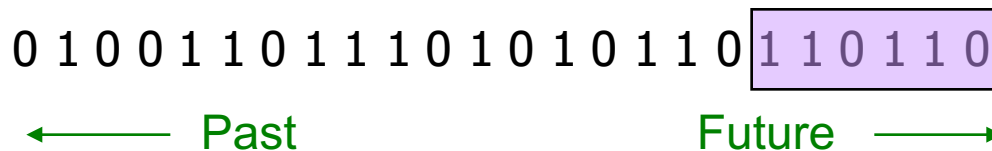
■ Problem:

- Given a stream of **0s** and **1s**
- 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 $N+1^{\text{st}}$ bit



Suppose $N=6$

Counting Bits (2)

- Obvious solution: store the most recent N bits
- But answering the query will take $O(k)$ time
 - Very possibly too much time
- And the **space requirements** can be too great
 - Especially if there are many streams to be managed in **main memory at once**, or N is huge.

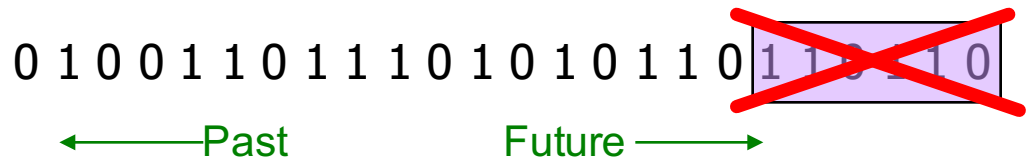
Counting Bits (3)

- You can not get an exact answer without storing the entire window

- **Real Problem:**

What if we cannot afford to store N bits?

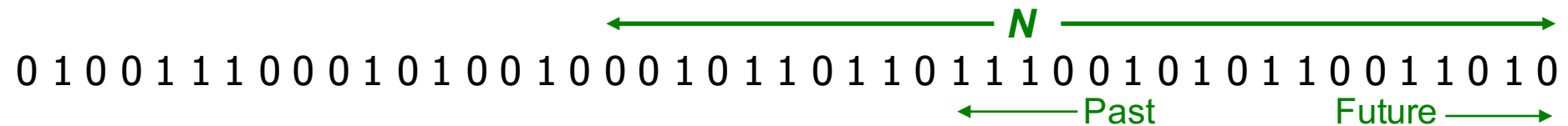
- E.g., we're processing 1 billion streams and
 $N = 1$ billion



- **But we are happy with an approximate answer.**

An attempt: Simple solution

- Q: How many 1s are in the last 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

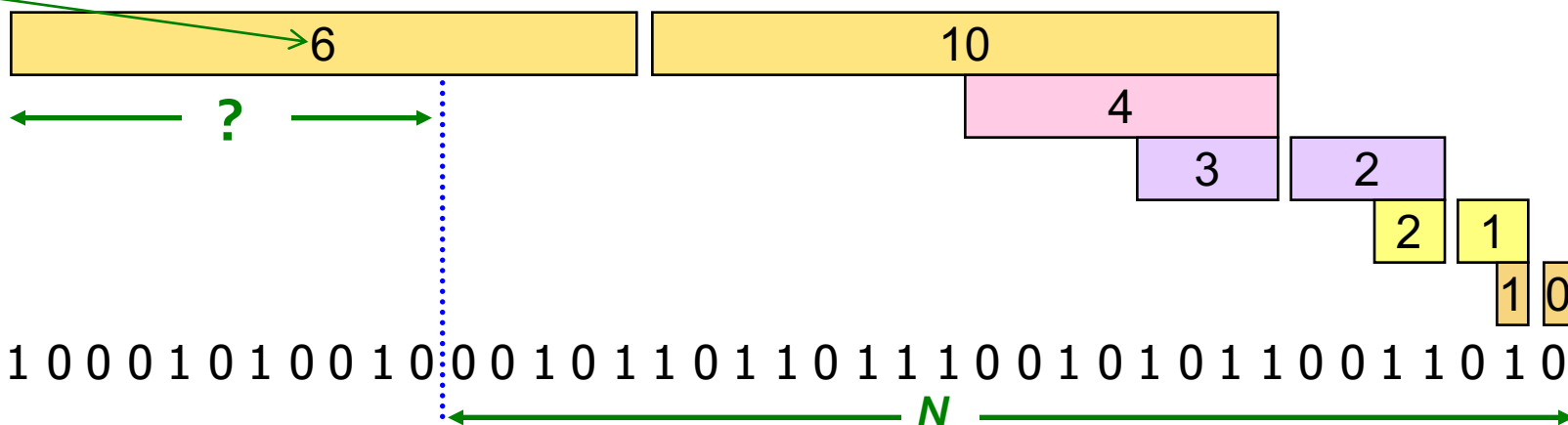
[Datar, Gionis, Indyk, Motwani]

