Chapter 8: Mining Data Streams

崔金华
电子邮箱: jhcui@hust.edu.cn
个人主页: https://csjhcui.github.io/

致谢:课件主要参考《Mining of Massive Datasets》(Second Edition) J. Leskovec, A. Rajaraman, J. Ullman的相关课件, 特此致谢!!!

Data Streams

华中科技大学 计算机科学与技术学院 School of Computer Science &Technology, HUST

- **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)

. . . 1, 5, 2, 7, 0, 9, 3
. . . a, r, v, t, y, h, b
. . . 0, 0, 1, 0, 1, 1, 0
time

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 Weibo, Twitter, Facebook

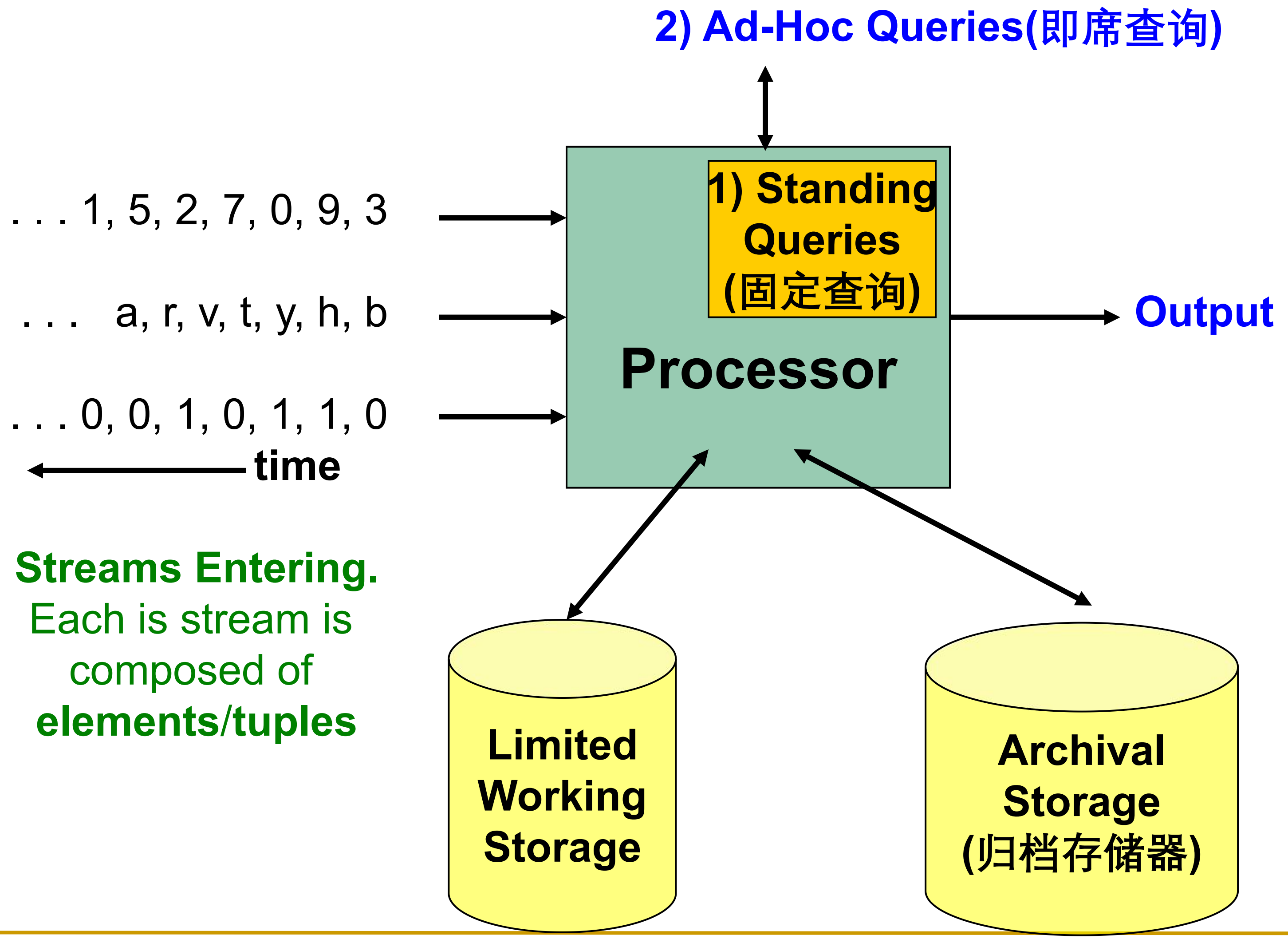
The Stream Model

- Input **elements** enter at a rapid rate, at one or more input ports (i.e., **streams**)
  - **We call elements of the stream tuples**

- **The system cannot store the entire stream accessibly**

- **Q: How do you make critical calculations about the stream using a limited amount of (secondary) memory?**

General Stream Processing Model



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

General Stream Processing Model

流处理的若干限制:

- 流元素的分发速度通常很快. 必须对元素进行实时处理, 否则可能永远失去处理它们的机会.
- 此外, 即使数据流很慢, 也可能存在多个这样的数据流. 所有数据流的内存需求加载一起可能超过内存的可用容量.

流处理的一般化结论:

- 通常情况下, 获得问题的近似解比精确解要高效得多.
- 一系列与哈希相关的技术被证明十分有用. 一般而言, 为了产生与精确解相当接近的近似解, 上述技术将十分有用的随机性引入算法行为中.

Problems on Data Streams

华中科技大学 计算机科学与技术学院 School of Computer Science &Technology, HUST

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

➢Sampling data from a stream
- Construct a random sample

➢Queries over sliding windows
- Number of items of type ***x*** in the last ***k*** elements of the stream

➢Filtering a data stream
- Select elements with property ***x*** from the stream

➢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
- 矩估计是独立流元素技术推广为更一般的问题, 不同流元素出现频率分布的计算。

➢Finding frequent recent elements
- Consider exponentially decaying windows, weighting the recent elements more heavily

Section 8.1: Sampling from a Data Stream

Sampling from a Data Stream

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

- **Two different cases:**
  - **(1) Case 1:** Sample a **fixed proportion** of elements in the stream (say 10 in 100)
    - As the stream grows the sample also gets bigger
  - **(2)Case 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
    - Q: What is the property of the sample we want to maintain?
    - A: For all time steps ***k***, each of elements seen so far has equal prob. of being sampled

Sampling a Fixed Proportion

- **Sampling case 1: Sampling fixed proportion**

- **Scenario:** Search engine query stream
  - **Stream of tuples:** (user, query, time)
  - **Answer questions such as: What fraction of the typical user’s queries were repeated over the past month?(过去一个月中典型用户所提交的重复查询的比率是多少？)**
  - 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

- **Another simple question: 用户提交的平均重复查询的比例是多少?**
  - Suppose each user issues $\boldsymbol{x}$ queries once and $\boldsymbol{d}$ queries twice (total of $\boldsymbol{x}+2\boldsymbol{d}$ queries).
  - Then correct answer: $\boldsymbol{d}/(\boldsymbol{x}+\boldsymbol{d})$
  - **Naïve solution: we keep 10% of the queries**
    - 抽样是对查询出现的抽样，而非按照查询本身抽样. 因此, 在原始$x+2d$个查询出现中, 最终会抽样出$(x+2d)$/10个查询出现样本.
    - 原来出现两次的查询仍然在样本中出现两次的概率为$d\times 1/10\times 1/10$=$d/100$
    - 原来出现两次的查询, 在抽样样本中占据2d/10次出现, 因此原来d个出现两次的查询在样本中仅出现一次的概率为2d/10-2d/100=18d/100
    - 所以, 该方案下的答案是$(\boldsymbol{d}/100)/(\boldsymbol{x}/10+\boldsymbol{d}/100+18\boldsymbol{d}/100)=\boldsymbol{d}/(10\boldsymbol{x}+19\boldsymbol{d})$

