INFO-H515: BIG DATA: DISTRIBUTED MANAGEMENT

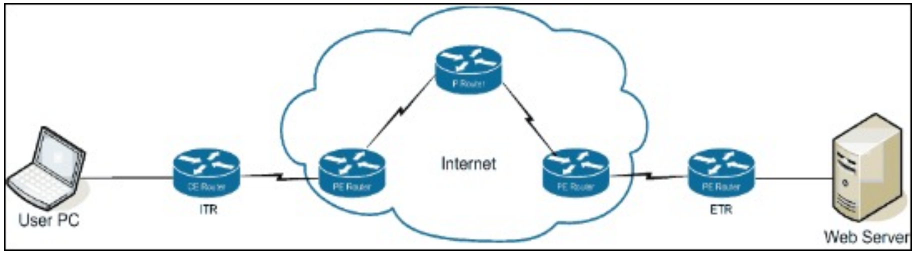
Lecture 4: Stream and Big Data Algorithms

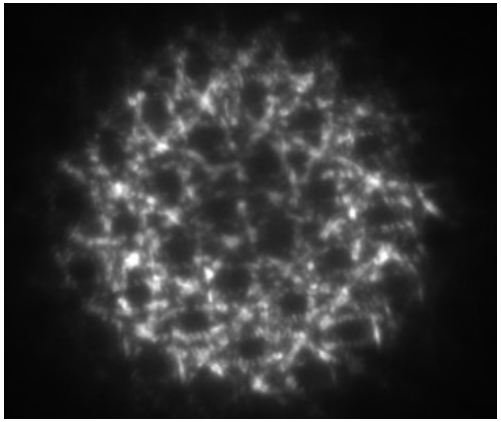
Dimitris Sacharidis

2023–2024

Introduction

- Many applications generate huge amounts of data
- Need to be processed on the fly
- Data too abundant to store all/compute exactly





Introduction

- Processing this data becomes a challenge

Requirements:

- Results generated on the fly
- Memory should scale sub-linear
- Constant-time processing of incoming data

- Note: only distributed computing does not solve the problem; 1000 computers can speed up a computation at most 1000 times

Lecture outline

Frequent items

Filtering

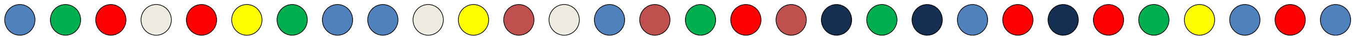
Distinct Counting

Similarity search

Frequent Items (also called Heavy Hitters)

Warming up problem: Frequent Items

- “Identify all frequent items that occur more than θN times in a stream S of N items” (θ in [0,1] or in %)

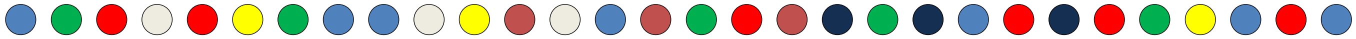


- Items arrive one by one; what we do not store will be inaccessible later on.
- How much space needed for an *exact* solution?

Frequent Items

- Counting every item is impossible
  - E.g., all pairs of people that phone to each other
- We do not know up front which combinations will be frequent

Example:



30 items; 🔵:8, 🔴:6, 🟢:5

All others are 3

If frequency θ is 20%: 🔵 and 🔴 need to be output

Frequent Items – Worst Case

Question: How much space needed for an *exact* solution?

Answer: worst case at least linear in n, to be exact Ω(n log(N/n)) bits, where n = number of possible items, N = length of the total stream.

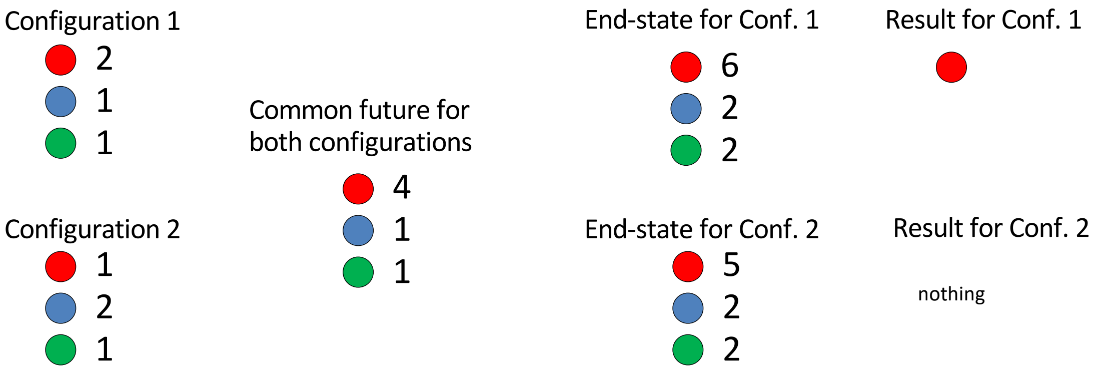
Sketch of Proof (for full proof: see https://www.cs.umd.edu/~samir/498/karp.pdf)

Let θ = 50%, and suppose we have already seen N/2 - 1 elements of the stream.

Configuration: the (item, count) pairs seen so far.

For any two distinct configurations (e.g., top and bottom in figure) the algorithm needs to be able to discern between them, i.e., be in a different memory state. Why? Otherwise, there can be a common future that gives different results per configuration. The algorithm would then be wrong in at least one of the two configurations.

Hence, we need at least log(#configurations) space. Why?



Configuration 1

🔴 2
🔵 1
🟢 1

Configuration 2

🔴 1
🔵 2
🟢 1

Common future for both configurations

🔴 4
🔵 1
🟢 1

End-state for Conf. 1

🔴 6
🔵 2
🟢 2

End-state for Conf. 2

🔴 5
🔵 2
🟢 2

Result for Conf. 1

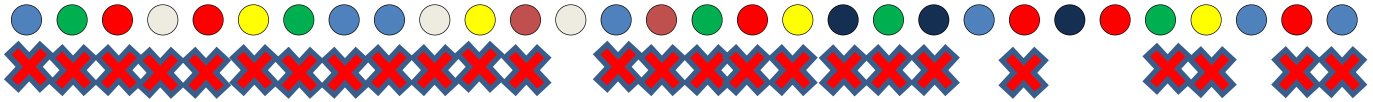
🔴

Result for Conf. 2

nothing

Frequent Items – Set Solution

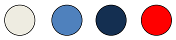
- “Given a stream, identify all frequent items that occur more than 20% of the time”



- Remove 5 elements of a different color to get S’:

  If ● was a frequent item, it still is!