- Name refers to the inventors:
 - Datar, Gionis, Indyk, and Motwani.
 - DGIM solution that **does not 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.

Idea: Exponential Windows

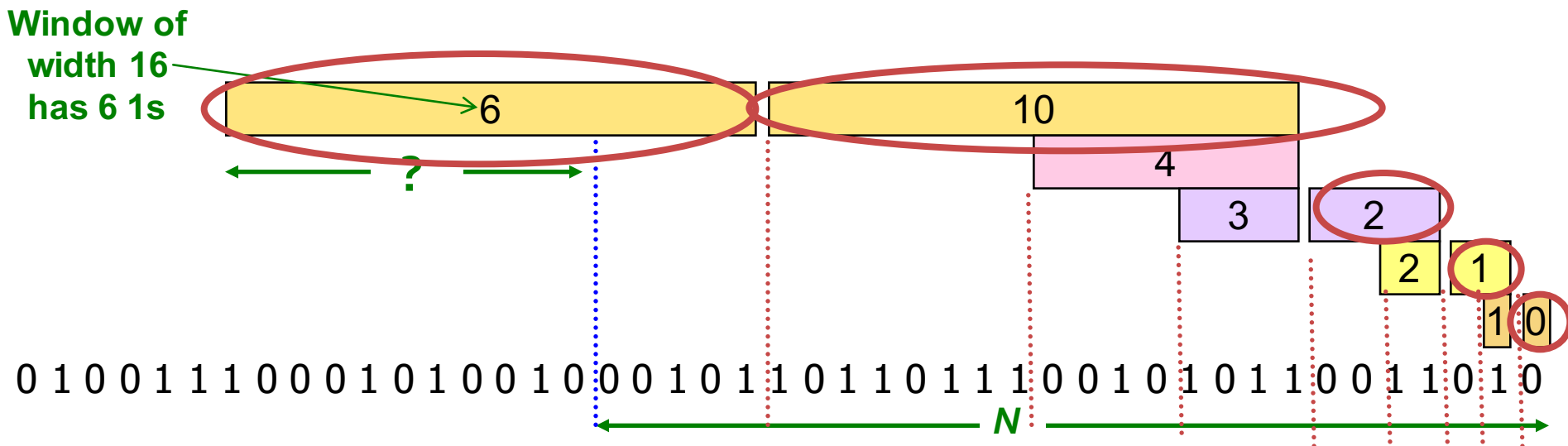
- **Solution that doesn't (quite) work:**
 - Summarize **exponentially increasing** regions of the stream, looking backward
 - Drop small regions if they begin at the same point as a larger region

Window of
width 16
has 6 1s



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

Example (count.)