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

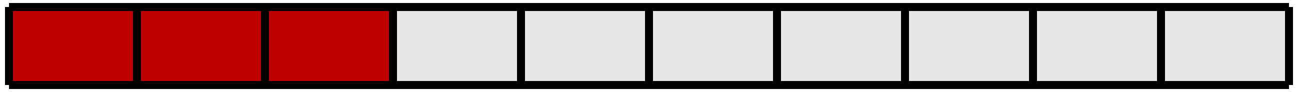
Generalized Solution

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

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***



e.g., 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

- **Sampling case 2: Fixed-size sample**
  - As the stream grows, the sample is of fixed size

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

Maintaining a Fixed-Size Sample

- **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. 2/5
  - At n= 7, each of the first 7 tuples is included in the sample S with equal prob. 2/7

- Impractical naïve solution: Store all the n tuples seen so far and out of them pick s at random

Solution: Fixed Size Sample

❑Algorithm (a.k.a. Reservoir Sampling, 蓄水池抽样算法)

- 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** (数学归纳法):
  - 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

华中科技大学
计算机科学与技术学院
School of Computer Science &Technology, HUST

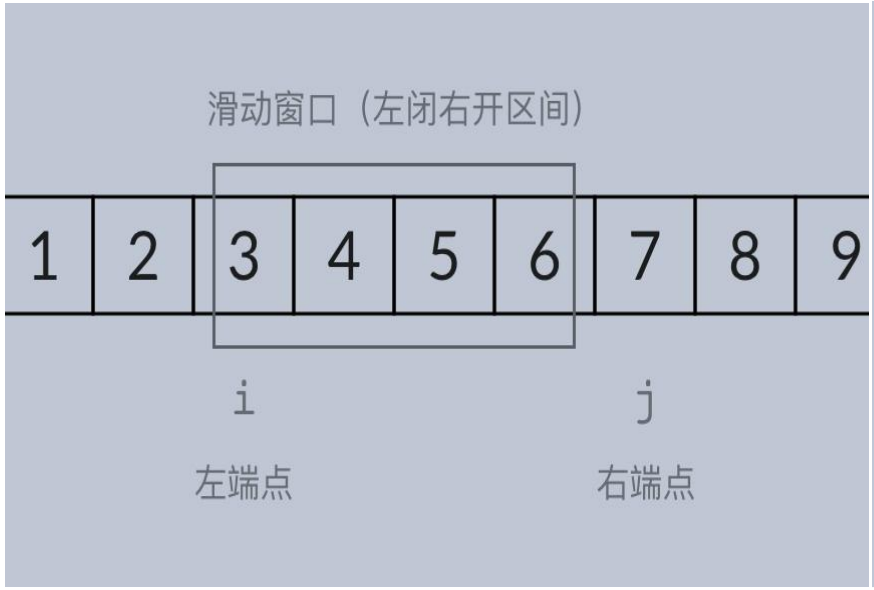
- **Inductive hypothesis:** After $\boldsymbol{n}$ elements, the sample $\boldsymbol{S}$ contains each element seen so far with prob. $\boldsymbol{s/n}$
- **Now element $n+1$ arrives**
- **Inductive step:** For elements already in $\boldsymbol{S}$, probability that the algorithm keeps it in $\boldsymbol{S}$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

$$\underbrace{\left(1-\frac{s}{n+1}\right)}_{\text{第n+1个元素被丢弃}}+\underbrace{\left(\frac{s}{n+1}\right)}_{\text{第n+1个元素没有被丢弃}}\underbrace{\left(\frac{s-1}{s}\right)}_{\text{样本中被保留的概率}}=\frac{n}{n+1}$$

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

Problems on Data Streams

- **Types of queries one wants on answer on a data stream:**
  - **Sampling data from a stream**
    - Construct a random sample
  - **Queries over sliding windows (e.g., DGIM)**
    - Number of items of type ***x*** in the last ***k*** elements of the stream
  - **Filtering a data stream**
    - Select elements with property *x* from the stream
  - **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 recent elements**
    - Consider exponentially decaying windows, weighting the recent elements more heavily



Section 8.2: 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)**
  - **Alternative:** elements received within a time interval ***T***
- **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 **0**s and **1**s
- Be prepared to answer queries of the form: **How many 1s are in the last *k* bits?** where $k \le N$

Obvious naïve 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 →

- But, remember you can not get an exact answer without storing the entire window.

Counting Bits

- Real Problem: **What if we cannot afford to store *N* bits?**
  - E.g., we are processing 1 billion(10亿) 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?**

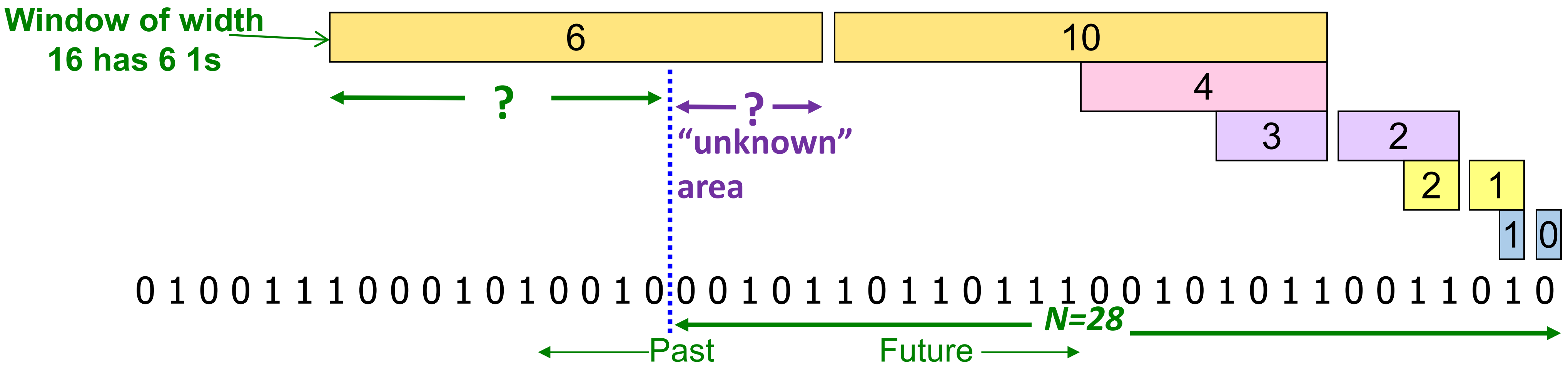
$\longleftarrow$ ***N*** $\longrightarrow$

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

$\longleftarrow$ Past   Future $\longrightarrow$

- 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? $N \cdot S/(S+Z)$
- **But, what if stream is non-uniform?**
  - What if distribution changes over time?

- **A strawman algorithm (稻草人算法) 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



How many 1s are in the last 28 bits?

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?

华中科技大学 计算机科学与技术学院 School of Computer Science &Technology，HUST

- ❑What is good in the strawman algorithm?
- ❑1、Stores only **O(log²*N*)** bits

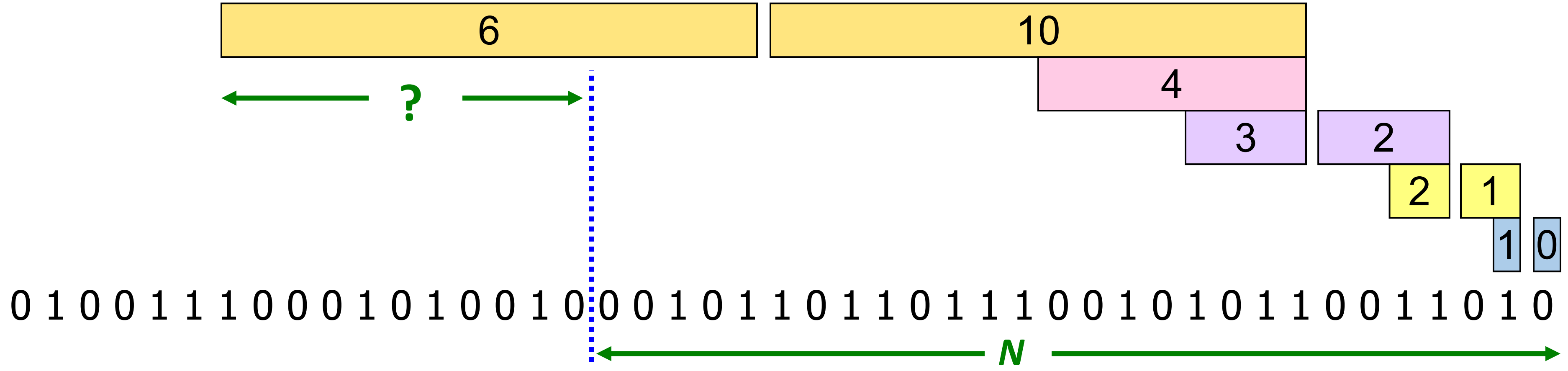
  Note: $\log^2 N$ means $(\log N)*(\log N)$