- Hence, removing 5 elements of different color gives us a smaller set, but we keep all frequent items.
  - Can be done repeatedly
  - Until no longer possible to remove 5 with distinct color
- Answer is a subset of the remaining (at most 4) colors:



Frequent Items – Stream Implementation

- "Identify all frequent items that occur more than θN times in a stream S of N items" (θ in [0,1])

  - Summary={}
  - For each item ● that arrives:
    - If (●, count) is in Summary:
      update count to count + 1
    - Else:
      add (●, 1) to Summary
    - If |Summary| $\geq 1/\theta$:
      decrease the count of all pairs in Summary
      remove all pairs with count = 0

Frequent Items - Summary

Algorithm by *Karp et al.*

- Problem:
  - Find all items exceeding frequency θN
- Space:
  - O(1 / θ) counters
  - Counters are log(N) bits each => O(1/ θ log(N)) space
- Time per update: O(1) *
- **Concession:**
  - **False positives or 2-pass**

* Karp et al. propose a data structure that allows O(1) in worst case

Solution 2: Lossy Counting of Frequencies

- What if:
  - We want to have frequencies
  - And bound on false positives

- Lossy counting algorithm by Manku et el.

We start with a simplified version and gradually extend it.

Manku, Gurmeet Singh, and Rajeev Motwani. "Approximate frequency counts over data streams." *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 2002.

Lossy Counting of Frequencies

- Following algorithm finds *superset* of θ-frequent items:
  - Initialization: none of the items has a counter
  - Item ● enters at time t:
    - If ● has a counter: count(●) ++
    - Else:
      - count(●) = 1
      - start(●) = t
    - For all other items △ with a counter do:
      - If count(△) / (t – start(△) + 1) < θ :
        - » Delete Counter for △
- Query time: return all items that have a counter

Lossy Counting of Frequencies

- Example: (20%)

●

	start	# (freq)
●	1	1 (100%)

Lossy Counting of Frequencies

- Example: (20%)

  🔵 🟢

	start	# (freq)
🔵	1	1 (50%)
🟢	2	1 (100%)

Lossy Counting of Frequencies

- Example: (20%)

  🔵 🟢 🔴 ⚪ 🔴

	start	# (freq)
🔵	1	1 (20%)
🟢	2	1 (25%)
🔴	3	2 (66%)
⚪	4	1 (50%)

Lossy Counting of Frequencies

- Example: (20%)

🔵 🟢 🔴 ⚪ 🔴 🟡

	start	# (freq)
🔵	1	1 (17%)
🟢	2	1 (20%)
🔴	3	2 (50%)
⚪	4	1 (33%)
🟡	6	1 (100%)

Lossy Counting of Frequencies

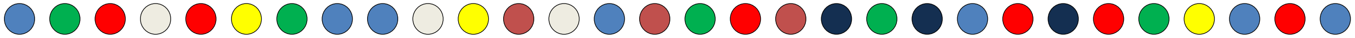
- Example: (20%)

  🔵 🟢 🔴 ⚪ 🔴 🟡 🟢 🔵 🔵

	start	# (freq)
🟢	2	2 (25%)
🔴	3	2 (29%)
🟡	6	1 (25%)
🔵	8	2 (100%)

Lossy Counting of Frequencies

- Example: (20%)

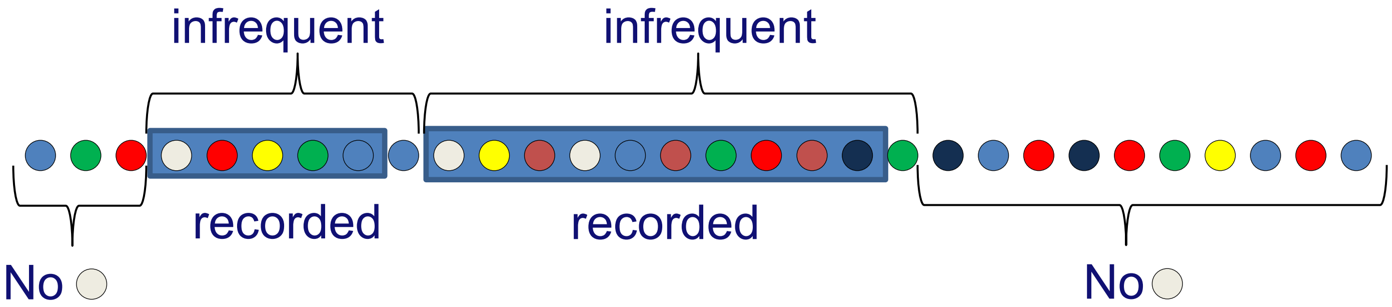


	start	# (freq)
🟢	2	1 (25%)
🔴	17	4 (29%)
🟡	27	1 (25%)
🔵	8	6 (26%)
⚫	19	3 (25%)

Lossy Counting - Correctness

- Why does it work?
  - If ○ is not recorded, ○ is not frequent in the stream

- Imagine marking when ○ was recorded:
  - If ○ occurs, recording starts
  - Only stopped if ○ becomes infrequent since start recording



- Whole stream can be partitioned into parts in which ○ is not frequent ➔ ○ is not frequent in the whole stream

