

# Data streams

Mining Massive Datasets

Carlos Castillo

Topic 10

# Sources

- Mining of Massive Datasets (2014) by Leskovec et al. (chapter 4)
  - Slides [part 1](#), [part 2](#)
- Tutorial: [Mining Massive Data Streams](#) (2019) by Michael Hahsler

# What is a data stream?

- A **potentially infinite sequence** of data points
  - Each data point can be a tuple or vector
- Examples:
  - web click-stream data → who clicks on what
  - computer network monitoring data
  - telecommunication connection data
  - readings from sensor nets
  - stock quotes

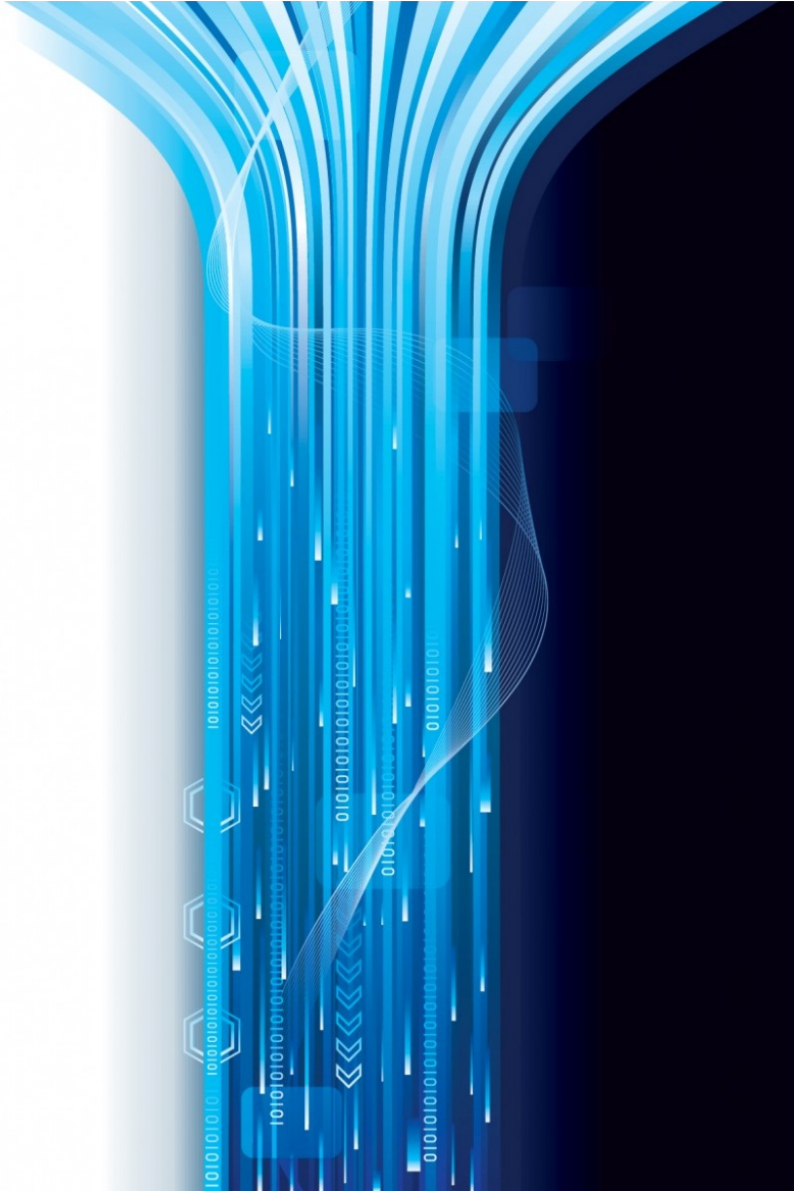
Do not confuse with “streaming,” which in vernacular typically means live video.

# Example: Apache server log

```
tecmint@TecMint ~ $ tailf /var/log/apache2/access.log
127.0.0.1 - - [31/Oct/2017:11:11:37 +0530] "GET / HTTP/1.1" 200 729 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:56.0) Gecko/20100101 Firefox/56.0"
127.0.0.1 - - [31/Oct/2017:11:11:37 +0530] "GET /icons/blank.gif HTTP/1.1" 200 1234
127.0.0.1 - - [31/Oct/2017:11:11:37 +0530] "GET /icons/folder.gif HTTP/1.1" 200 1234
127.0.0.1 - - [31/Oct/2017:11:11:37 +0530] "GET /icons/text.gif HTTP/1.1" 200 5678
127.0.0.1 - - [31/Oct/2017:11:11:38 +0530] "GET /favicon.ico HTTP/1.1" 404 500
127.0.0.1 - - [31/Oct/2017:11:12:05 +0530] "GET /tecmint/ HTTP/1.1" 200 787 "http://127.0.0.1/"
127.0.0.1 - - [31/Oct/2017:11:12:05 +0530] "GET /icons/back.gif HTTP/1.1" 200 4096
127.0.0.1 - - [31/Oct/2017:11:13:58 +0530] "GET /tecmint/Videos/ HTTP/1.1" 200 10240
127.0.0.1 - - [31/Oct/2017:11:13:58 +0530] "GET /icons/compressed.gif HTTP/1.1" 200 10240
127.0.0.1 - - [31/Oct/2017:11:13:58 +0530] "GET /icons/movie.gif HTTP/1.1" 200 10240
```

# Key properties of data streams

- **Unbounded size**
  - Data cannot be persisted on disk
  - Only summaries can be stored
- **Transient**
  - Single pass over the data
  - Sometimes real-time processing is needed
- **Dynamic**
  - May require incremental updates
  - May require to forget old data
  - Concepts “drift”
- **Temporal order** is often important



# Applications

- **Mining query streams**
  - A search engine wants to know what queries are more frequent today than yesterday
- **Mining click streams**
  - A newspaper wants to know when one of its pages starts getting an unusual number of hits per hour
- **Mining social network news feeds**
  - A social media platform wants to show trending topics

# Applications (cont.)

- **Sensor Networks**
  - Many sensors feeding into a central controller
- **Telephone call records**
  - Data feeds into customer bills as well as settlements between telephone companies
- **IP packets monitored at a switch**
  - Gather information for optimal routing
  - Detect denial-of-service attacks

# Why not simply use a relational DB?

<b>Relational DBMS</b>	<b>DSMS (Stream)</b>
persistent relations	transient streams
only current state is important	history matters
not real-time	real-time
low update rate	stream!
one time queries	continuous queries

Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom (2002). Models and issues in data stream systems. In PODS '02, pages 1–16, ACM Press.