  - ➢一个region所需存储空间*region个数 =log N*2 log N (since 2个 2^0, 2个2^1,…, 2个$2^{\log N}$)

- ❑2、Easy update as more bits enter
- ❑3、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

- **DGIM**(**Datar, Gionis, Indyk, Motwani**) solution: Instead of summarizing fixed-length blocks, summarize blocks with specific number of **1s**:
  - Let the block ***sizes*** (number of **1s**) increase exponentially

1001010110001011 0101010101010101 10 1010101010101110 1010101 110101 000 101 1 001 0

$\longleftarrow$ *N* $\longrightarrow$

- DGIM solution that does not assume uniformity, and it stores $\boldsymbol{O}(\log^2 \boldsymbol{N})$ bits per stream

DGIM Method

- When there are few **1s** in the window, block sizes stay small, so errors are small
- DGIM gives approximate answer, never off by more than 50%
  - Error factor can be reduced to any fraction > 0, but with more complicated algorithm and proportionally more stored bits

DGIM: Timestamps

- Each bit in the stream has a ***timestamp*（时间戳）**，starting **1**, **2,** …
- Record timestamps modulo ***N*** (**the window size**), so we can represent any **relevant** timestamp in $\boldsymbol{O(log_2N)}$ bits

DGIM: Buckets

- A ***bucket*** (**桶**, 注意这儿不是哈希中的桶) in the DGIM method is a record consisting of:
  - **(A)** The **timestamp** of its end **[O(log *N*) bits]**
  - **(B)** The number of **1s (**called **size of the bucket, 桶的大小)** between its beginning and end **[O(log log *N*) bits]**

10010101100010110101010101010110101010101011101010101110101000101100 10

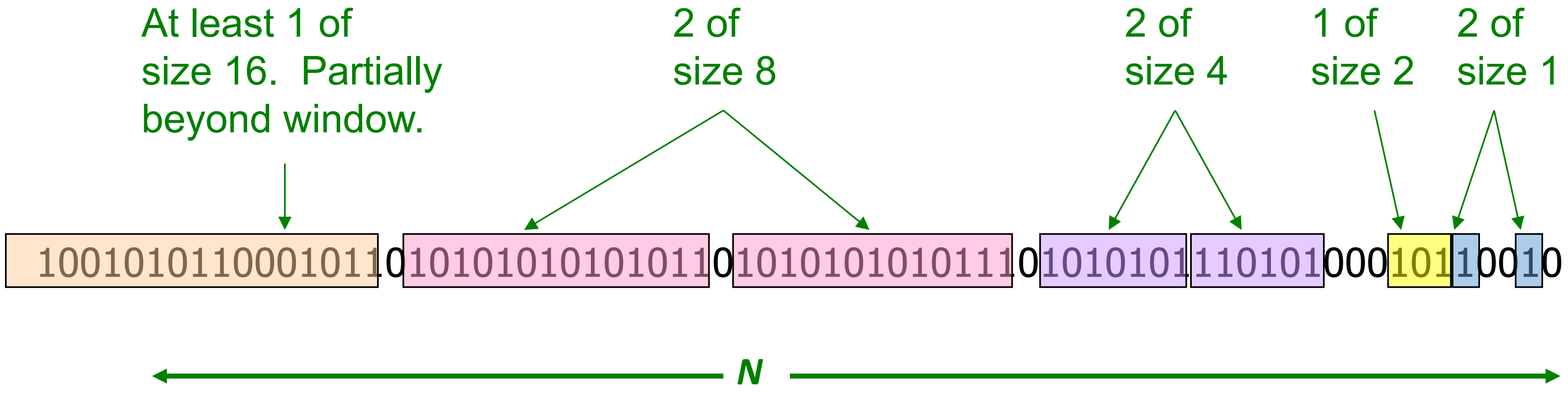
$\longleftarrow$ ***N*** $\longrightarrow$

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

DGIM: Buckets

- **Constraint on buckets (续):**
  - 桶最右边的位置总是为1
  - 每个1的位置都在某个桶中
  - 一个位置只能属于一个桶
  - 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.
  - 桶之间存在一些0是允许的

Example: Bucketized Stream



Properties of buckets that are maintained:

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

Updating Buckets

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

10010101100010110101010101010101101010101010111101010101110101000101100 10

Bit of value 1 arrives

001010110001011010101010101010110101010101011110101010111010100010110010 1

Two old blue buckets get merged into a yellow bucket

001010110001011010101010101010110101010101011110101010111010100010110010 1

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

01011000101101010101010101011010101010101111010101011101010001011001011 01

Buckets get merged…

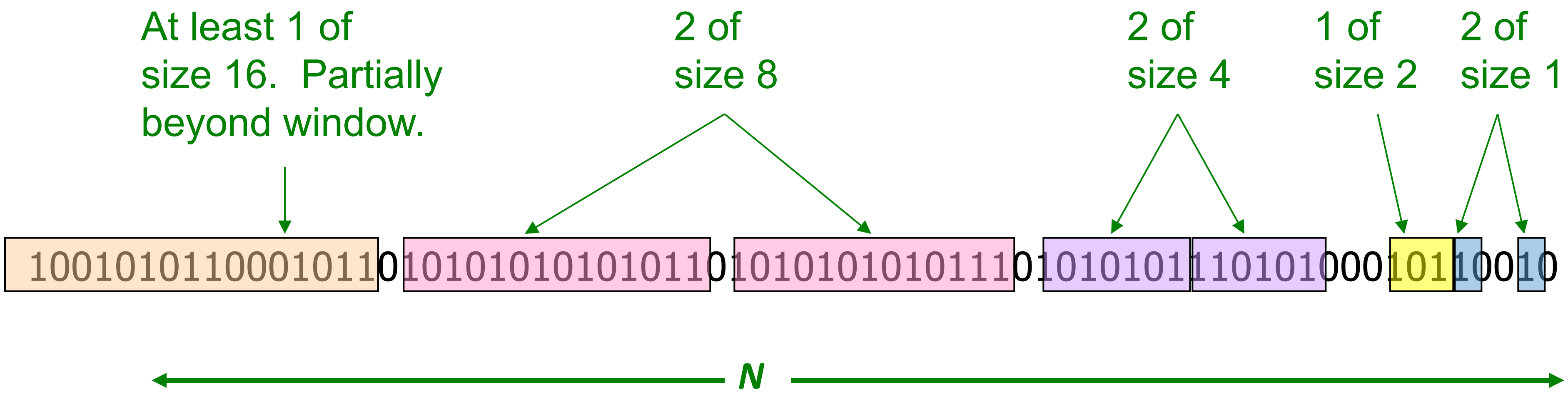
01011000101101010101010101011010101010101111010101011101010001011001011 01

State of the buckets after merging

01011000101101010101010101011010101010101111010101011101010001011001011 01

How to Query?

- **To estimate the number of 1s in the most recent *N* bits:**
  - **Sum** the **sizes of all buckets** but the last (note "size" means the number of 1s in the bucket).
  - Add **half the size** of the last bucket (error at most **50%**).
- **Remember:** We do not know how many **1s** of the last bucket are still within the wanted window.