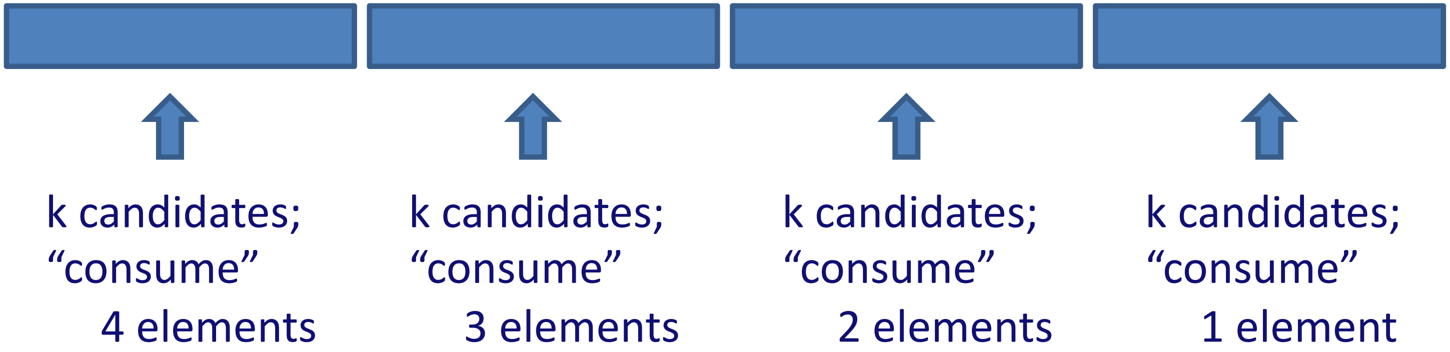
Lossy Counting – Space Requirements

- Let N be the length of the stream
- θ minimal frequency threshold. Let k=1/θ

- Item a is in the summary if:
  - a appears once among last k items
  - a appears twice among last 2k items
  - …
  - a appears x times among last xk items
  - …
  - a appears θN times among last N items

Lossy Counting – Space Requirements

- Divide stream in blocks of size k = 1/θ



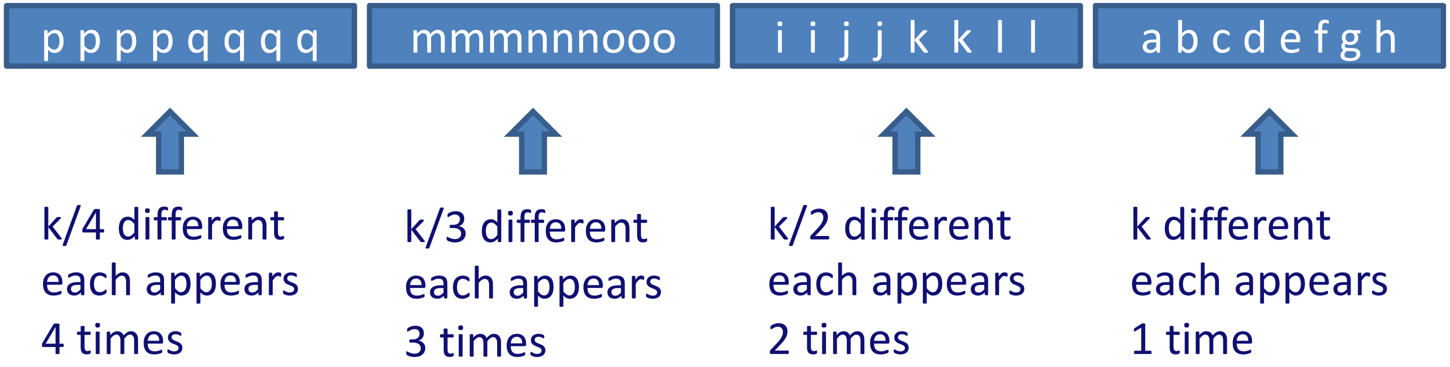
k candidates; “consume” 4 elements

k candidates; “consume” 3 elements

k candidates; “consume” 2 elements

k candidates; “consume” 1 element

- Constellation with maximum number of candidates:



p p p p q q q q

mmmnnnooo

i i j j k k l l

a b c d e f g h

k/4 different each appears 4 times

k/3 different each appears 3 times

k/2 different each appears 2 times

k different each appears 1 time

Lossy Counting – Space Requirements

- Hence total space requirement worst case:

  $\Sigma_{i=1\ldots N/k}$ k/i $\approx$ k log(N/k)

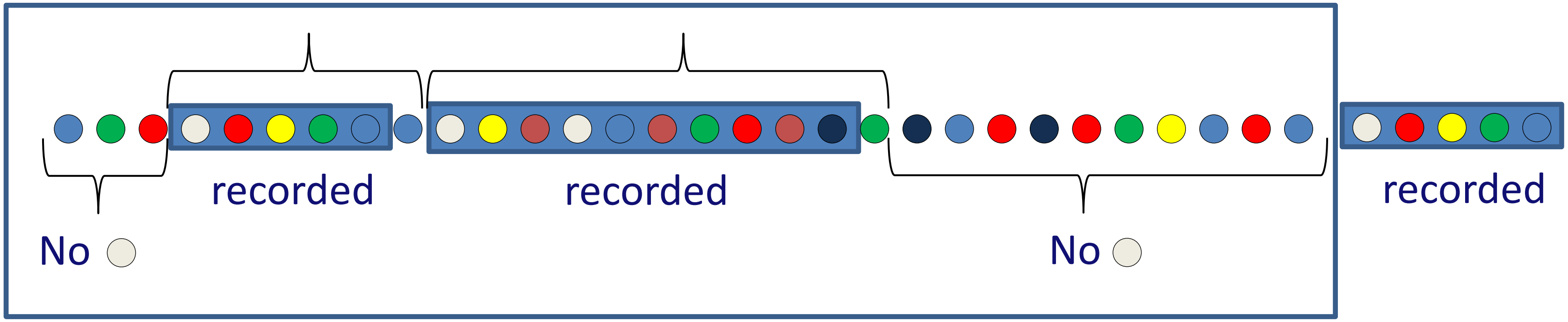
- Recall: k = 1/θ
- Worst case space requirement:

  1/θ log(Nθ) candidates;

- O(1/θ log(Nθ)) counters to maintain

Lossy Counting – Guarantee

- Run the algorithm with $\varepsilon < \theta$ as threshold
- Guaranteed: at any point in time, the true frequency of ○ is in the interval [count(○)/N , count(○)/N+ε]



Less than εN occurrences of ○

- Report all items ○ with count(○) $\geq (\theta - \varepsilon)$ N
  - All items reported have frequency at least $\theta - \varepsilon$
  - All items with frequency θ are reported

Frequent Items - Summary

Karp’s algorithm:

- $O(1/\theta)$ counters
- No false negatives, but may have false positives

Lossy Counting:

- $1/\varepsilon \log(N\varepsilon)$ space ***worst case*** *(usually much better!)*
- Maximum error of ε on counts
- No false negatives, only false positives in the range $[\ \theta - \varepsilon, \theta\ ]$

- There exist many other algorithms (e.g., CM-sketch)

Frequent Items - Applications

- Automatically block IP-traffic between pairs of addresses taking up more than 1% of the bandwidth