# Why do we need new algorithms?

	<b>Traditional</b>	<b>Stream</b>
<b>passes</b>	multiple	single
<b>processing time</b>	unlimited	restricted
<b>memory</b>	disk	main memory
<b>results</b>	typically accurate	approximate
<b>distributed</b>	typically not	often

**Source:** Joao Gama, Data Stream Mining Tutorial, ECML/PKDD, 2007

# A generic stream-processing architecture

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

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

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

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

time

Ad-Hoc  
Queries

Standing  
Queries

**Processor**

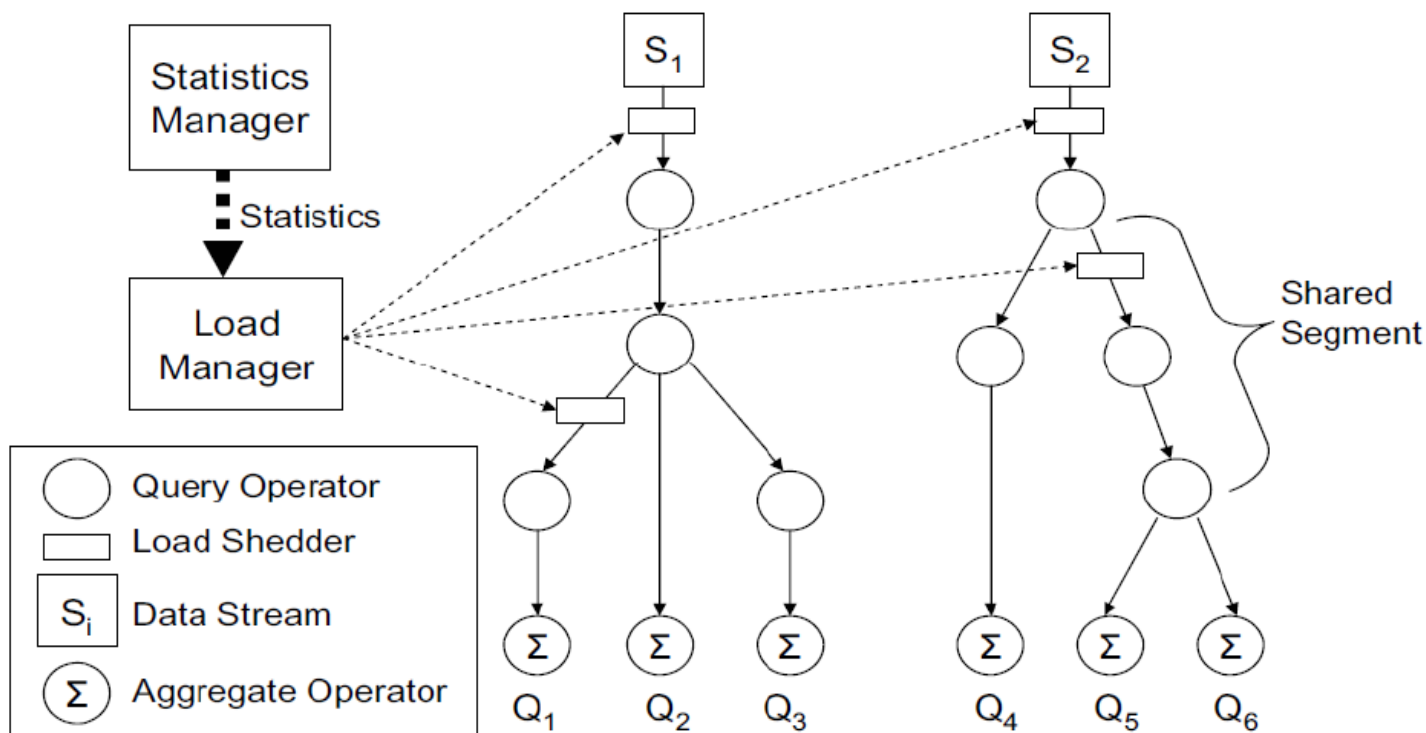
**Output**

**Limited  
Working  
Storage**

**Archival  
Storage**

# Load shedding

# Too much data? Ignore some of it



# Sampling a fixed proportion

# Sampling a fixed proportion

- Example stream:  $\langle \text{user}, \text{query}, \text{timestamp} \rangle$  from a search engine query log
- Suppose we have space to store  $1/r$  of the stream
  - E.g.: 1/10th, 1/100th, 1/1000th,
- Naïve solution:
  - Generate uniform random number in  $0 \dots (r-1)$   
`numpy.random.uniform(0, r)`
  - If the number is 0, keep the item

# What can we do with this sample?

- Approximate most frequent query
  - Pick the most frequent in the sample
- Approximate frequency of a query
  - Multiply observed frequency by  $r$
- Do people ask query  $q$ ?
  - Approximate answer (with some prob. of error)

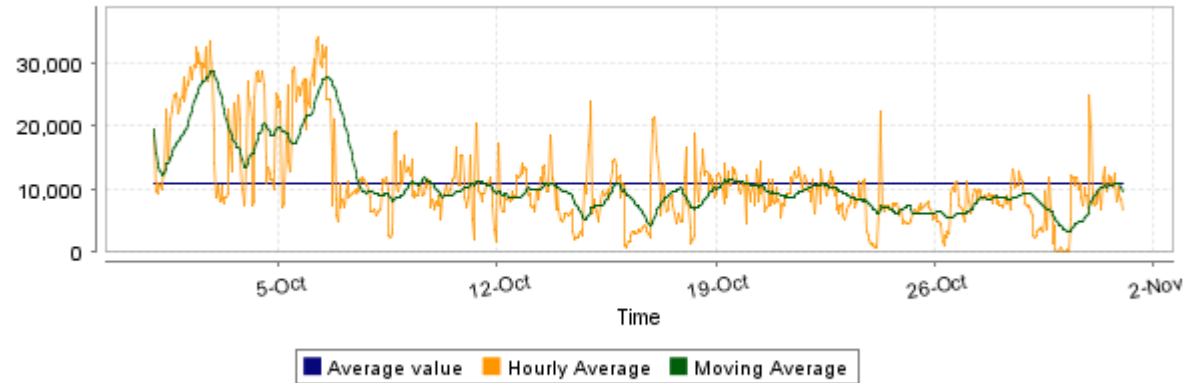
# Try it!

- We want to tell if we have seen item  $q$
- Suppose we have seen  $n$  items
- Suppose we have sampled a fraction  $1/r$
- Suppose item  $q$  appears with probability  $p(q)$
- What is the probability of a:
  - False Positive? (*Item  $q$  was not in the stream but we said it was*)
  - False Negative? (*Item  $q$  was in the stream but we said it was not*)



# What can we do with this...? (cont.)

- Approximate num. queries per minute



- Peak frequency
  - Multiply observed peak by  $r$

# But there are questions we cannot answer well

- **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  $1/10^{\text{th}}$  of the queries ( $r=10$ )

- Sample will contain  $x/10$  of the singleton queries at least once

- Sample will contain  $2d/10$  of the duplicate queries at least once


- **Sample will contain  $d/100$  pairs of duplicates**

- $d/100 = 1/10 \cdot 1/10 \cdot d$

- **Of the  $d$  duplicates,  $18d/100$  will be seen once\***

- $18d/100 = ((1/10 \cdot 9/10) + (9/10 \cdot 1/10)) \cdot d$

- So the sample-based answer is


$$\frac{\frac{x}{10} + \frac{18d}{100} + \frac{d}{100}}{\frac{d}{100}} = \frac{d}{10x + 19d}$$