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

$$6/16 \times 5 = 30/16 \sim 2$$

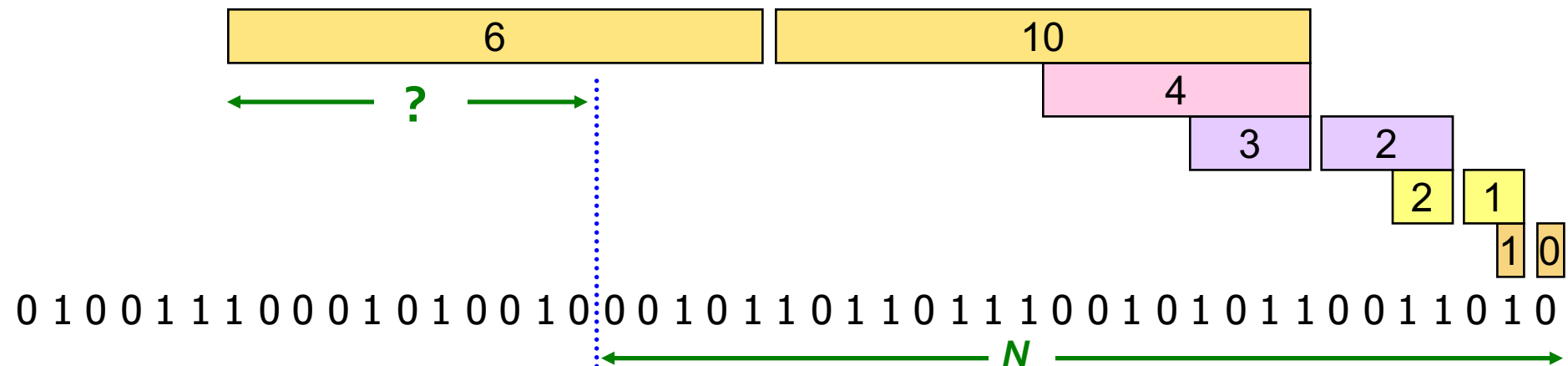
$$0 + 1 + 2 + 10 = 13 + 2 = 15$$

What's Good?

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

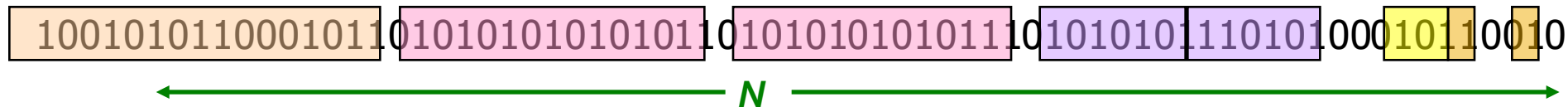
What's Not So Good?

- 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 **1s**:
 - Let the block *sizes* (number of **1s**) increase exponentially
- When there are few 1s in the window, block sizes stay small, so errors are small

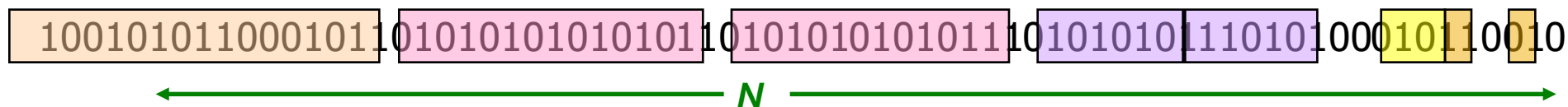


DGIM: Timestamps

- 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 $O(\log_2 N)$ bits.