Proof: Error Bound

华中科技大学 计算机科学与技术学院 School of Computer Science &Technology，HUST

❑ **Why is error 50%? Let’s prove it!**

At least 16 1s

11111111100000000011101010101011010101010101110101010111010100010110010

N

- Suppose the last bucket *b* has size $\mathbf{2}^{r}$. 分两种情况讨论:
- **Case A**: 估计值大于真实值. 最坏情况下, 桶*b*中只有最右边一位在查询范围内, 且所有比该桶小的桶都仅有一个桶. 那么, 预估值为$\mathbf{2}^{r-1}+\mathbf{2}^{r-1}+\mathbf{2}^{r-2}+\ldots+\mathbf{4}+\mathbf{2}+\mathbf{1}=\mathbf{2}^{r}-1+\mathbf{2}^{r-1}$, 真实值为$1+\mathbf{2}^{r-1}+\mathbf{2}^{r-2}+\ldots+\mathbf{4}+\mathbf{2}+\mathbf{1}=\mathbf{2}^{r}$. 那么估计值最多比真实值大50%.
- **Case B**: 估计值小于真实值. 最坏情况下, 桶*b*中所有的1都在查询范围内. 那么桶*b*的预估值根据要求为$\mathbf{2}^{r}/\mathbf{2}$, 桶*b*的真实值为$\mathbf{2}^{r}$, 错误最多为真实值的50%.

Further Reducing the Error in DGIM

- (扩展优化) 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

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

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

Extensions (1)

- 另一种查询应答: Can we use the same trick to answer queries **How many 1s in the last *k*?** where ***k*** < ***N***?
  - **A:** Find earliest bucket **B** that at overlaps with ***k***. Then, number of **1s** is the **sum of sizes of more recent buckets + ½ size of B**

1001010110001011 0 1010101010101011 0 1010101010101111 0 1010101 110101 000 101 1 00 1 0

k

Extensions (2)

- 另一种流场景的处理: Can we handle the case where the stream is **not bits, but integers** (e.g., positive integers), and we want the **sum of the last *k* elements**?
  - **E.g.,** Amazon: Avg. price of last ***k*** sales

Extensions (2)

Solution 1: If you know all have at most *m* bits

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

Solution 2: Use buckets to keep partial sums

- Sum of elements in size b bucket is at most 2^b

2 5 7 1 3 8 4 6 7 9 1 3 7 $\boxed{6\ 5}\boxed{3\ 5}\boxed{7\ 1}\boxed{3}\boxed{3}\boxed{1}\boxed{2}\boxed{2}\boxed{6}$

2 5 7 1 3 8 4 6 7 9 1 3 7 6 5 $\boxed{3\ 5\ 7\ 1}\boxed{3\ 3}\boxed{1\ 2}\boxed{2}\boxed{6}\boxed{\mathbf{3}}$

2 5 7 1 3 8 4 6 7 9 1 3 7 6 5 $\boxed{3\ 5\ 7\ 1}\boxed{3\ 3}\boxed{1\ 2}\boxed{2}\boxed{6}\boxed{3}\boxed{\mathbf{2}}$

2 5 7 1 3 8 4 6 7 9 1 3 7 6 5 $\boxed{3\ 5\ 7\ 1}\boxed{3\ 3}\boxed{1\ 2}\boxed{2}\boxed{6}\boxed{3}\boxed{2}\boxed{\mathbf{5}}$

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

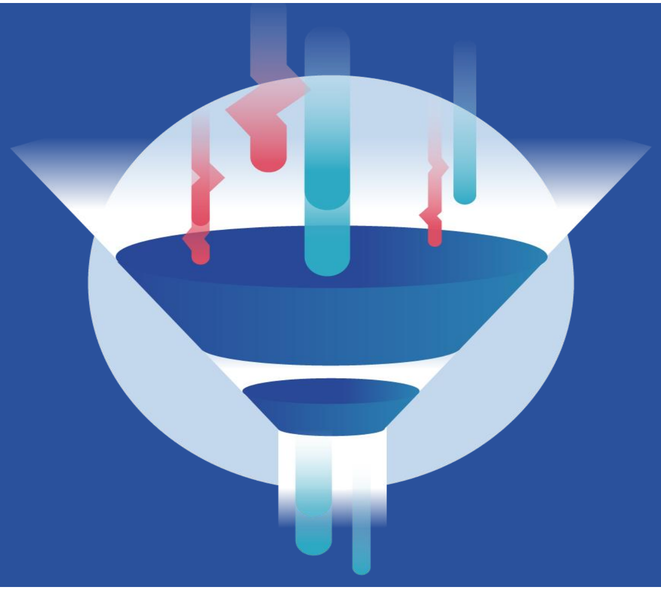
Bucket sizes:

16	8	4	2	1

Problems on Data Streams

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

- **Sampling data from a stream**
  - Construct a random sample
- **Queries over sliding windows**
  - Number of items of type ***x*** in the last ***k*** elements of the stream
- **Filtering a data stream (e.g., Bloom filter)**
  - 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 recent elements**
  - Consider exponentially decaying windows, weighting the recent elements more heavily



Section 8.3: 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 1: Email spam filtering (垃圾邮件过滤)
- We know 1 billion "good" email addresses
- If an email comes from one of these, it is **NOT** spam

❑Example 2: 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

❑Example 3: Web crawler
- It keeps, centrally, 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.

Bloom filter

- A **Bloom filter (布隆过滤器)** is an array of n bits (or called bit-array, 位数组), together with a number of k hash functions.
  - Initially, all n bits are 0.
  - A collection of hash functions $h_1, h_2, \ldots, h_k$. Each hash function maps key values to n buckets (corresponding to the n bits of the bit-array).
  - A set **S** of m key values (m个键值组成的集合S).

- The purpose of the Bloom filter is to allow through all stream elements whose keys are in **S**, while rejecting most of the stream elements whose keys are not in **S**.

Bloom filter

- A Bloom filter for web crawler :
  - placed on the stream of URL's will declare that certain URL's have been seen before.
  - Others will be declare new, and will be added to the list of URL's that need to be crawled.

Example

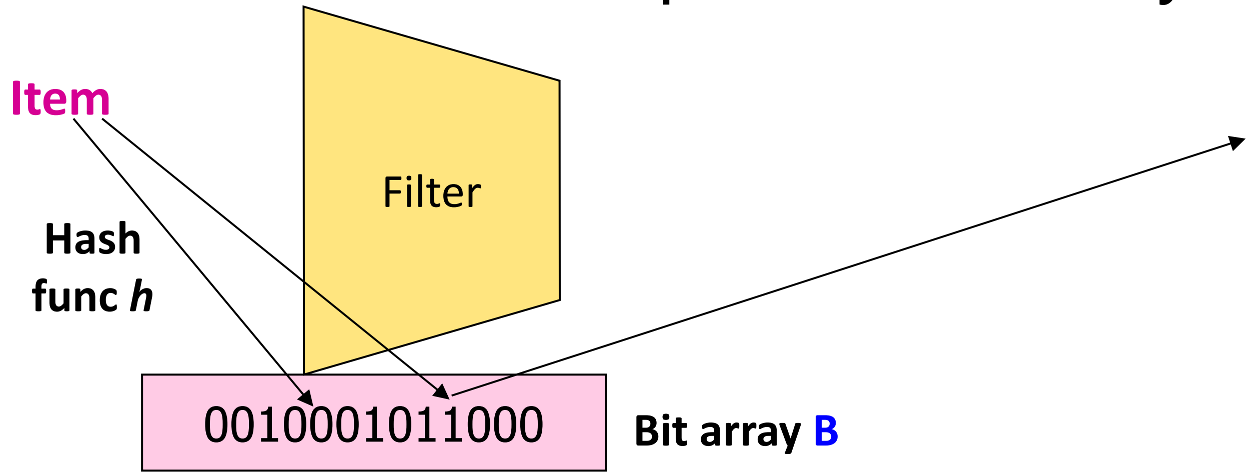
- Example: Use N=11 bits for our Bloom filter. Stream elements = integers. Use two hash functions:
- $h_1(x)$=
  - Take odd(奇数)-numbered bits from the right in the binary representation of x.
  - Treat it as an integer i.
  - Result is i modulo 11.
- $h_2(x)$=
  - Same, but takes even (偶数)-numbered bits.