<u>using lossy counting:</u>

- Set threshold to 1.1% (ε=0.001, θ=0.01)
- 1000 log(N/1000) counters worst case
    - 6000 counters for 1,000,000,000 stream length
- Constant time per item
- Can be implemented inside a router

Frequent Items - Applications

- Automatically block IP-traffic between pairs of addresses taking up more than 1% of the bandwidth

  using lossy counting:
  - Set threshold to 1.1% (ε=0.001, θ=0.01)
  - 1000 log(N/1000) counters worst case
    - 6000 counters for 1,000,000,000 stream length
  - Constant time per item
  - Can be implemented inside a router

- Give all words with a frequency of more than 0.01% in a collection of books
  - Karp: 2 scans maintaining 10,000 strings

Filtering

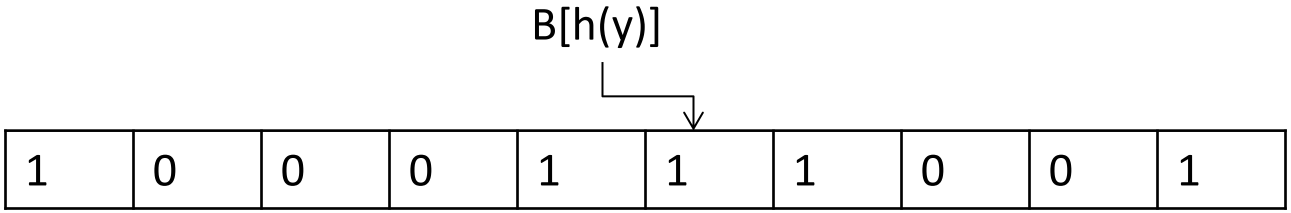
Filtering

- Suppose we are not interested in all stream elements, but only in a subset
  - Telecom network: not all calls, but only calls of a group of users having a specific tariff plan
  - Internet packages: traffic to certain suspicious websites
  - Only allow mails to a large whitelist of email addresses

- Stream is possibly captured in a distributed way and selection may be periodically changing
  - Think about our blacklisted pairs of IP addresses

Filtering

- Problem setting:
  - Stream of items x, target set A of items
  - Only keep x if x is in A
- Depending on size of A, keeping the whole set of keys for filtering may take too much memory
- Use a bitstring B of length m instead, and a hash function h(x) of modulo m
- The i-th bit B[i] is set to 1 if there is an x in A such that h(x) = i; we denote this as B[h(x)]=1
- Then, we test membership of y in A by checking B[h(y)]
- What can we say for the possible outcomes?



B[h(y)]

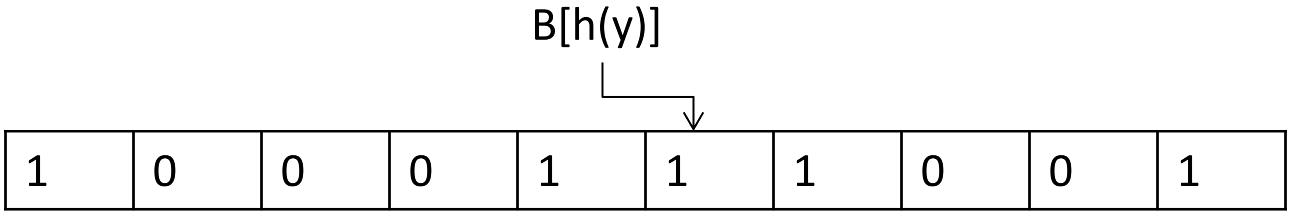
1	0	0	0	1	1	1	0	0	1

Filtering: Analysis

- No false negatives
  - If y is in A, the bit B[h(y)] will be 1
- There may be **false positives**, though:
  - if $\boldsymbol{x} \in \boldsymbol{A}$, but $\boldsymbol{y\ not\ in\ A}$, and B[h(x)]=B[h(y)], then the test y in A will incorrectly yield True

To calculate the **false positive** probability, we reason as follows.

Let m = length of bitstring, n = number of elements in A



1	0	0	0	1	1	1	0	0	1

Probability that a single element a in A leaves B[h(y)] untouched

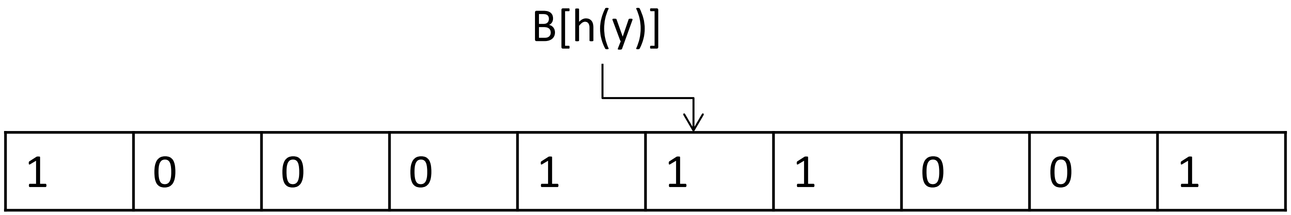
$$\frac{m-1}{m}$$

Filtering: Analysis

- No false negatives
  - If y is in A, the bit B[h(y)] will be 1
- There may be **false positives**, though:
  - if $\boldsymbol{x} \in \boldsymbol{A}$, but $\boldsymbol{y}\ \boldsymbol{not\ in\ A}$, and B[h(x)]=B[h(y)], then the test y in A will incorrectly yield True

To calculate the **false positive** probability, we reason as follows.

Let m = length of bitstring, n = number of elements in A



B[h(y)]

1	0	0	0	1	1	1	0	0	1

Probability that every element a in A leaves B[h(y)] untouched

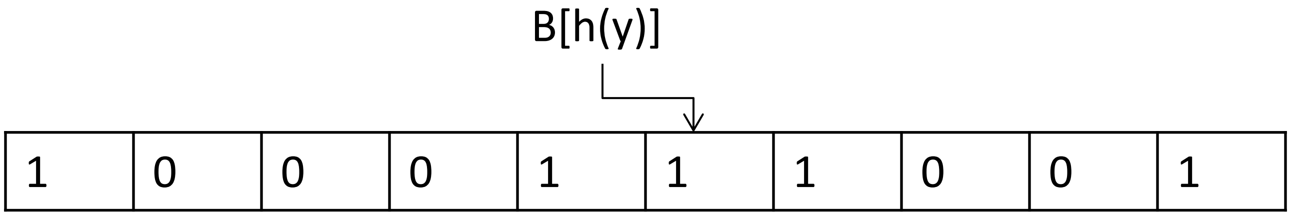
$$\left(\frac{m-1}{m}\right)^{n}$$

Filtering: Analysis

- No false negatives
  - If y is in A, the bit B[h(y)] will be 1
- There may be **false positives**, though:
  - if $\boldsymbol{x \in A}$, but $\boldsymbol{y\ not\ in\ A}$, and B[h(x)]=B[h(y)], then the test y in A will incorrectly yield True

To calculate the **false positive** probability, we reason as follows.