DGIM: Buckets

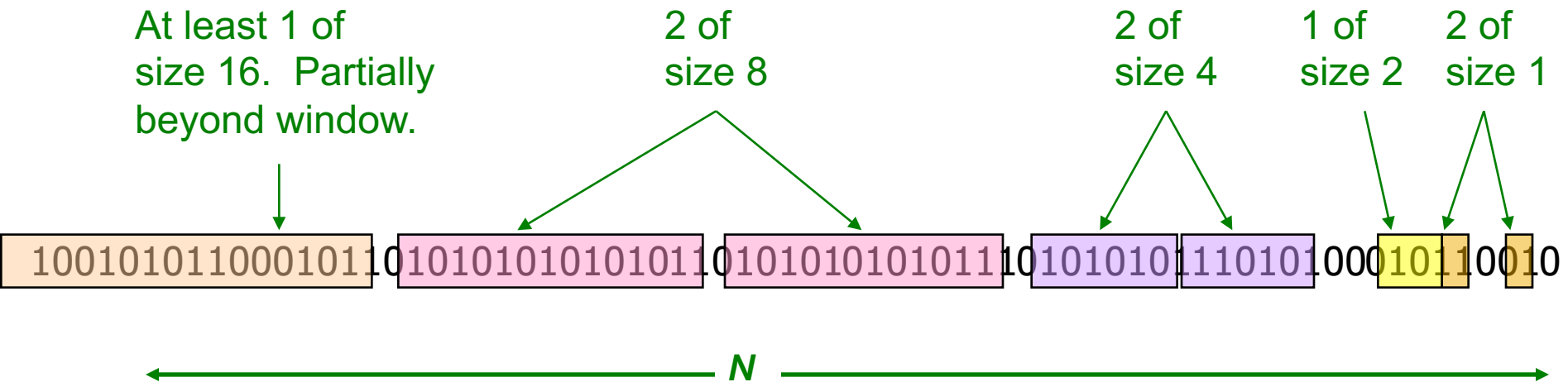
- A **bucket** in the DGIM method is a record consisting of:
 - (A) The timestamp of its end [$O(\log N)$ bits]
 - (B) The number of 1s between its beginning and end [$O(\log \log N)$ bits]
- **Constraint on buckets:**
Number of **1s** must be a power of 2
 - That explains the $O(\log \log N)$ in (B) 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:

1001010110001011010101010101011010101010101110101010111010101011101010100010110010

Bit of value 1 arrives

0010101100010110101010101010110101010101011101010101110101010111010101000101100101

Two orange buckets get merged into a yellow bucket

0010101100010110101010101010110101010101011101010101110101010111010101000101100101

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

01011000101101010101010101011010101010101110101010111010101000101100101101

Buckets get merged...

01011000101101010101010101011010101010101110101010111010101000101100101101

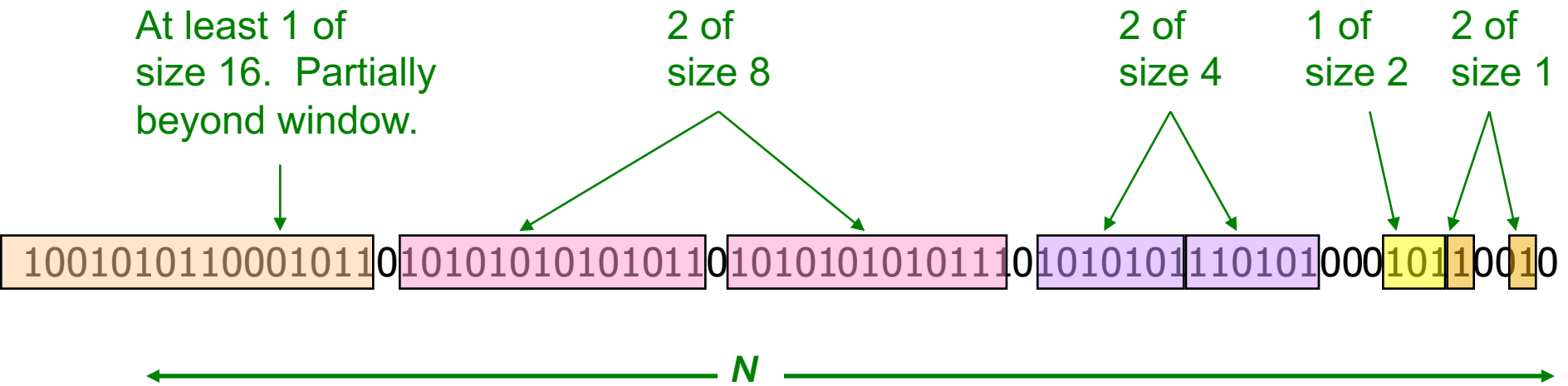
State of the buckets after merging

01011000101101010101010101011010101010101110101010111010101000101100101101

How to Query?