\* Copy A is in the selected part, copy B in the unselected part, or viceversa

# But there are questions we cannot answer well (cont.)

- 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  $1/10^{\text{th}}$  of the queries ( $r=10$ )

- Sample will contain  $x/10$  of the singleton queries at least once

- Sample will contain  $2d/10$  of the duplicate queries at least once

- Sample will contain  $d/100$  pairs of duplicates

- $d/100 = 1/10 \cdot 1/10 \cdot d$

- Of the  $d$  duplicates,  $18d/100$  will be seen once

- $18d/100 = ((1/10 \cdot 9/10) + (9/10 \cdot 1/10)) \cdot d$

- So the sample-based answer is

$$\begin{array}{c}
 \text{Observed duplicates} \\
 \frac{\frac{x}{10} + \frac{18d}{100} + \frac{d}{100}}{\text{Observed singletons} \quad \text{Observed duplicates}} = \frac{d}{10x + 19d}
 \end{array}$$

**WRONG!**

# How do we solve it?

- We need to **sample  $1/r$  of users** and all of their queries

How do we do this?

```
<user1, query, timestamp>
<user2, query, timestamp>
<user2, query, timestamp>
<user3, query, timestamp>
<user1, query, timestamp>
<user3, query, timestamp>
<user2, query, timestamp>
<user1, query, timestamp>
<user2, query, timestamp>
...
```

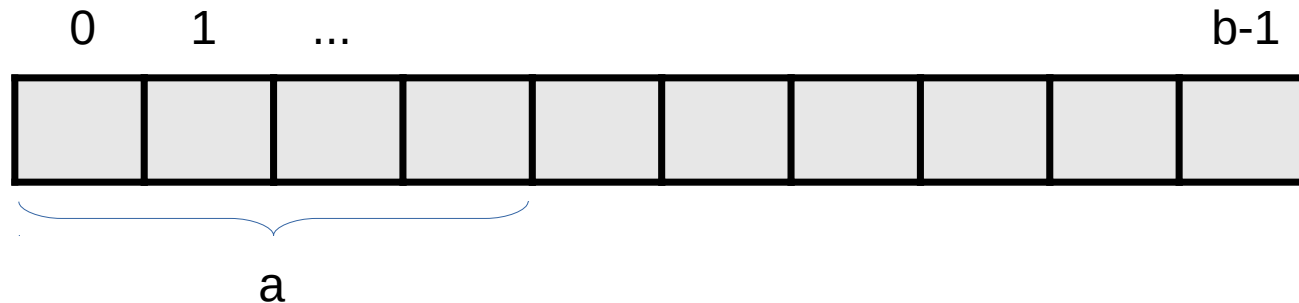
# How do we solve it?

- We need to **sample  $1/r$  of users** and all of their queries
- How do we do this?
  - **Hashing!**
  - Given  $\langle \text{user}, \text{query}, \text{timestamp} \rangle$
  - Compute  $h(\text{user}) \rightarrow 0, 1, \dots, (r-1)$
  - Keep tuple if hash value is 0

```
<user1, query, timestamp>
<user2, query, timestamp>
<user2, query, timestamp>
<user3, query, timestamp>
<user1, query, timestamp>
<user3, query, timestamp>
<user2, query, timestamp>
<user1, query, timestamp>
<user2, query, timestamp>
...
```

# In general ...

- To sample a fraction  $a/b$  of a stream by key
- Compute  $h(\text{key}) \rightarrow 0, 1, \dots, (b-1)$
- Keep if  $h(\text{key}) < a$



# Sampling a fixed-size sample

# A fixed-size sample

- **We normally do not know the stream size**
- **We just know how much storage space we have**
- Suppose we have storage space  $s$  and want to maintain a random sample  $S$  of size  $s=|S|$
- **Requirement:** after seeing  $n$  items, each of the  $n$  items should be in our sample with probability  $s/n$ 
  - *No item should have an advantage or disadvantage*



# Bad solutions

- Suppose stream =  $\langle a, f, e, b, g, r, u, \dots \rangle$
- **Requirement:** after seeing  $n$  items, each of the  $n$  items should be in our sample with probability  $s/n$
- Suppose  $s=2$ 
  - Always keep first 2? No, because then  $p(a) = 1 \neq 0 = p(e)$
  - Always keep last 2? No, because then  $p(a) = 0 \neq 1 = p(u)$
- Sample some ... which? Then evict some ... which?

# Reservoir sampling

(one of the most beautiful algorithms of this course)

- Elements  $x_1, x_2, x_3, \dots, x_i, \dots$
- Store all first  $s$  elements  $x_1, x_2, \dots, x_s$
- Suppose element  $x_n$  arrives
  - With probability  $1 - s/n$ , ignore this element
  - With probability  $s/n$ :
    - Discard a random element from the reservoir
    - Insert element  $x_n$  into the reservoir

# Try it!

- Suppose input is  $\langle a, b, c, \dots \rangle$
- Suppose  $s = 2$
- We have just processed element 3 = “c”
- What is:
  - Probability “a” is in the sample?
  - Probability “b” is in the sample?
  - Probability “c” is in the sample?
- If you are done quickly, try one more element, “d”

## RESERVOIR SAMPLING

Store all first  $s$  elements  $x_1, x_2, \dots, x_s$

When element  $x_n$  arrives

- With probability  $1 - s/n$ , ignore
- With probability  $s/n$ :
  - Discard randomly from reservoir
  - Insert element  $x_n$  into the reservoir

# Proof by induction

- **Inductive hypothesis:** after  $n$  elements seen each of them is sampled with probability  $s/n$
- **Inductive step:** element  $x_{n+1}$  arrives,
  - what is the probability than an already-sampled element  $x_i$  stays in the sample?

$$\underbrace{\left(1 - \frac{s}{n+1}\right)}_{x_{n+1} \text{ not sampled}} + \underbrace{\left(\frac{s}{n+1}\right)}_{x_{n+1} \text{ sampled}} \cdot \underbrace{\left(\frac{s-1}{s}\right)}_{x_i \text{ not evicted}} = \frac{n}{n+1}$$

# Proof by induction (cont.)

- Tuple  $x_{n+1}$  is sampled with probability  $\frac{s}{n+1}$  ✓
- Tuples  $x_i$  with  $i \leq n$ 
  - Were in the sample with probability  $s/n$
  - Stay in the sample with probability  $n/(n+1)$
  - Hence, are in the sample with probability

$$\frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1} \quad \checkmark$$

# Recency-biased reservoir sampling

- Before we had  $p(i) = s/n$ 
  - Probability of element  $x_i$  to be included
  - Reservoir of size  $s$
  - Stream so far of size  $n$
- Suppose we want a different  $p(i) \propto f(i,n)$ 
  - Example:  $f(i,n)$  larger for more recent items