Let m = length of bitstring, n = number of elements in A



B[h(y)]

1	0	0	0	1	1	1	0	0	1

Probability that some element in A sets B[h(y)] to 1:

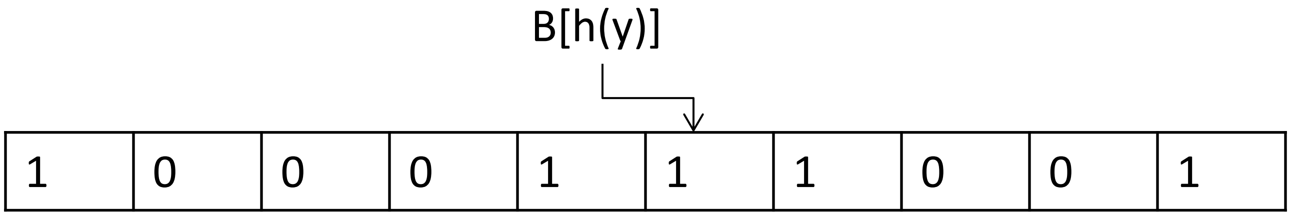
$$1 - \left(\frac{m-1}{m}\right)^{n}$$

Filtering: Analysis

- No false negatives
  - If y is in A, the bit B[h(y)] will be 1
- There may be **false positives**, though:
  - if $\boldsymbol{x \in A}$, but $\boldsymbol{y\ not\ in\ A}$, and B[h(x)]=B[h(y)], then the test y in A will incorrectly yield True

To calculate the **false positive** probability, we reason as follows.

Let m = length of bitstring, n = number of elements in A



B[h(y)]

1	0	0	0	1	1	1	0	0	1

Probability that some element in A sets B[h(y)] to 1:

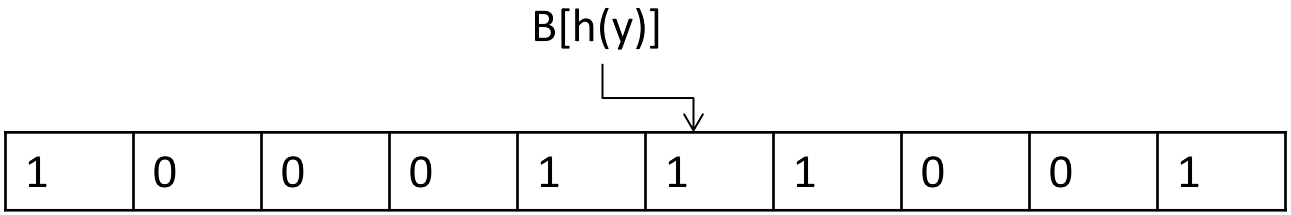
$$P[FP] = 1 - \left(\frac{m-1}{m}\right)^n$$

Filtering: Analysis

- No false negatives
  - If y is in A, the bit B[h(y)] will be 1
- There may be **false positives**, though:
  - if $\boldsymbol{x} \in \boldsymbol{A}$, but $\boldsymbol{y\ not\ in\ A}$, and B[h(x)]=B[h(y)], then the test y in A will incorrectly yield True

To calculate the **false positive** probability, we reason as follows.

Let m = length of bitstring, n = number of elements in A



B[h(y)]

1	0	0	0	1	1	1	0	0	1

Probability that some element in A sets B[h(y)] to 1:

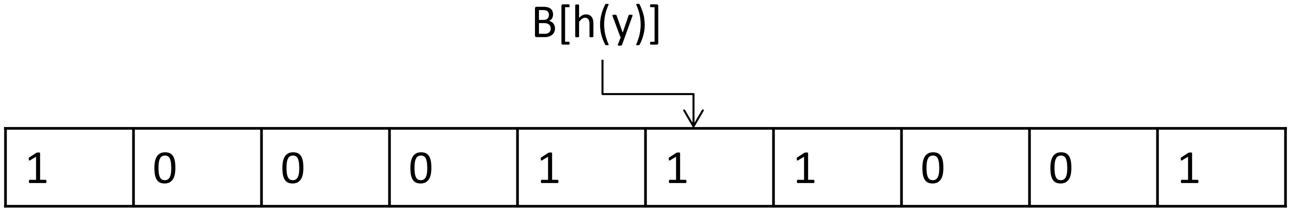
$$P[FP] = 1 - \left(\frac{m-1}{m}\right)^{n} = 1 - \left(1 - \frac{1}{m}\right)^{m\frac{n}{m}}$$

Filtering: Analysis

- No false negatives
  - If y is in A, the bit B[h(y)] will be 1
- There may be **false positives**, though:
  - if $\boldsymbol{x} \in \boldsymbol{A}$, but $\boldsymbol{y\ not\ in\ A}$, and B[h(x)]=B[h(y)], then the test y in A will incorrectly yield True

To calculate the **false positive** probability, we reason as follows.