Example –Continued

Stream element	h_1 奇数位置(从右边开始)组成的二进制数转换成十进制数据	h_2 偶数...	Filter contents 00000000000 $N = 11$ 哈希值对应的位置修改为1(初始全为0. 从左边数 0, 1, 2, ..., N-1)
25=11001	5	2	00100100000
159=10011111	7	0	10100101000
585=1001001001	9	7	10100101010

Bloom filter lookup

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



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***.

Example

- **Example**: Suppose we have the same Bloom filter as before, and we have set the filter to 10100101010. Lookup element y=118=1110110(binary).
- Answer: Compute $h(y)$ for each hash function y:
  - $h_1(y)$=14 modulo 11=3.
  - $h_2(y)$=5 modulo 11=5.
  - As bit 5 is 1, but bit 3 is 0 (10100101010), so we are sure y is not in the set.

Performance of Bloom filters

- Unfortunately, the Bloom filter can have false positives (伪正例, 误判, 假阳).
  - It can declare a element has been seen before when it hasn't.
  - 如果某个元素的键值在S中出现, 那么该元素肯定能够通过布隆过滤器.
  - If it say "never seen," then it is truly new.

- **Probability of a false positive (假阳率):** Depends on the density of 1's in the array and the number of hash functions

$$=(\boldsymbol{fraction\ of\ 1's})^{\#\ \boldsymbol{of\ hash\ functions}}$$

False positive(伪正例):某些元素不在其中, 但它被认为是在其中

Proof: Throwing darts

- More accurate analysis for the probability of a false positive (假阳率) in Bloom filter, 使用**throwing darts (飞镖投掷)飞镖投掷模型**来模拟布隆过滤: Turning random bits from 0 to 1 is like throwing m darts(飞镖) at n targets (靶位), at random.

- Q: How many targets are hit by at least one dart? Fraction of 1s in the bit array? (多少个靶位至少被投中一次?)
  - Probability a given target is hit by a given dart= $1/n$
  - Probability none of m darts hit a given target=$(1-1/n)^m$. Rewrite as $(1-1/n)^{n(\frac{m}{n})} \cong \left(\frac{1}{e}\right)^{\frac{m}{n}} = e^{-m/n}$
  - So, probability that a target gets at least one dart=1-$e^{-m/n}$

Note:k 个哈希函数, k*� darts(飞镖个数, 插入元素个数. 其中m是集合S中元素个数) , � targets (靶位, the array of � bits)

Example

- **Example**: Suppose we use an array of 1 billion (10亿) bits, 5 hash functions, and we insert 500 million (5亿) elements. That is n=10^9, and m=$5*10^8$. What is probability of the false positive?
- Answer:
  - Probability that a target gets at least one dart=1-$e^{-m/n}$=1-0.607=0.393
  - Fraction of 1s in the bit array under 5 hash=1-$e^{-km/n}$=1-0.082=0.918
  - Probability of the false positive =$(0.918)^5$ =0.652.
- Summary: **假阳率=$(1-e^{-km/n})^k$**

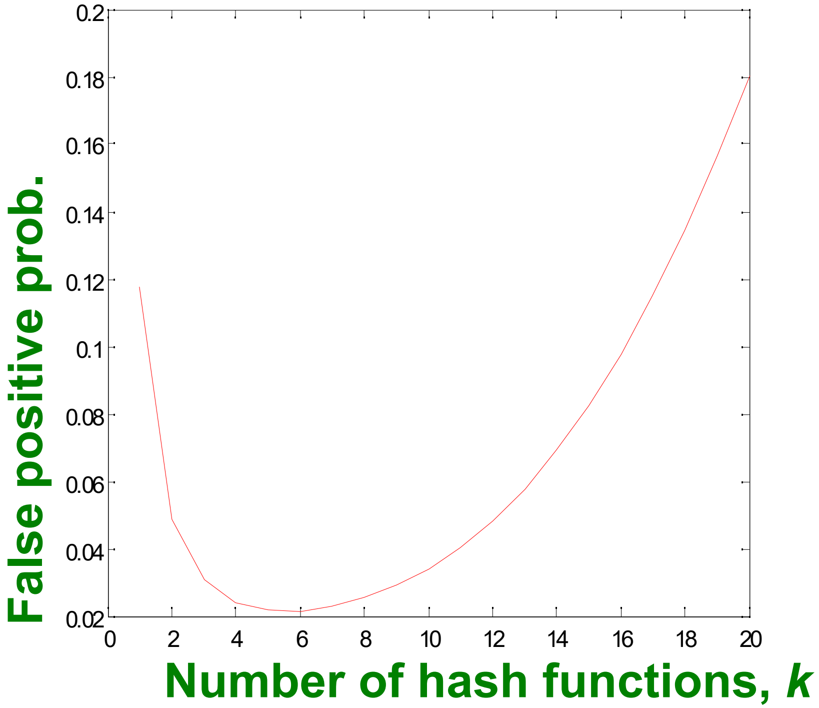
Note:k 个哈希函数, k*� darts(飞镖个数, 插入元素个数. 其中m是集合S中元素个数) , � targets (靶位, the array of � bits)

Bloom Filter – Analysis

- m = 1 billion(10亿), n = 8 billion(80亿)
  - k = 1: $(1 - e^{-1/8})$ = **0.1175**
  - k = 2: $(1 - e^{-1/4})^2$ = **0.0493**

假阳率=$(1-e^{-km/n})^k$

- **What happens as we keep increasing *k*?**



- **"Optimal" value of** $k = \frac{n}{m} * ln(2)$
  - **In our case:** Optimal **k = 8 ln(2) = 5.54 ≈ 6**
    - **Error at k = 6**: $(1 - e^{-3/4})^6$ = **0.0216**

Bloom Filter: 小结

- **Bloom filters use an array of � bits, together with a number of � hash functions. Bloom filters use limited memory, guarantee no false negatives 伪反例 (but with false positives 伪正例)**
  - Great for pre-processing before more expensive checks
- **Suitable for hardware implementation**
  - Hash function computations can be parallelized

Problems on Data Streams

华中科技大学 计算机科学与技术学院 School of Computer Science &Technology, HUST

- **Types of queries one wants on answer on a data stream:**
  - **Sampling data from a stream**
    - Construct a random sample
  - **Queries over sliding windows**
    - Number of items of type ***x*** in the last ***k*** elements of the stream
  - **Filtering a data stream**
    - Select elements with property ***x*** from the stream
  - **Counting distinct elements (e.g., Flajolet-Martin, CMS)**
    - Number of distinct elements in the last ***k*** elements of the stream
  - **Estimating moments**
    - Estimate avg./std. dev. of last *k* elements
  - **Finding frequent recent elements**
    - Consider exponentially decaying windows, weighting the recent elements more heavily

Section 8.4: Counting Distinct Elements

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

- **Applications 1: 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?)

- **Applications 2: How many different Web pages does each customer request in a week?**

- **Applications 3: 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

- 将流元素哈希到一个足够长的位串, 对独立元素个数进行预估. 当位串足够长, 哈希函数(实际上会选择多个不同的哈希函数)的可能结果数目大于全集中的元素个数.
- 哈希函数的性质: 相同元素的哈希值也相同.

- **Flajolet-Martin Approach (FM-sketch, FM算法)**基本思想:
  - 流中看到的不同元素越多, 那么我们看到的不同哈希值也会越多.
  - 看到的不同哈希值越多的同时, 也越可能看到其中有个值变得”异常”. 使用了一个具体的异常性质是该值后面会以多个0结束.