# Recency-biased reservoir sampling (cont.)

- Suppose we want  $p(i) \propto f(i, n) = e^{-\lambda(n-i)}$
- Parameter  $\lambda \in [0, 1]$  is a decay factor and  $s < \frac{1}{\lambda}$
- Algorithm: reservoir starts empty

At time  $n$ , it is  $F(n) \in [0, 1]$  full

$x_{n+1}$  arrives and is inserted with probability  $\lambda \cdot s$

If  $x_{n+1}$  is inserted, remove from reservoir a random element with probability  $F(n)$

# Bloom filters



# Filtering a data stream

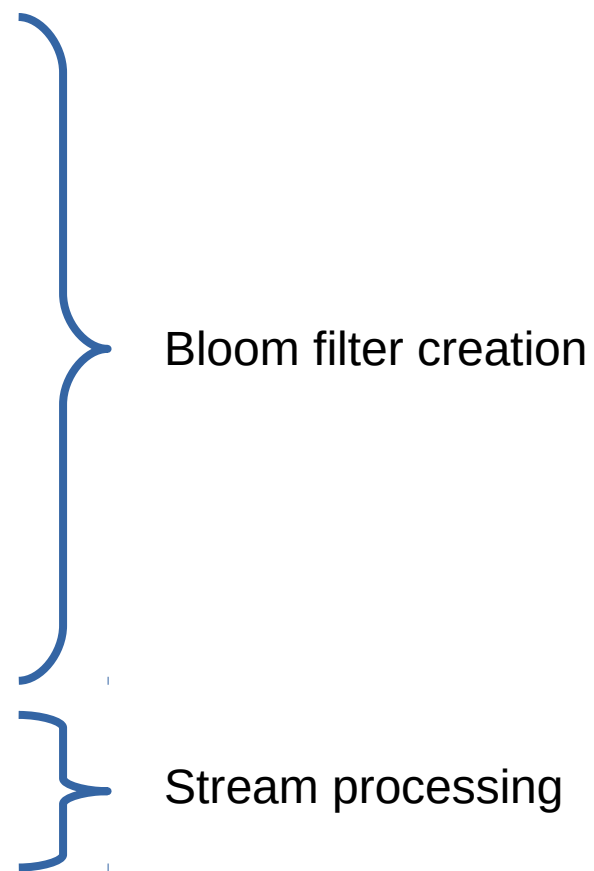
- Suppose we have a large set  $S$  of keys
- We want to filter a stream  $\langle \text{key}, \text{data} \rangle$  to let pass only the elements for which  $\text{key} \in S$
- Example: key is an e-mail address, we have a total of  $|S|=10^9$  allowed e-mail addresses

Naïve solution?

# Filtering a data stream

- Suppose we have a large set  $S$  of keys
- We want to filter a stream  $\langle \text{key}, \text{data} \rangle$  to let pass only the elements for which  $\text{key} \in S$
- Example: key is an e-mail address, we have a total of  $|S|=10^9$  allowed e-mail addresses
- Naïve solution? Hash table won't work, too big!

# Bloom Filter (1-bit case)

- Given a set of keys  $S$
  - Create a bit array  $B[]$  of  $n$  bits
    - Initialize to all 0s
  - Pick a hash function  $h$  with range  $[0, n)$ 
    - For each member of  $s \in S$ 
      - Hash to one of  $n$  buckets
      - Set that bit to 1, i.e.,  $B[h(s)] \leftarrow 1$
  - For each element  $a$  of the stream
    - Output  $a$  if and only if  $B[h(a)] == 1$
- 
- Bloom filter creation
- Stream processing

# Bloom Filter is an approximate filter

- Can it output an element with a key not in  $S$ ?
- Can it not output an element with a key in  $S$ ?

# Bloom Filter is an approximate filter

- Can it output an element with a key not in  $S$ ?

Yes, due to hash collisions  $h(x)=h(y)$  when  $x \neq y$

- Can it not output an element with a key in  $S$ ?

No, because  $h(x)$  is always the same for  $x$

**Bloom filters are *permissive* (not *strict*)**

# Bloom filter

- A bloom filter is:
  - An array of  $n$  bits, initialized as 0
  - A collection of hash functions  $h_1, h_2, \dots, h_k$
  - A set  $S$  of  $m$  key values
- The purpose of the bloom filter is to allow all stream items whose key is in  $S$

# Bloom filter initialization

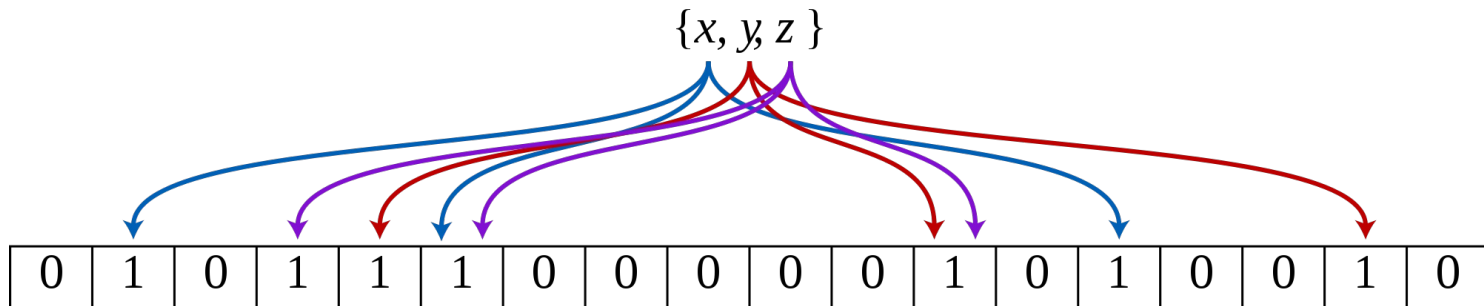
For all positions  $i$  in  $[0, n-1]$

$B[i] \leftarrow 0$

For all keys  $K$  in  $S$ :

For every hash function  $h_1, h_2, \dots, h_k$

$B[h_i(K)] \leftarrow 1$



# Bloom filter usage

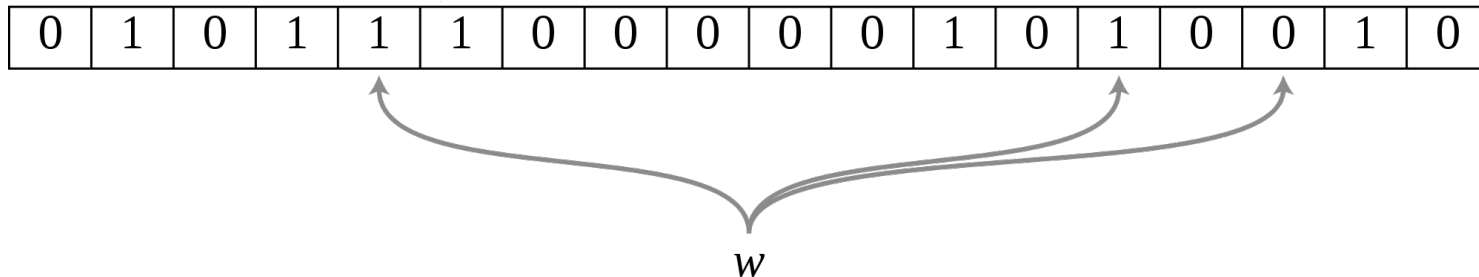
For each input element  $\langle \text{key}, \text{data} \rangle$