Let m = length of bitstring, n = number of elements in A



B[h(y)]

1	0	0	0	1	1	1	0	0	1

$$\left(1-\frac{1}{m}\right)^{m} \approx e^{-1}$$

Probability that some element in A sets B[h(y)] to 1:

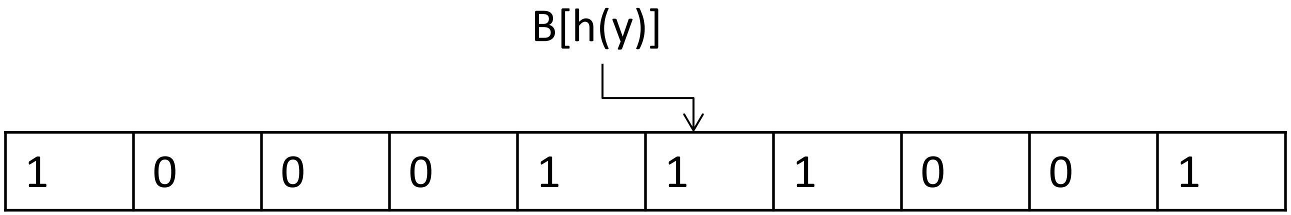
$$P[FP] = 1 - \left(\frac{m-1}{m}\right)^{n} = 1 - \left(1-\frac{1}{m}\right)^{m\frac{n}{m}}$$

Filtering: Analysis

- No false negatives
  - If y is in A, the bit B[h(y)] will be 1
- There may be **false positives**, though:
  - if $\boldsymbol{x} \in \boldsymbol{A}$, but $\boldsymbol{y\ not\ in\ A}$, and B[h(x)]=B[h(y)], then the test y in A will incorrectly yield True

To calculate the **false positive** probability, we reason as follows.

Let m = length of bitstring, n = number of elements in A



B[h(y)]

1	0	0	0	1	1	1	0	0	1

Probability that some element in A sets B[h(y)] to 1:

$$P[FP] = 1 - \left(\frac{m-1}{m}\right)^n = 1 - \left(1 - \frac{1}{m}\right)^{m\frac{n}{m}} \approx 1 - e^{-\frac{n}{m}}$$

Filtering: Analysis

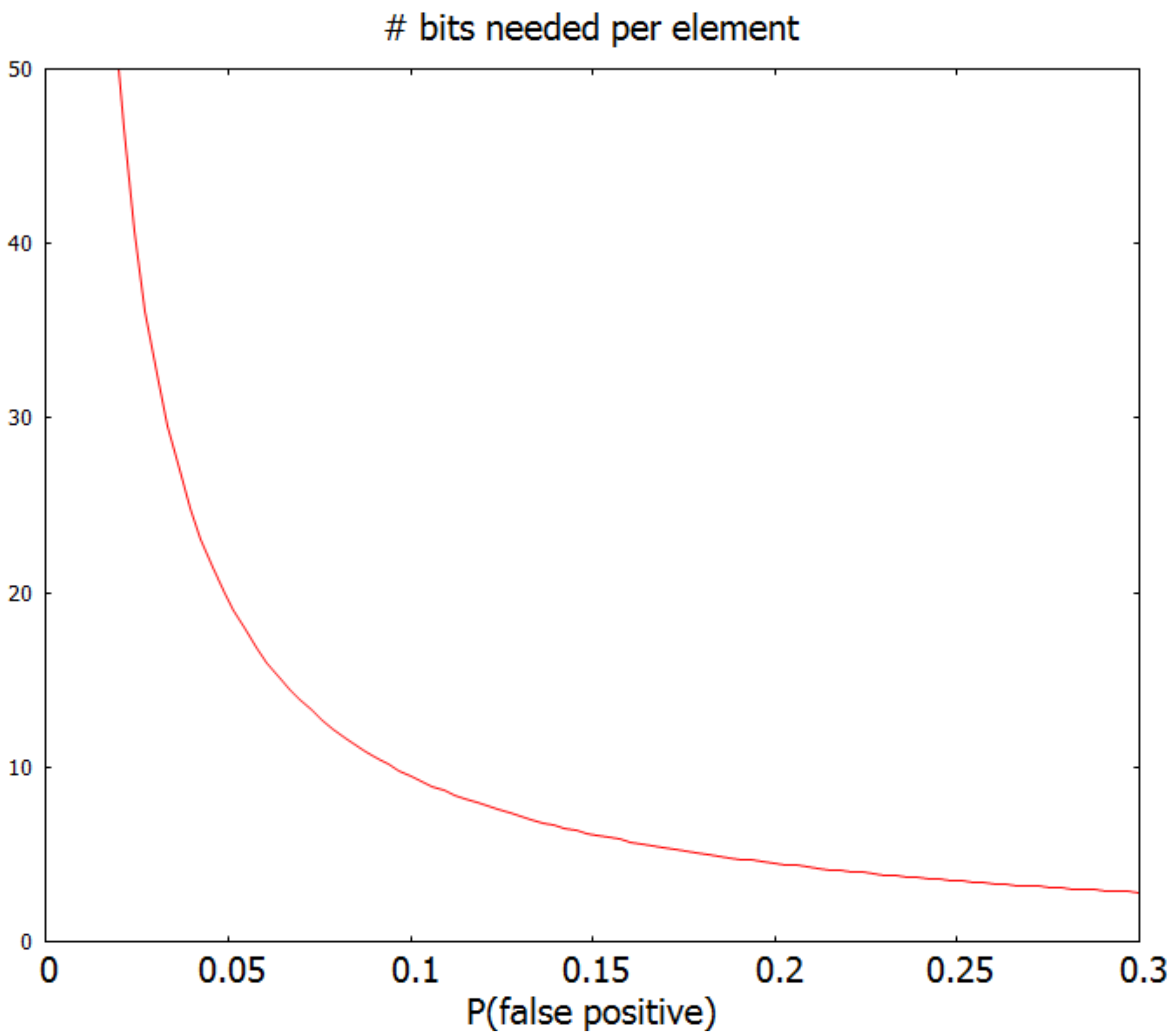
- No false negatives
  - If y is in A, the bit B[h(y)] will be 1
- There may be **false positives**, though:
  - if $\boldsymbol{x} \in \boldsymbol{A}$, but $\boldsymbol{y\ not\ in\ A}$, and B[h(x)]=B[h(y)], then the test y in A will incorrectly yield True

Let m = length of bitstring, n = number of elements in A

$$P[FP] = 1 - \left(\frac{m-1}{m}\right)^{n} = 1 - \left(1 - \frac{1}{m}\right)^{m.\frac{n}{m}} \approx 1 - e^{-\frac{n}{m}}$$