Flajolet-Martin Approach

- **FM算法**:
- Pick a hash function h that maps each of the N elements to at least $\log_2 N$ bits.
- **尾长**: For each stream element a, let $r(a)$ be the number of trailing **0s** in $h(a)$ (流元素a的哈希值位串的尾部0数目叫做尾长)
  - $r(a)$ = 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
- **So, estimated number of distinct elements $N' = 2^R$**

Flajolet-Martin Approach Example

- 例: 给定流数据序列$\{e_1, e_2, e_3, e_2\}$(因此独立元素数目N=3). 假设给定哈希函数$h(e)$: $h(e_1)=2=(0010)_2$; $h(e_2)=8=(1000)_2$; $h(e_3)$=10=$(1010)_2$. 根据FM算法, 预估流数据中独立元素数目N′为多少?
- 解:
  - 将$\boldsymbol{R}$初始化为0. 首先对流数据序列中第1项流元素e_1, $r(e_1)=1>\boldsymbol{R}$, 更新$\boldsymbol{R}$= 1; 对于序列中第2项e_2, $r(e_2)=3>\boldsymbol{R}$, 更新$\boldsymbol{R}$= 3; 对于序列中第3项e_3, $r(e_3)=1\le\boldsymbol{R}$, 不更新$\boldsymbol{R}$; 对于序列中第4项$e_2$, $r(e_2)=3\le\boldsymbol{R}$, 不更新$\boldsymbol{R}$. 估计独立元素数目为$\mathrm{N}'=2^R=2^3=8$.

在该例中实际值N=3, 估计值N′=8, 误差较大. 因此, 在实际应用中为了减小误差提高精度, 通常采用一系列的哈希函数$h_1(a)$, $h_2(a)$, $h_3(a)$, …计算一系列的独立元素预估值N'_1, N'_2, N'_3……, 最后进行特定综合统计得到最终的估计值.

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 $\log_2 N$ 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 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:

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

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 **$m/2^r \to 0$**
    - So, the probability of finding a tail of length *r* tends to **0**
  - If $m >> 2^r$, then prob. tends to **0**
    - $(1-2^{-r})^m \approx e^{-m2^{-r}} = 0$ as **$m/2^r \to \infty$**
    - So, the probability of finding a tail of length *r* tends to **1**
- **Thus, 2^R will almost always be around *m*!**

Why It Doesn' t Work

- **E[2^R] is actually infinite**
  - Probability halves when $R \rightarrow R+1$, but value doubles

- **Workaround involves using many hash functions h_i and getting many samples of R_i**

- **How are samples R_i combined?**
  - **Average?** What if one very large value 2^{R_i}?
  - **Median?** All estimates are a power of **2**
  - **Solution:将哈希函数分成多个小组;每个组内求平均值; 然后在所有平均值中取中位数**

Example

- 例: 对于序列S, 使用3*4 = 12个互不相同的哈希函数$h(e)$, 分成3组, 每组4个哈希函数, 使用12个$h(e)$估算出12个估计值N′: 第1组的4个估计值为<2, 2, 4, 4>; 第2组的4个估计值为<8, 2, 2, 2>; 第3组的4个估计值为<2, 8, 8, 2>. 求最终的估计值.
- 解:
  - 第1组的4个估计值的算术平均值为(2+2+4+4)/4=3;
  - 第2组的4个估计值的算术平均值为(8+2+2+2)/4=3.5;
  - 第3组的4个估计值的算术平均值为(2+8+8+2)/4=5;
  - 然后3个组的估计值分别为<3, 3.5, 5>，那么中位数为3.5；因此3.5 ≈ 4即为最终的估计值.

补充: Count-Min Sketch

- You want to count elements. But, you don't need exact results.

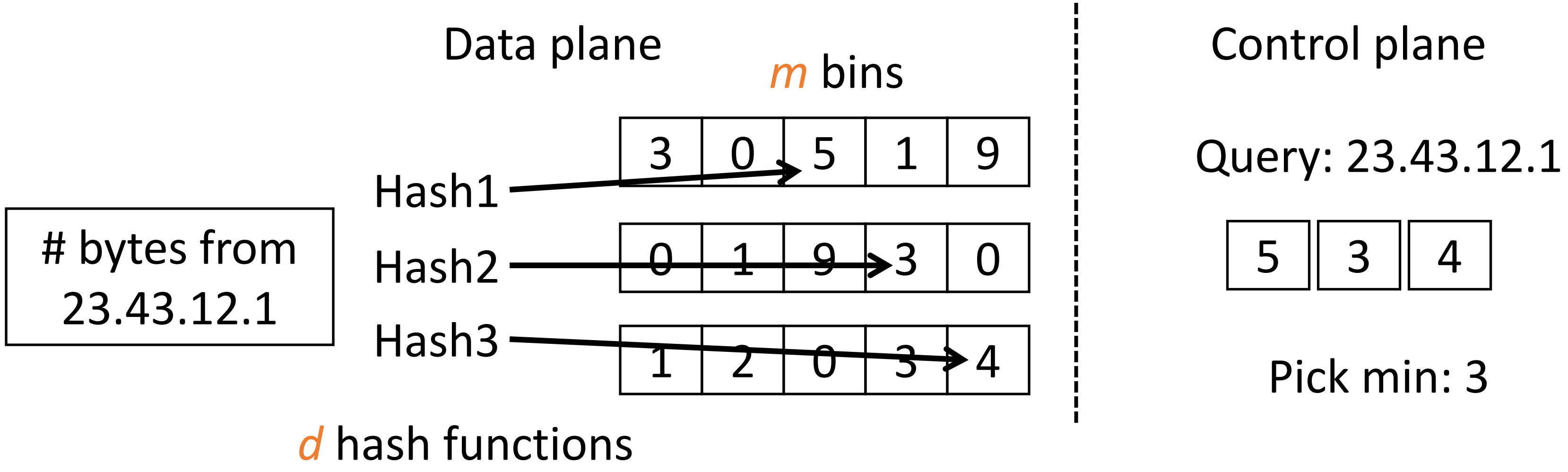
- Initial assumption: Create an integer array of length x initially filled with 0s. Each incoming element gets mapped to a number between 0 and x. The corresponding counter in the array gets incremented.
- To query an element's count, simply return the integer value at it's position. You are completely right: There will be collisions!

Count-Min Sketch

- 解决方案: Use multiple arrays with different hash functions to compute the index.
- When queried, return the minimum of the numbers the arrays. → **Count-Min Sketch算法 (简写CMS)**
  - It was first introduced in 2005 by Graham Cormode and S. Muthu Muthukrishnan, and has since become a popular algorithm for big data analytics.
  - CMS用于大规模数据流中的频率估计问题, 如计算一个元素在数据集中出现的次数. 在数据大小非常大时，通过牺牲准确性提高效率的算法.

Count-Min Sketch

❑Example:



Count-Min Sketch

- Only over-estimates, never under-estimates the true count.
- 优势: CMS has a constant memory and time consumption independent of the number of elements.
- 不足: The relative error may be high for low-frequent elements in CMS.

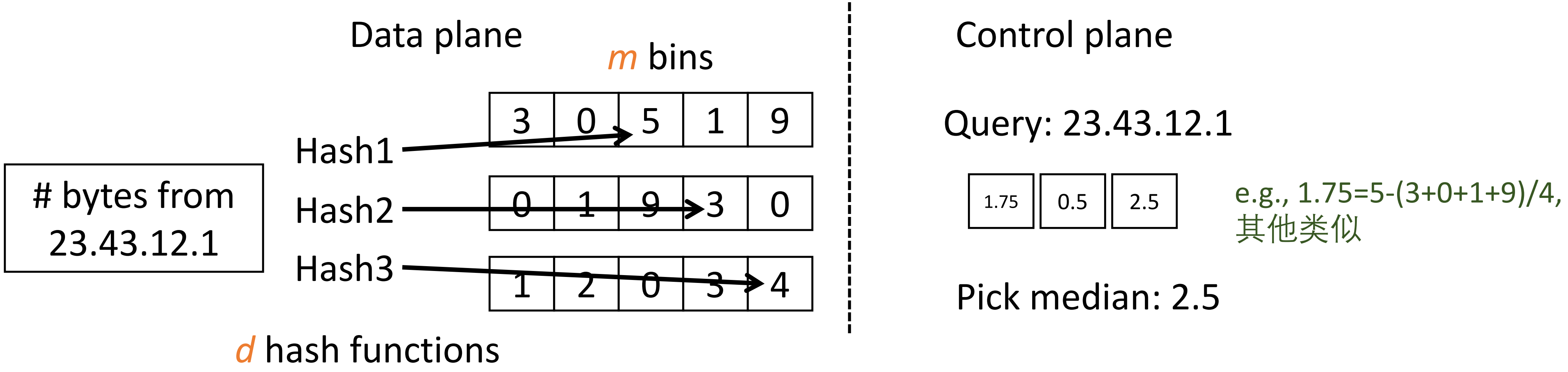
Count-Mean-Min Sketch

- CMS算法对于低频的元素, 结果不太准确，主要是因为哈希冲突比较严重，产生了噪音. 例如当m=20时，有1000个数hash到这个20桶，平均每个桶会收到50个数，这50个数的频率重叠在一块.