$\text{allow} \leftarrow \text{TRUE}$

For every hash function  $h_1, h_2, \dots, h_k$

$\text{allow} \leftarrow \text{allow} \wedge B[h_i(K)] == 1$

output element if  $\text{allow} == \text{TRUE}$





# Characteristics of Bloom Filters

- Are lax (not strict) and let some items pass
  - May require a second-level check to make filter strict, for instance store output on disk files and then check against hash tables (slower)
- Implementations can be very fast
  - E.g., use hardware words for the bit table

# Preliminaries for the analysis: targets and darts

- Suppose we throw  $y$  darts at  $x$  targets
  - All darts will hit one of the targets
- After throwing the darts, how many distinct targets can we expect to hit at least once?
  - Prob. that a given dart will not hit a given target is  $(x-1)/x = 1-1/x$
  - Prob. none of the  $y$  darts will hit a given target is  $(1-1/x)^y = (1-1/x)^{x(y/x)}$
  - Using  $(1-\varepsilon)^{1/\varepsilon} \simeq 1/e$  for small  $\varepsilon$
  - Prob. none of the  $y$  darts will hit a given target is  $(1/e)^{y/x}$

# Analysis of the 1-bit Bloom Filter

- Each element of  $S$  is a dart  $|S|=y$
- Each bit in the array is a target  $n=x$
- Suppose  $y=|S|=10^9$  (1 G) and  $x=n=8 \times 10^9$  (8 GB)
- Prob. that a given bit is NOT set to 1 (dart does not hit the target) is  $(1/e)^{y/x} = (1/e)^{1/8}$
- Prob. bit is set to 1 is  $1 - (1/e)^{1/8} = 0.1175$

About 12% of bits are set to one in the Bloom Filter  
this is also the false-hit probability of this method

# General case

- $|S|=m$  keys, array has  $n$  bits
- $k$  hash functions
- Targets  $x=n$ , darts  $y=km$
- Probability that a bit remains 0 is  $e^{-km/n}$
- Example:  
We can pick  $k=n/m$  to obtain collision probability  $1/e = 37\%$

# Analysis of a 2-bit Bloom Filter

- Suppose  $|S|=10^9$  (1 G) and  $n=8 \times 10^9$  (8 GB)
- Suppose we use two hash functions
- Prob. that a given bit is NOT set to 1 (dart does not hit the target) is  $(1/e)^{y/x} = (1/e)^{1/4}$
- Prob. a bit is set to 1 is  $1 - (1/e)^{1/4}$
- Prob. two bits are set to 1 is  $(1 - (1/e)^{1/4})^2 = 0.0493$
- We have a false hit probability of about 5% with two hash functions, while the probability was about 12% with only one

# How many hash functions to use?

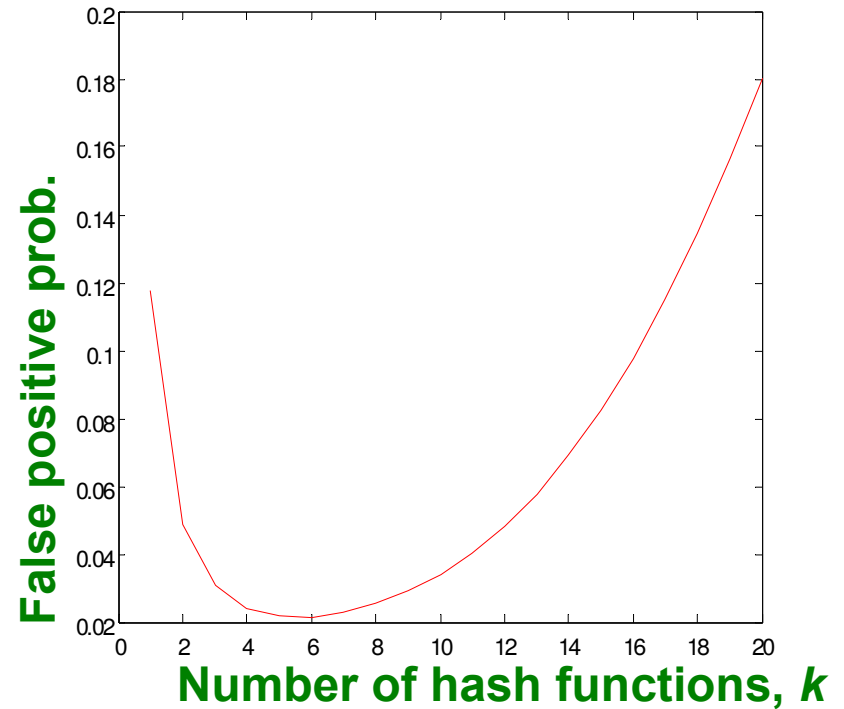
Too few: test is too unspecific. Too many: table becomes too crowded.

- **$m = 1$  billion,  $n = 8$  billion**

$$k = 1: (1 - e^{-1/8}) = 0.1175$$

$$k = 2: (1 - e^{-1/4})^2 = 0.0493$$

- **What happens as we keep increasing  $k$ ?**
  - “Optimal” value of  $k$ :  $n/m \ln(2)$
  - **In our case:** Optimal  $k = 8 \ln(2) = 5.54 \approx 6$
  - Error at  $k = 6$ :  $(1 - e^{-1/6})^2 = 0.0235$



# Probabilistic counting

# Motivating example

## how many neighbors?

- Let  $n(u, h)$  be the number of nodes reachable through a path of length up to  $h$  from node  $u$
- Naïve method
  - Maintain a set for each node  $u$ , initialize  $S(u) = \{u\}$
  - Repeat  $h$  times:

$$S(u) = S(u) \cup \bigcup_{v \text{ neighbor of } u} S(v)$$

- Answer  $n(u, h) = |S(u)|$



# What is the problem with this method?

- Let  $n(u, h)$  be the number of nodes reachable through a path of length up to  $h$  from node  $u$
- Naïve method
  - Maintain a set for each node  $u$ , initialize  $S(u) = \{u\}$
  - Repeat  $h$  times:

$$S(u) = S(u) \cup \bigcup_{v \text{ neighbor of } u} S(v)$$

- Answer  $n(u, h) = |S(u)|$

# Let's look at each node

- We will receive a stream of items
  - Our neighbors at distance  $\leq h$
  - Repeated many times because of loops
- We want to use a small amount of memory
- We don't care which items are in the stream
- We just want to know how many are distinct

# Counting fishes with pebbles

- Normally, to count with pebbles, you add one pebble every time you see an event
- How do you extend this method to count up to 1000 fishes with 10 pebbles?
- Assume you have access to a random number generator but not to an abacus for ... reasons