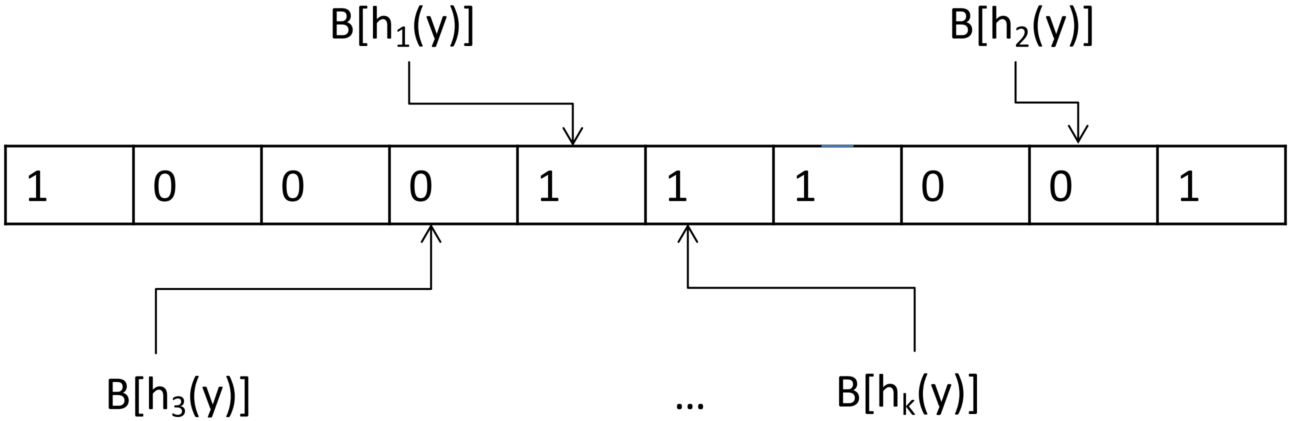
Hence, if we want to reduce the probability of a false positive to ε, we will need m= $-\frac{n}{\ln(1-\varepsilon)}$ bits in our bitstring.

Filtering: Analysis



Bloom Filter

- We can do much better, however
  - Using multiple (k) hash functions; an element a $\in A$ will set all bits B[h_j(a)]



- Membership test for $y \in A$:
  Only return true if *all* bits are 1.

Bloom Filter: Analysis

- For k=1 we had: $P[FP|k=1] \approx 1 - e^{-\frac{n}{m}}$
  We need 1 in every of the k positions:

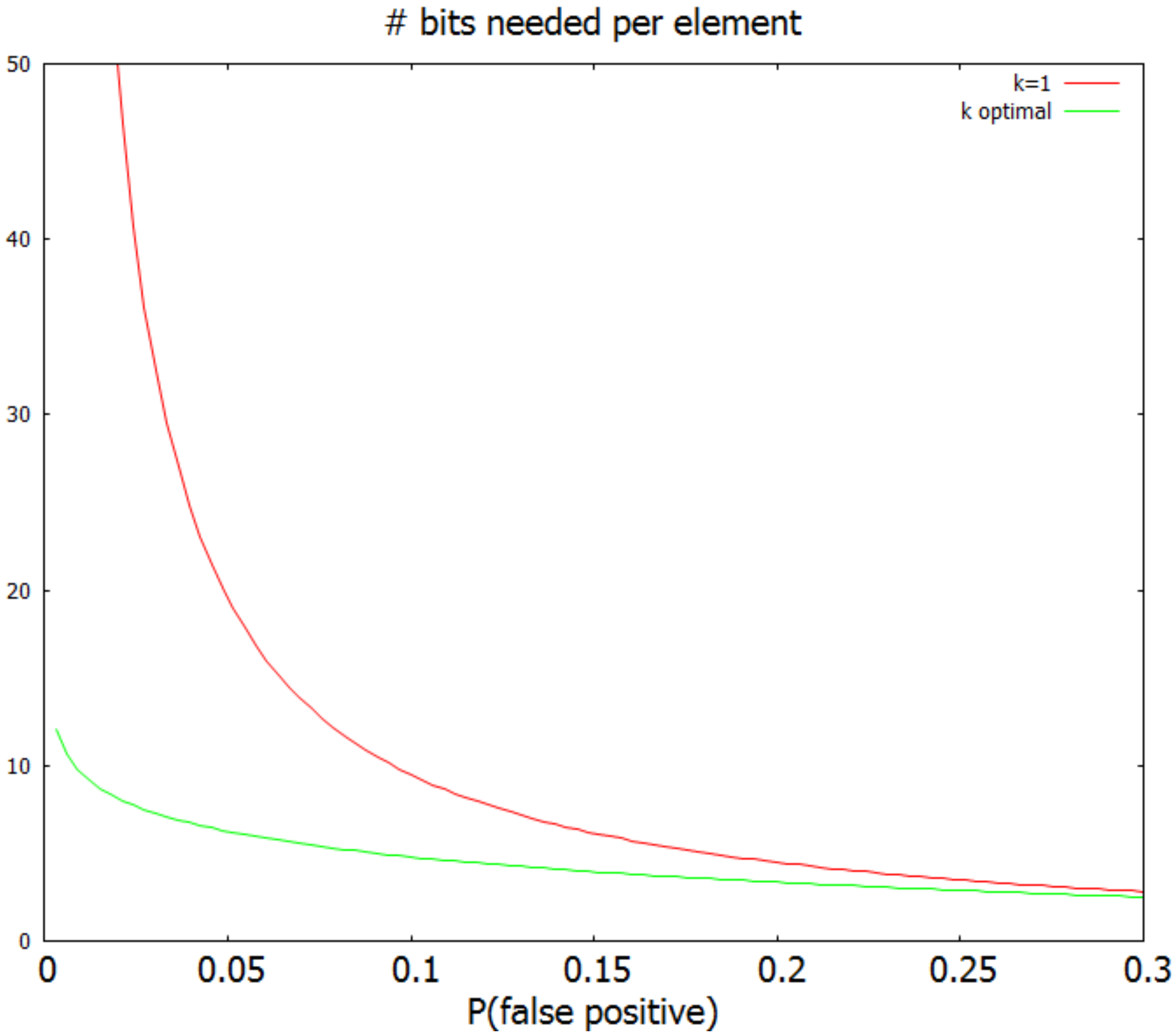
$$P[FP] \approx \left(1 - e^{-\frac{kn}{m}}\right)^{k}$$

Assumes independent probabilities and k different hash values (realistic if k<<m)

- It can be shown that if we want to reduce the probability of a false positive to ε, we will need:

$$m = \frac{-n\ln(\varepsilon)}{\ln(2)^2} \text{ and } \mathrm{k} = -\frac{\ln(\varepsilon)}{\ln(2)} = \log_2\left(\frac{1}{\varepsilon}\right)$$