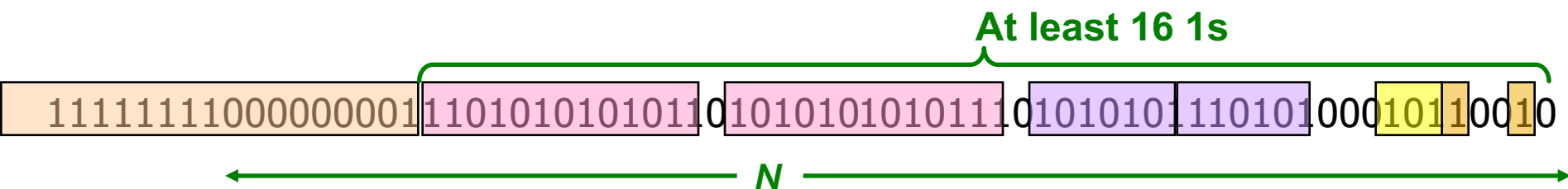
- To estimate the number of 1s in the most recent N bits:
 1. Sum the sizes of all buckets but the last
(note “size” means the number of 1s in the bucket)
 2. Add half the size of the last bucket
- **Remember:** We do not know how many 1s of the last bucket are still within the wanted window.

Example: Bucketized Stream



Error Bound: Proof

- Why is error 50%?
- 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 $1 + 2 + 4 + \dots + 2^{r-1} = 2^r - 1$
- Thus, error at most 50%



Further Reducing the Error

- Instead of maintaining **1** or **2** of each size bucket, we ($r > 2$) **allow either $r-1$ or r buckets**
 - 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**.

More algorithms for streams

- Sampling data **from a stream**
- Filtering a data stream: **Bloom filters**
- Counting distinct elements: Flajolet-Martin
- Estimating moments: AMS method.

Sampling from a data stream

- Method 1: sample a **fixed portion** of elements
 - e.g., 1/10
- Method 2: maintain a **fixed-size** sample.

Sampling from a Data Stream: Sampling a fixed proportion

- **As the stream grows the sample also gets bigger**

Sampling from a Data Stream

- Since **we can not store the entire stream**, one obvious approach is to store a **sample**
- **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** over a potentially infinite 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**.

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 $1/10^{\text{th}}$ 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.

Example: Unique search queries

- The length of the sample is **10% of the length** of the whole stream
- Suppose a query is **unique**
 - It has a **10% chance** of being in the sample
- Suppose a query **occurs exactly twice** in the stream
 - It has an **18% chance** of appearing exactly once in the sample.
 - $(1/10 \cdot 9/10) + (9/10 \cdot 1/10) = 0.18$
- And so on ... The fraction of unique queries in the stream is unpredictably large.

Problem with Naïve 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)
 - **Correct answer:** $d/(x+d)$
 - **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**
 - $18d/100 = ((1/10 \cdot 9/10) + (9/10 \cdot 1/10)) \cdot d$
 - d_j selected, d_j' not selected: $1/10 \cdot 9/10$
 - d_j not selected, d_j' selected: $9/10 \cdot 1/10$
 - **So the sample-based answer is**
$$\frac{\frac{x}{10} + \frac{d}{100} + \frac{18d}{100}}{\frac{x}{10} + \frac{d}{100} + \frac{18d}{100}} = \frac{d}{10x+19d} \neq d/(x+d)$$

Solution: Sample Users

Our mistake: we sampled based on the **position** in the stream, rather than **the value of the stream element**

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
- **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 from a Data Stream:

Sampling a fixed-size sample

- **As the stream grows, the sample is of fixed size**

Problem with fixed portion sample

- Sample size may grow too big when data stream in
 - Even 10% could be too big
- Idea: throw away some queries
- Key: do this consistently
 - remove all or none of occurrences of a query.

Controlling the sample size

- Put an upper bound on the sample size
 - Start out with 10%
- Solution:
- Hash queries to a large # of buckets, say 100
 - Take for the sample those elements hashing to buckets 0 through 9.
- When sample grows too big, throw away bucket 9
- Still too big, get rid of 8, and so on.

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 .

Filtering 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

- **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.