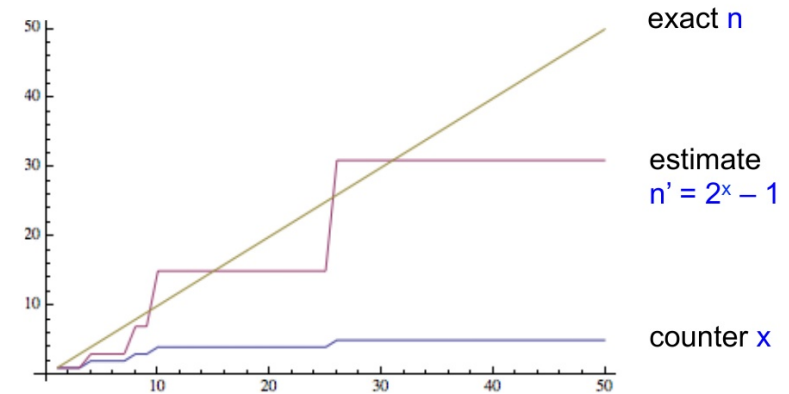
# Morris' probabilistic counting (1977)

$x \leftarrow 0$

For each of the  $n$  events:

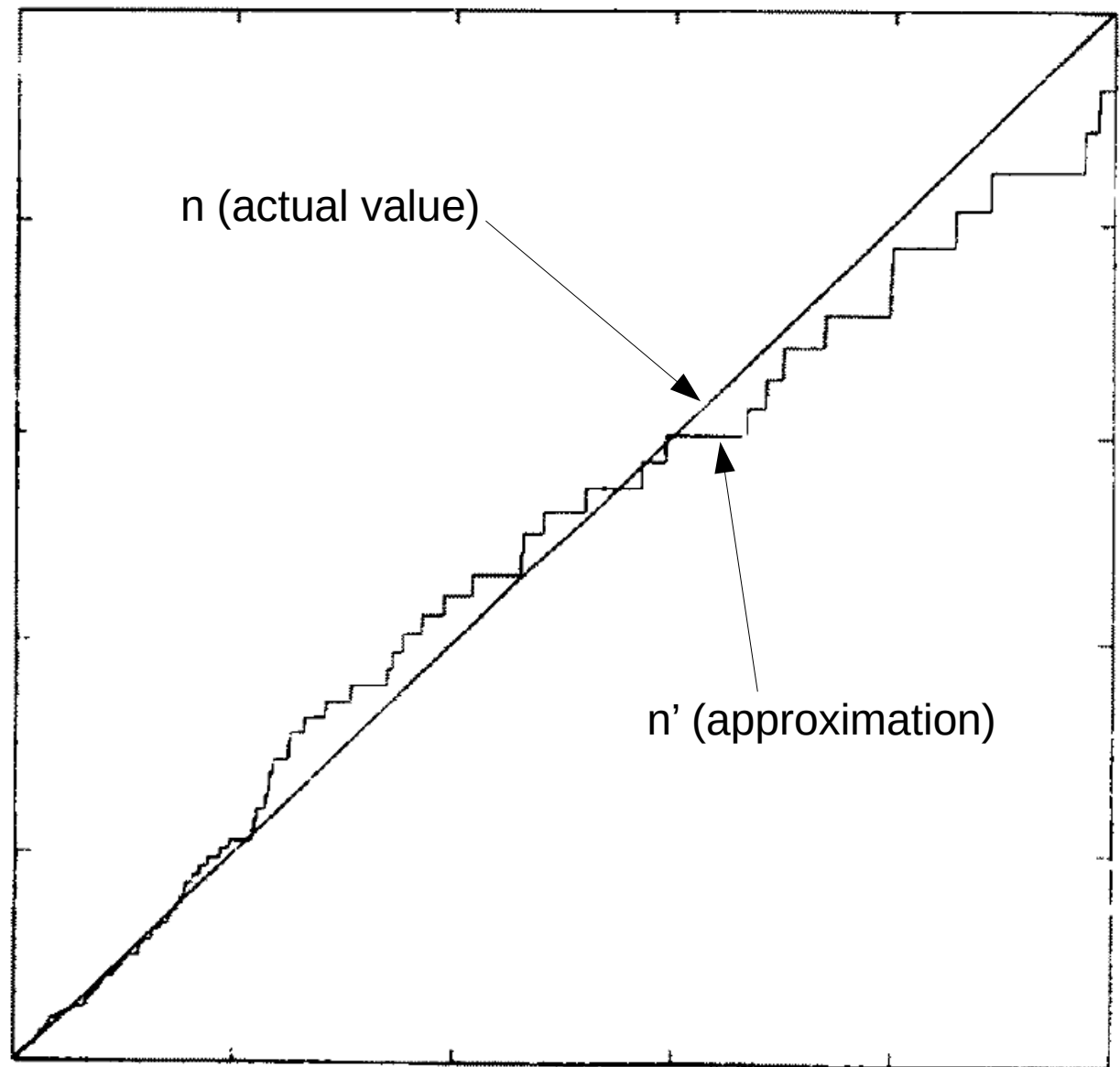
$x \leftarrow x + 1$  with probability  $(1/2)^x$

Return estimate  $n' = 2^{x+1} - 1$



*Counter  $x$  needs only  $\log_2(n)$  bits*

# Simulation results by Flajolet (1985)



# Morris' algorithm provides an unbiased estimator

- Init  $x=0$ , let  $p_x = 2^{-x}$ , estimate  $n' = 2^x - 1$
- $n = 1$ 
  - before:  $x = 0$   $p_0 = 1$ ;
  - prob. 1:  $x \rightarrow 1$
  - estimate  $n' = 2^1 - 1 = 1 = n$
- $n = 2$ 
  - before:  $x = 1$ ;  $p_1 = \frac{1}{2}$
  - prob.  $\frac{1}{2}$ :  $x$  stays at 1;  $n' = 2^1 - 1 = 1$
  - prob.  $\frac{1}{2}$ :  $x \rightarrow 2$ .  $n' = 2^2 - 1 = 3$
  - $E[n'] = \frac{1}{2} \times 1 + \frac{1}{2} \times 3 = 2 = n$

# Morris' algorithm provides an unbiased estimator (cont.)

Let  $X(n)$  denote random counter  $x$  after  $n^{\text{th}}$  arrival

Initialize  $X(0) = 0$ ; increment w.p.  $p_x = 2^{-x}$

Estimate  $n' = 2^{X(n)} - 1$

$$\begin{aligned} E[2^{X(n)}] &= \sum_{j=1, \dots, n-1} \Pr[X(n-1) = j] E[2^{X(n)} \mid X(n-1) = j] \\ &= \sum_{j=1, \dots, n-1} \Pr[X(n-1) = j] (p_j 2^{j+1} + (1 - p_j) 2^j) \\ &= \sum_{j=1, \dots, n-1} \Pr[X(n-1) = j] (2^j + 1) \\ &= E[2^{X(n-1)}] + 1 \end{aligned}$$