- **Count-Mean-Min Sketch**算法（简称CMMS）做了如下改进：
  - 新到达一个查询, 按照 Count-Min Sketch的流程取出它的d个sketch;
  - 对于每个hash函数, 估算出一个噪音, 噪音等于该行所有整数(除了被查询的这个元素)的平均值;
  - 用该行的sketch减去该行的噪音, 作为真正的sketch;
  - 返回d个真正的sketch中的那个中位数.
  - Count-Mean-Min Sketch算法能够显著改善在长尾数据上的精确度.

Count-Mean-Min Sketch

❑Example：



Problems on Data Streams

華中科技大學 计算机科学与技术学院 School of Computer Science &Technology，HUST

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

- **Sampling data from a stream**
  - Construct a random sample
- **Queries over sliding windows**
  - Number of items of type ***x*** in the last ***k*** elements of the stream
- **Filtering a data stream**
  - Select elements with property ***x*** from the stream
- **Counting distinct elements**
  - Number of distinct elements in the last ***k*** elements of the stream
- **Estimating moments (e.g., AMS method)**
  - Estimate avg./std. dev. of last ***k*** elements
- **Finding frequent recent elements**
  - Consider exponentially decaying windows, weighting the recent elements more heavily

Section 8.5: Computing 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 (该元素出现的次数)

- The ***k*th moment** **(kth-order moment, k阶矩)** is

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

Special Cases

- **0^{th} moment (0阶矩)=** number of distinct elements $\sum_{i\in A}(m_i)^k$
  - The problem just considered, e.g., FM-sketch
- **1^{st} moment (一阶矩)=** count of the numbers of elements = length of the stream
  - Easy to compute
- **2^{nd} moment (二阶矩)=** ***surprise number*** **(奇异数)** ***S*** = a measure of how uneven the distribution is
  - Example: stream of length 100, where 11 distinct values
  - Item counts: 10, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, Surprise S $= 10^2 + 10 * 9^2$=910
  - Item counts: 90, 1, 1, 1, 1, 1, 1, 1 ,1, 1, 1, Surprise S $=90^2 + 10 * 1^2 =$ 8,110
  - 奇异数越小, 说明分布越均衡. 反之则越不均衡.

AMS Method

- **AMS method (Alon, Matias, and Szegedy Algorithm, AMS 算法)** works for all moments, and it gives an **unbiased estimate**
  - Note: we will just concentrate on the **2nd moment** S

- **We pick and keep track of many variables *X* (一定数量的变量):**
  - For each variable ***X*,** we store ***X.el*** and ***X.val***
    - ***X.el*** corresponds to the item ***i*** （该变量的元素）
    - ***X.val*** corresponds to the **count** of item ***i*** （该变量的统计值）
  - Note this requires a count in main memory, so number of ***X*** s is limited
  - **Our goal is to compute 奇异数** $S = \sum_i m_i^2$

AMS Method

- 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***. We set ***X.el = i***
  - Then we maintain count ***c*** (***X.val = c***) of the number of ***i*s** in the stream starting from the chosen time ***t*** (***X.val = 1*** 初始默认. 从当前t时刻开始在流读取过程中, 每当再次看到该元素*i*，将其对应的X.val 加1)

- **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, $(X_1, X_2, \ldots X_k)$, and our final estimate will be $S = \frac{1}{k} * \sum_{j=1}^{k} f(X_j)$

Example

Count:	1	1	1	2	1	2	2	2	3	3	3	3	4	5	4
Stream:	a	b	c	b	d	a	c	d	a	b	d	c	a	a	b
	1	2	**3**	4	5	6	7	**8**	9	10	11	12	**13**	14	15

- For **AMS method,** assume we keep track of 3 *Xs* (X_1, X_2, X_3)
  - Pick 3 random time, 3rd, 8th, and 13th, use these to define 3 *Xs*
  - For time 3rd, $X_1.el = c$, and $X_1.val = 1$. At time 4th, X_1 keeps the original value. The same for time 5th, and 6th. For time 7th, $X_1.val = 2$
  - For time 8th, $X_2.el = d$, and $X_2.val = 1$. at time 9th, 10th, X_1 and X_2 keep old values. At time 11th, $X_2.val = 2$. at time 12th, $X_1.val = 3$
  - For time 13th, $X_3.el = a$, and $X_3.val = 1$. at time 14th, $X_3.val = 2$. at time 15th, 3 *Xs* unchanged.
  - So, $S = \frac{1}{3}$**[15*(2*3-1)+15*(2*2-1)+ 15*(2*2-1)]=55.**
  - In this example, n=15, a count for 5, b count for 4, c count for 3, d count for 3, so **2nd moment** $S = \sum_i m_i^2 = 5^2 + 4^2 + 3^2 + 3^2 = 59$

Combining Samples

- **In practice:**
  - Compute $f(X) = n(2c - 1)$ for as many variables $\boldsymbol{X}$ as you can fit in memory
  - Average them in groups
  - Take median of averages

- **Problem: Streams never end**
  - We assumed there was a number $\boldsymbol{n}$, the number of positions in the stream
  - But real streams go on forever, so $\boldsymbol{n}$ is a variable – the number of inputs seen so far

Streams Never End: Fixups

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

Higher-Order Moments in AMS method

华中科技大学 计算机科学与技术学院 School of Computer Science &Technology, HUST

- **For estimating kth moment we essentially use the same algorithm but change the estimate:**
  - For **k=2** we used ***n* (2•c – 1)**
  - For **k=3** we use: ***n* (3•c^2 – 3c + 1)**   (where c=**X.val**)
- **Why?**
  - **For k=2:** Remember we had $(1+3+5+\ldots+2m_i-1)$ 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:** **c^3 - (c-1)^3 = 3c^2 - 3c + 1**
- **Generally estimating *k*th moment:** Estimate = $n\,(c^k-(c-1)^k)$

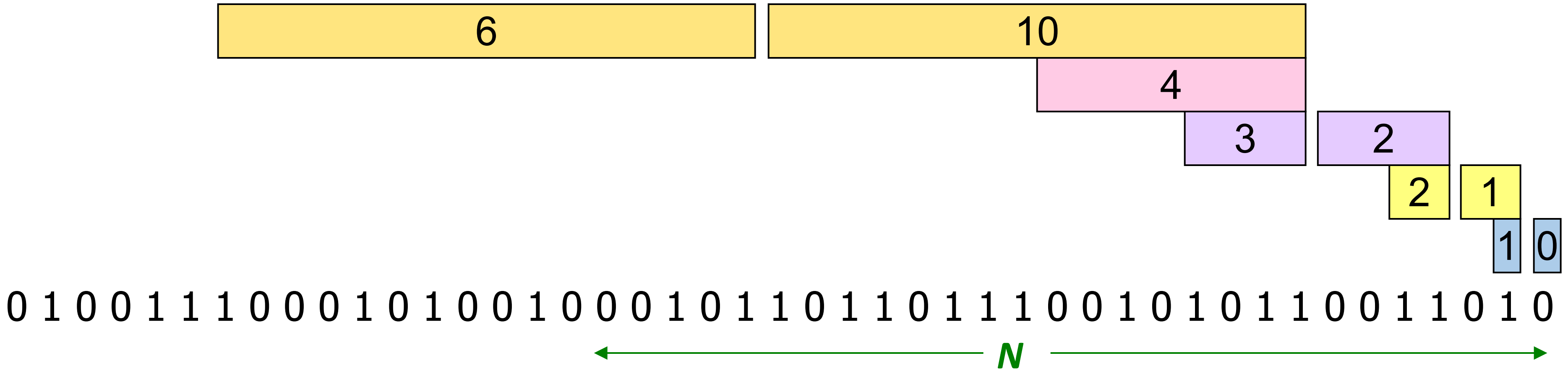
Problems on Data Streams

- **Types of queries one wants on answer on a data stream:**
  - **Sampling data from a stream**
    - Construct a random sample
  - **Queries over sliding windows**
    - Number of items of type ***x*** in the last ***k*** elements of the stream
  - **Filtering a data stream**
    - Select elements with property ***x*** from the stream
  - **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 recent elements (e.g. EDW)**
    - Consider exponentially decaying windows, weighting the recent elements more heavily