Filtering Stream Content

- Consider a **web crawler**
- It keeps a **list of all the URL's** it has found so far
- It assigns these URL's to any of a number of parallel tasks;
 - these tasks stream back the URL's they find in the links they discover on a page
- It needs to filter out those URL's it **has seen before.**

Role of the Bloom Filter

- A Bloom filter placed on the stream of URL's will declare that **certain URL's have been seen before**
- Others will be **declared new**, and **will be added** to the list of URL's that need to be crawled
- Unfortunately, the Bloom filter can have **false positives**
 - It can declare a **URL has been seen before** when it hasn't
 - But if it says **"never seen," then it is truly new (no False Negative)**.

How a Bloom Filter Works

- A **Bloom filter** is an **array of bits**, together with a number of **hash functions**
- The argument of each hash function is a **stream element**, and it returns a **position** in the array
- Initially, all bits are 0
- When input x arrives, we set to 1 the bits $h(x)$,
- for each hash function h .

Example: Bloom Filtering

- Use $N = 11$ bits for our filter
- Stream elements = integers
- Use two hash functions:
 - $h1(x) =$
 - Take **odd numbered** bits from the **right** in the **binary representation** of x
 - Treat it as an integer i
 - Result is $i \bmod 11$
 - $h2(x) =$ same, but take **even numbered** bits.

Example: Building the filter

$h_1(x)$ = **odd position** bits from the **right**

$h_2(x)$ = **even position**

Stream element

Convert Decimal to Binary

h_1

h_2

Filter

0 0 0 0 0 0 0 0 0 0 0

25 = 1 1 0 0 1

5

2

0 0 1 0 0 1 0 0 0 0 0

159 = 1 0 0 1 1 1 1 1

7

0

1 0 1 0 0 1 0 1 0 0 0

585 = 1 0 0 1 0 0 1 0 0 1

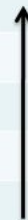
9

7

1 0 1 0 0 1 0 1 0 1 0

*Recall: Convert Decimal to Binary: eg: 25 to 11001

2	25	Remainder
2	12	1
2	6	0
2	3	0
2	1	1
	0	1



Bloom Filter Lookup

- Suppose element **y appears** in the stream, and we want to know **if we have seen y before**
- **Compute $h(y)$** for each hash function y
- If all the resulting **bit positions are 1**, say we have **seen y before** (false positive)
- If at least one of these **positions is 0**, say we have **not seen y before** (false negative).

Example: Lookup

- Suppose we have the same Bloom filter as before, and we have set the filter to 101**0**0**1**01010.
- Lookup element **y = 118** = **1110110** (binary).
- **h1(y)** = 14 modulo 11 = **3**.
- **h2(y)** = 5 modulo 11 = 5.
- Bit 5 is 1, but bit 3 is 0, so we are **sure y is not in the set**.

Performance of Bloom Filters

- **Probability of a false positive** depends on the **density of 1's** in the array and the **number of hash functions**
 - $= (\text{fraction of 1's}) \times \# \text{ of hash functions}$
- The number of 1's is approximately the number of elements inserted times the number of hash functions
 - But collisions lower that number slightly.

Analysis: Throwing Darts (1)