Iterating:  $E[2^{X(n)}] = E[2^{X(0)}] + n = 1 + n$

Therefore:  $E[2^{X(n)} - 1] = n$

# Flajolet-Martin algorithm for counting distinct elements

- For every element  $u$  in the stream, compute hash  $h(u)$
- Let  $r(u)$  be the number of trailing zeros in hash value
  - Example: if  $h(u) = 001011101\underline{000}$  then  $r(u) = 3$
- Maintain  $R = \max r(u)$  seen so far
- Output  $2^R$  as an estimator of the number of distinct elements seen so far



# Flajolet-Martin algorithm

## (intuitive, hand-waving explanation)

- Let  $r(u)$  be the number of trailing zeros in hash value, keep  $R = \max r(u)$ , output  $2^R$  as estimate
- Repeated items don't change our estimates because their hashes are equal
- About  $\frac{1}{2}$  of distinct items hash to \*\*\*\*\*0
  - To actually see a \*\*\*\*\*0, we expect to wait until seeing 2 distinct items
- About  $\frac{1}{4}$  of distinct items hash to \*\*\*\*\*00
  - To actually see a \*\*\*\*\*00, we expect to wait until seeing 4 items
- ...
- If we actually saw a hash value of \*\*\*000...0 (having  $R$  trailing zeros) then on expectation we saw  $2^R$  different items

# Flajolet-Martin, correctness proof

- Let  $m$  be the number of distinct elements
- Let  $z(r)$  be the probability of finding a tail of  $r$  zeroes
- We will prove that
$$z(r) \rightarrow 1 \text{ if } m \gg 2^r$$
$$z(r) \rightarrow 0 \text{ if } m \ll 2^r$$
- Hence  $2^r$  should be around  $m$

# Flajolet-Martin, correctness proof (cont.)

- Probability a hash value ends in  $r$  zeroes =  $(1/2)^r$ 
  - Assuming  $h(u)$  produces values at random
  - Prob. random binary ends in  $r$  zeroes =  $(1/2)^r$
- Probability of seeing  $m$  distinct elements and NOT seeing a tail of  $r$  zeroes =  $(1 - (1/2)^r)^m$

# Flajolet-Martin, correctness proof (cont.)

- Probability of seeing  $m$  distinct elements and NOT seeing a tail of  $r$  zeroes  $= (1 - (1/2)^r)^m$
- Remember  $(1-\varepsilon)^{1/\varepsilon} \simeq 1/e$  for small  $\varepsilon$
- Hence 
$$\left(1 - \left(\frac{1}{2}\right)^r\right)^m = \left(1 - \left(\frac{1}{2}\right)^r\right)^{\frac{m \left(\frac{1}{2}\right)^r}{\left(\frac{1}{2}\right)^r}} \approx \left(\frac{1}{e}\right)^{\left(\frac{m}{2^r}\right)}$$

# Flajolet-Martin, correctness proof (cont.)

- Probability of seeing  $m$  distinct elements and NOT seeing a tail of  $r$  zeroes  $\approx (1/e)^{\left(\frac{m}{2^r}\right)}$
- If  $m \gg 2^r$ , this tends to 0
  - We almost certainly will see a tail of  $r$  zeroes
- If  $m \ll 2^r$ , this tends to 1
  - We almost certainly will not see a tail of  $r$  zeroes
- Hence,  $2^r$  should be around  $m$

# Flajolet-Martin: increasing precision

- Idea: repeat many times or compute in parallel for multiple hash functions
- How to combine?
  - **Average?**  $E[2^r]$  is infinite, extreme values will skew the number excessively
  - **Median?**  $2^r$  is always a power of 2
- **Solution:** group hash functions, take median of values obtained in each group, then average across groups

# Let's go back to counting neighbors

## Naïve method:

Maintain a set for each node  $u$ , initialize  $S(u) = \{u\}$

Repeat  $h$  times:  $S(u) = S(u) \cup \bigcup_{v \text{ neighbor of } u} S(v)$

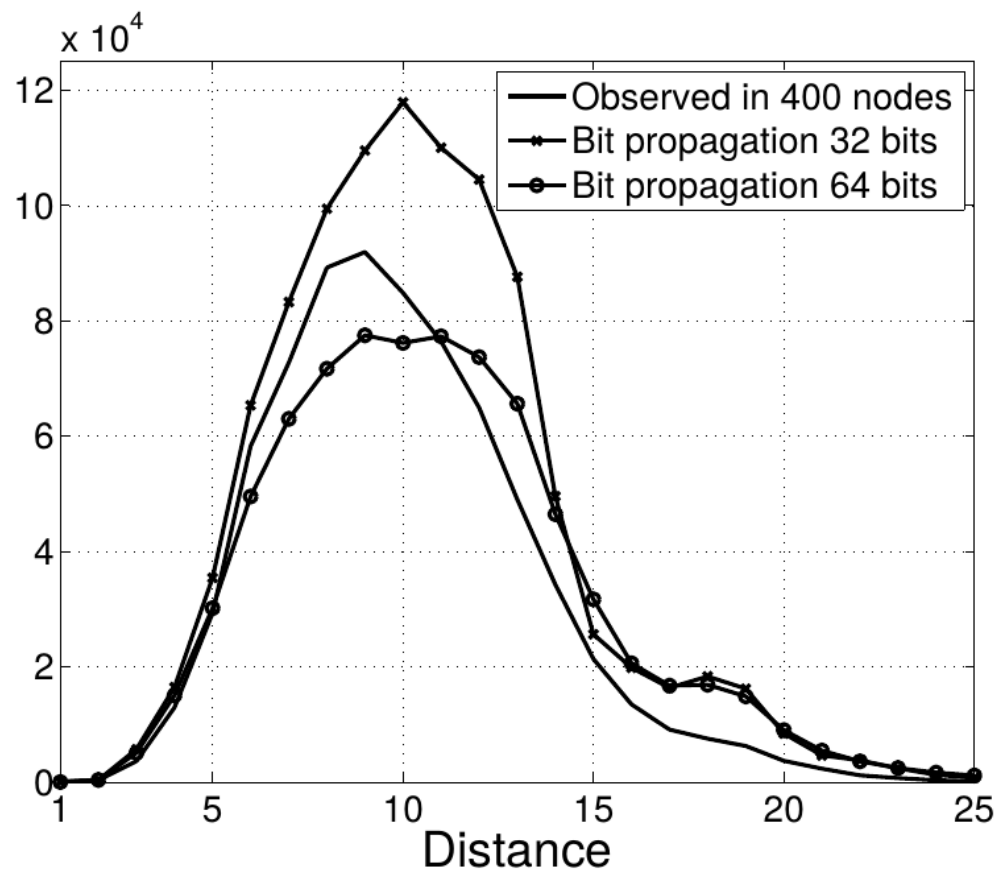
Answer  $n(u, h) = |S(u)|$

## ANF method:

```
// Set  $\mathcal{M}(x, 0) = \{x\}$ 
FOR each node  $x$  DO
     $M(x, 0) =$  concatenation of  $k$  bitmasks
                  each with 1 bit set ( $P(\text{bit } i) = .5^{i+1}$ )
FOR each distance  $h$  starting with 1 DO
    FOR each node  $x$  DO  $M(x, h) = M(x, h - 1)$ 
    // Update  $\mathcal{M}(x, h)$  by adding one step
    FOR each edge  $(x, y)$  DO
         $M(x, h) = (M(x, h) \text{ BITWISE-OR } M(y, h - 1))$ 
    // Compute the estimates for this  $h$ 
    FOR each node  $x$  DO
        Individual estimate  $\hat{IN}(x, h) = (2^b) / .77351$ 
        where  $b$  is the average position of the least zero bits
        in the  $k$  bitmasks
```