Section 8.6: Counting frequent recent elements

Counting Itemsets

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



Extensions

- **In principle, you could count frequent pairs or even larger sets the same way**
  - **One stream per itemset**

- **Drawbacks:**
  - Only approximate
  - **Number of itemsets is way too big**

Exponentially Decaying Windows

- **Exponentially decaying windows (E.D.W., 指数衰减窗口): A heuristic for selecting likely frequent item(sets)**
  - **What are "currently" most popular movies?**
    - Instead of computing the raw count in last ***N*** elements, compute a **smooth aggregation** over the whole stream
- If stream is $a_1, a_2, \ldots$ and we are taking the sum of the stream, take the answer at time t to be $= \sum_{i=1}^{t} a_i (1-c)^{t-i}$

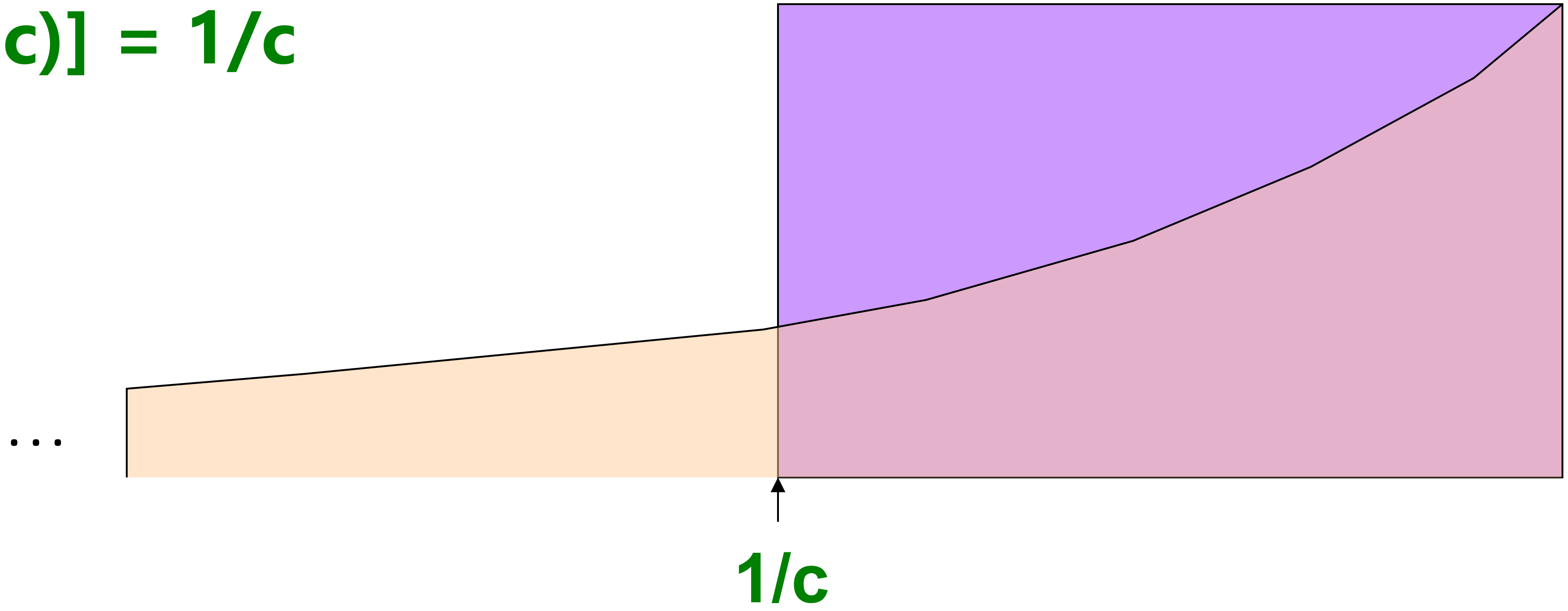
  c is a constant, presumably tiny, like $\mathbf{10^{-6}}$ or $\mathbf{10^{-9}}$
- **When new a_{t+1} arrives:** Multiply current sum by **(1-c)** and add $\mathbf{a_{t+1}}$

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 **δ_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?**
- 可以使用指数衰减窗口来解决:
  - c如果为10^{-9}, 也就是大概给出能够容纳近10亿次的滑动窗口.
  - 每个电影想象有一个独立位流来表示其购买记录. 如果电影票数据流中的位置对应当前电影, 那么这个电影的独立位流中相应位置为1, 否则为0.
  - 那这个滑动窗口中所有1的衰减求和的结果值就度量了这个电影的热门程度.
- **由于流中电影数目非常大, 所以我们希望对于非热门电影的值不用记录.**
- **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**

Example: Counting Items

- **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 (否则所有的电影的得分之和会大于1/c)
- So, **2/c** is a limit on the number of movies being counted at any time(任意时间进行计数的电影数目的上限是2/c)
- 由于任何电影票销售记录实际上只会关注一小部分电影, 所以真正计算的电影数目还会远小于2/c

Extension to Itemsets

- 扩展: **Count (some) itemsets in an E.D.W.**
  - **What are currently "hot" itemsets?**
    - **Problem:** Too many itemsets to keep counts of all of them in memory
- **When a basket B comes in:**
  - Multiply all counts by **(1-c)**
  - For uncounted items in **B**, create new count
  - Add **1** to count of any item in **B** and to any **itemset** contained in **B** that is already being counted
  - **Drop counts < ½**
  - Initiate new counts (next slide)

Initiation of New Counts

- Start a count for an itemset $\boldsymbol{S} \subseteq \boldsymbol{B}$ if every proper subset of ***S*** had a count prior to arrival of basket ***B***
  - **Intuitively:** If all subsets of **S** are being counted this means they are "**frequent/hot**" and thus ***S*** has a potential to be "**hot**"
- **Example:**
  - Start counting ***S***={**i, j**} iff both **i** and **j** were counted prior to seeing ***B***
  - Start counting ***S***={**i, j, k**} iff {**i, j**}, {**i, k**}, and {**j, k**} were all counted prior to seeing ***B***

How many counts do we need?

华中科技大学 计算机科学与技术学院 School of Computer Science &Technology，HUST

- Counts for single items **< (2/c)•(avg. number of items in a basket)**
- **Counts for larger itemsets = ??**
- **But we are conservative about starting counts of large sets**
  - If we counted every set we saw, one basket of **20** items would initiate **1M** counts

Chapter 8 总结

华中科技大学 计算机科学与技术学院 School of Computer Science &Technology, HUST

Types of queries on a data stream:

- **Sampling data from a stream - Sampling fixed proportion、Fixed-size sample, e.g., Reservoir Sampling**
  - Construct a random sample
- **Queries over sliding windows - DGIM**
  - Number of items of type ***x*** in the last ***k*** elements of the stream
- **Filtering a data stream - Bloom filter**
  - Select elements with property ***x*** from the stream
- **Counting distinct elements - FM, CMS**
  - Number of distinct elements in the last ***k*** elements of the stream
- **Estimating moments – AMS**
  - Estimate avg./std. dev. of last ***k*** elements
- **Finding frequent recent elements – EDW**
  - Consider exponentially decaying windows, weighting the recent elements more heavily