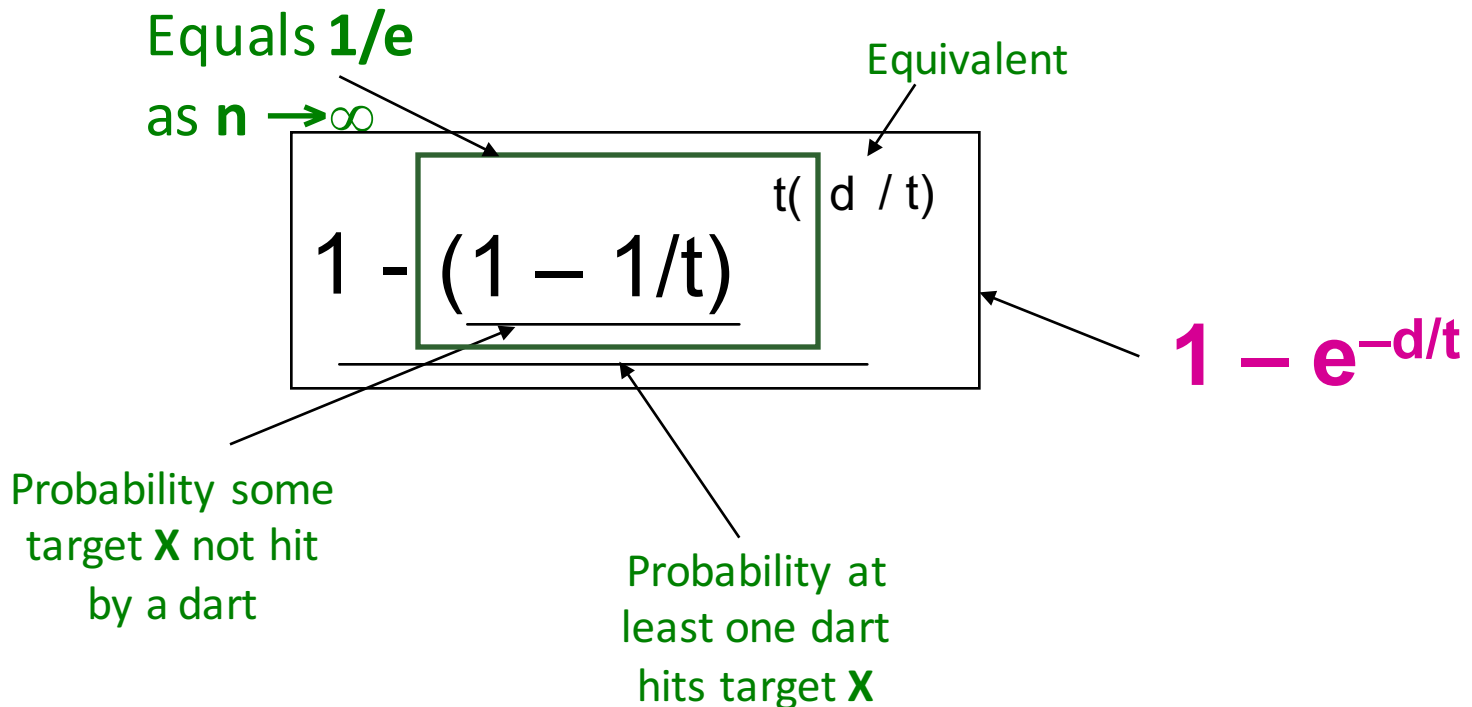
- Turning random bits from 0 to 1 is like throwing d darts at t targets, at random
- **More accurate analysis for the number of false positives**
- **Consider:** If we throw d darts into t equally likely targets, **what is the probability that a target gets at least one dart?**
- **In our case:**
 - **Targets** = bits/buckets
 - **Darts** = hash values of items.

Analysis: Throwing Darts (2)

- We have d darts, t targets
- **What is the probability that a target gets at least one dart?**
- Probability a given target is hit by a given dart
 $= 1/t$
- Probability none of d darts hit a given target is
 $= (1 - 1/t)^d$

Analysis: Throwing Darts (2)

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



Example: Throwing Darts

- **Fraction of 1s in the array B =**
= probability of false positive = $1 - e^{-d/t}$
- Example: Suppose we use an array of **1 billion bits**, **5 hash functions**, and **we insert 100 million elements**
- That is, **$t = 10^9$** , and **$d = 5 * 10^8$**
 - The **fraction of 0's** that remain will be $e^{-1/2} = 0.607$
 - **Density of 1's** = 0.393
- Probability of a false positive = $(0.393)^5 = 0.00937$.

Summary

- DBMS vs **Stream Management**
- Stream data processing and type of queries
- 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.
- Sampling data **from a stream**
 - Method 1: sample a **fixed portion** of elements
 - Method 2: maintain a **fixed-size** sample
- Filtering a data stream: **Bloom filters**