# Example of another variant of the same type of algorithm

- More repetitions of the algorithm yield better precision





# Estimating moments

# Moments of order k

- If a stream has  $A$  distinct elements, and each element has frequency  $m_i$
- The  $k^{\text{th}}$  order moment of the stream is  $\sum_i m_i^k$
- **The  $0^{\text{th}}$  order moment** is the number of distinct elements in the stream
- **The  $1^{\text{st}}$  order moment** is the length of the stream

# Moments of order k (cont.)

- The  $k^{\text{th}}$  order moment of the stream is  $\sum_i m_i^k$
- **The 2<sup>nd</sup> order moment** is also known as the “surprise number” of a stream (large values = more uneven distribution)

$$\sum m_i^2$$

$m_i$	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9	i=10	i=11	2 <sup>nd</sup> moment
Seq1	10	9	9	9	9	9	9	9	9	9	9	910
Seq2	90	1	1	1	1	1	1	1	1	1	1	8110

# Method for second moment

- Assume (for now) that we know  $n$ , the length of the stream
- We will sample  $s$  positions
- For each sample we will have  $X.element$  and  $X.count$
- We sample  $s$  random positions in the stream
  - $X.element$  = element in that position,  $X.count \leftarrow 1$
  - When we see  $X.element$  again,  $X.count \leftarrow X.count + 1$
- Estimate second moment as  $n(2 \times X.count - 1)$

# Method for second moment (cont.)

- Example: a, b, c, b, d, a, c, d, a, b, d, c, a, a, b  
 $m_a = 5, m_b = 4, m_c = 3, m_d = 3$   
second moment =  $5^2 + 4^2 + 3^2 + 3^2 = 59$
- Suppose we sample  $s=3$  variables  $X_1, X_2, X_3$
- Suppose we pick the 3<sup>rd</sup>, 8<sup>th</sup>, and 13<sup>th</sup> position at random
- $X_1.\text{element}=c, X_2.\text{element}=d, X_3.\text{element}=a$
- $X_1.\text{count}=3, X_2.\text{count}=2, X_3.\text{count}=2$  (we count forwards only!)
- Estimate  $n(2 \times X.\text{count} - 1)$ , first estimate =  $15(6-1) = 75$ ,  
second estimate  $15(4-1) = 45$ , third estimate  $15(4-1) = 45$ ,  
average of estimates =  $55 \approx 59$

# Method for second moment (cont.)


- Example: a, b, c, b, d, a, c, d, a, b, d, c, a, a, b
- Suppose we pick the 3<sup>rd</sup>, 8<sup>th</sup>, and 13<sup>th</sup> position at random
- $X_1.\text{element}=c$ ,  $X_2.\text{element}=d$ ,  $X_3.\text{element}=a$
- $X_1.\text{count}=3$ ,  $X_2.\text{count}=2$ ,  $X_3.\text{count}=2$

# Why this method works?

- Let  $e(i)$  be the element in position  $i$  of the stream
- Let  $c(i)$  be the number of times  $e(i)$  appears in positions  $i, i+1, i+2, \dots, n$
- Example:  $a, b, c, b, d, a, c, d, a, b, d, c, a, a, b$   

$c(6) = ?$

# Why this method works?

- Let  $e(i)$  be the element in position  $i$  of the stream
- Let  $c(i)$  be the number of times  $e(i)$  appears in positions  $i, i+1, i+2, \dots, n$
- Example:  $a, b, c, b, d, \underline{a}, c, d, \underline{a}, b, d, c, \underline{a}, \underline{a}, b$   
 $c(6) = 4$   (remember: we count forwards only!)



# Why this method works? (cont.)

- $c(i)$  is the number of times  $e(i)$  appears in positions  $i, i+1, i+2, \dots, n$
- $E[n(2 \times X.\text{count} - 1)]$  is the average of  $n(2c(i) - 1)$  over all positions  $i=1\dots n$

$$E[n(2 \times X.\text{count} - 1)] = \frac{1}{n} \sum_{i=1}^n n(2c(i) - 1)$$

$$E[n(2 \times X.\text{count} - 1)] = \sum_{i=1}^n (2c(i) - 1)$$

# Why this method works? (cont.)

$$E[n(2 \times X.\text{count} - 1)] = \sum_{i=1}^n (2c(i) - 1)$$

- Now focus on element  $a$  that appears  $m_a$  times in the stream
  - The last time  $a$  appears this term is  $2c(i) - 1 = 2 \times 1 - 1 = 1$
  - Just before that,  $2c(i) - 1 = 2 \times 2 - 1 = 3$
  - ...
  - Until  $2m_a - 1$  for the first time  $a$  appears
- Hence

$$E[n(2 \times X.\text{count} - 1)] = \sum_a 1 + 3 + 5 + \dots + (2m_a - 1) = \sum_a m_a^2$$

# For higher order moments ( $v = X.\text{count}$ )

- For **second order** moment
  - We use  $n(2v-1) = n(v^2 - (v-1)^2)$
- For **third order** moment
  - We use  $n(3v^2 - 3v + 1) = n(v^3 - (v-1)^3)$
- For  **$k^{\text{th}}$  order** moment
  - We use  $n(v^k - (v-1)^k)$

# For infinite streams

- Use a **reservoir sampling** strategy
- If we want  $s$  samples
  - Pick the first  $s$  elements of the stream setting  $X_i.\text{element} \leftarrow e(i)$  and  $X_i.\text{count} \leftarrow 1$  for  $i=1\dots s$
  - When element  $n+1$  arrives
    - Pick  $X_{n+1}.\text{element}$  with probability  $s/(n+1)$ , evicting one of the existing elements at random and setting  $X.\text{count} \leftarrow 1$
- As before, probability of an element is  $s/n$

# Summary

# Things to remember

- Reservoir sampling
- Bloom filter
- Probabilistic counting algorithms:
  - Morris
  - Flajolet-Martin
- $k^{\text{th}}$  order moments of a stream

# Exercises for this topic

- Mining of Massive Datasets (2014) by Leskovec et al.
  - Exercises 4.2.5
  - Exercises 4.3.4
  - Exercises 4.4.5
  - Exercises 4.5.6