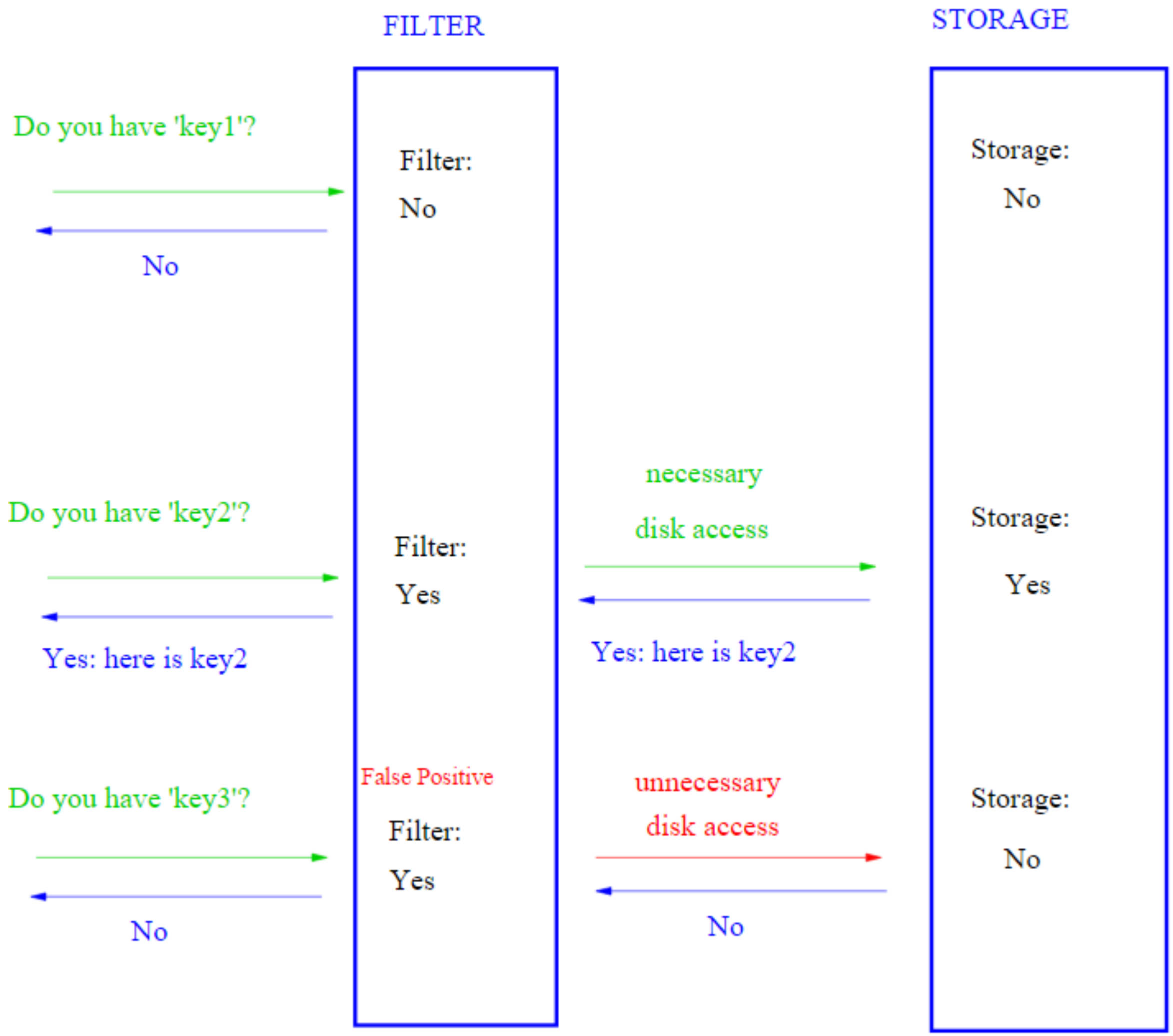
Bloom Filter: Analysis



Bloom Filters: Usage

- Bloom filters are great for filtering, especially in a distributed setting
  - Google Chrome: Bloom filters to filter malicious URLs;
    - No blacklists being distributed, only Bloom filter
    - Only in case of a hit, full check of URL
  - Google Bigtable and Apache Cassandra: Bloom filters to avoid costly disk lookups for non-existing keys

Bloom Filters: avoid costly disk lookups



"Bloom filter speed" by Alexmadon at English Wikipedia
http://commons.wikimedia.org/wiki/File:Bloom_filter_speed.svg#/media/File:Bloom_filter_speed.svg

Maintaining Large Sets

- Bloom filters are great for filtering
- However, they are usually *not* suitable for:

```
Item $a$ arrives in the stream:
    add $a$ to bloom filter B
Use B to compute property of A = { a | a seen in stream }
```

- Memory requirements of a Bloom filter grow *linear* with size of the set A
  - Bloom filter more suitable to represent medium-size set over a huge domain of possible values
    - E.g., set of blacklisted IP-addresses

Distinct Counting

Counting number of distinct items

Problem: give the number of distinct (unique) items in a stream.

- Highly useful property:
  - Number of distinct visitors to a website
  - Estimate cardinality of projecting a relation onto a subset of its attributes

Neither heavy hitters nor filters are suited for counting the number of distinct items in a stream

Distinct Counting

How many people attend my presentation?

Distinct Counting: Attempt 1a

```
S={}
N=0
**Whenever** a person P enters the room:
        if P not in S:
                $S = S \cup \{ P \}$
                N+=1
```

Exact algorithm

Complexity:

- **Space** O(N len(identifier of P))
- **Time per person entry:** log(N)

How many people attend my presentation?

N is the number of distinct people

Distinct Counting: Attempt 1b

How many people attend my presentation?

```
S={}
N=0
Whenever a person P enters the room:
    if h(P) not in S:
        S = S ∪ { h(P) }
        N+=1
```

h(P) denotes hash of identifier of P

Near exact algorithm if range(h) large enough

Complexity:

- **Space** O(N log(N))
- **Time per person entry** O(log(N))

Distinct Counting

This solution is not satisfactory at all:

- Space N log(N) is completely unacceptable
- Time log(N) per entry is barely acceptable

We will introduce an alternative: Hyperloglog sketch

- Space log(log(N))
- Constant update time
- But approximate

HLL is based on the idea of Flajolet-Martin (FM) sketches

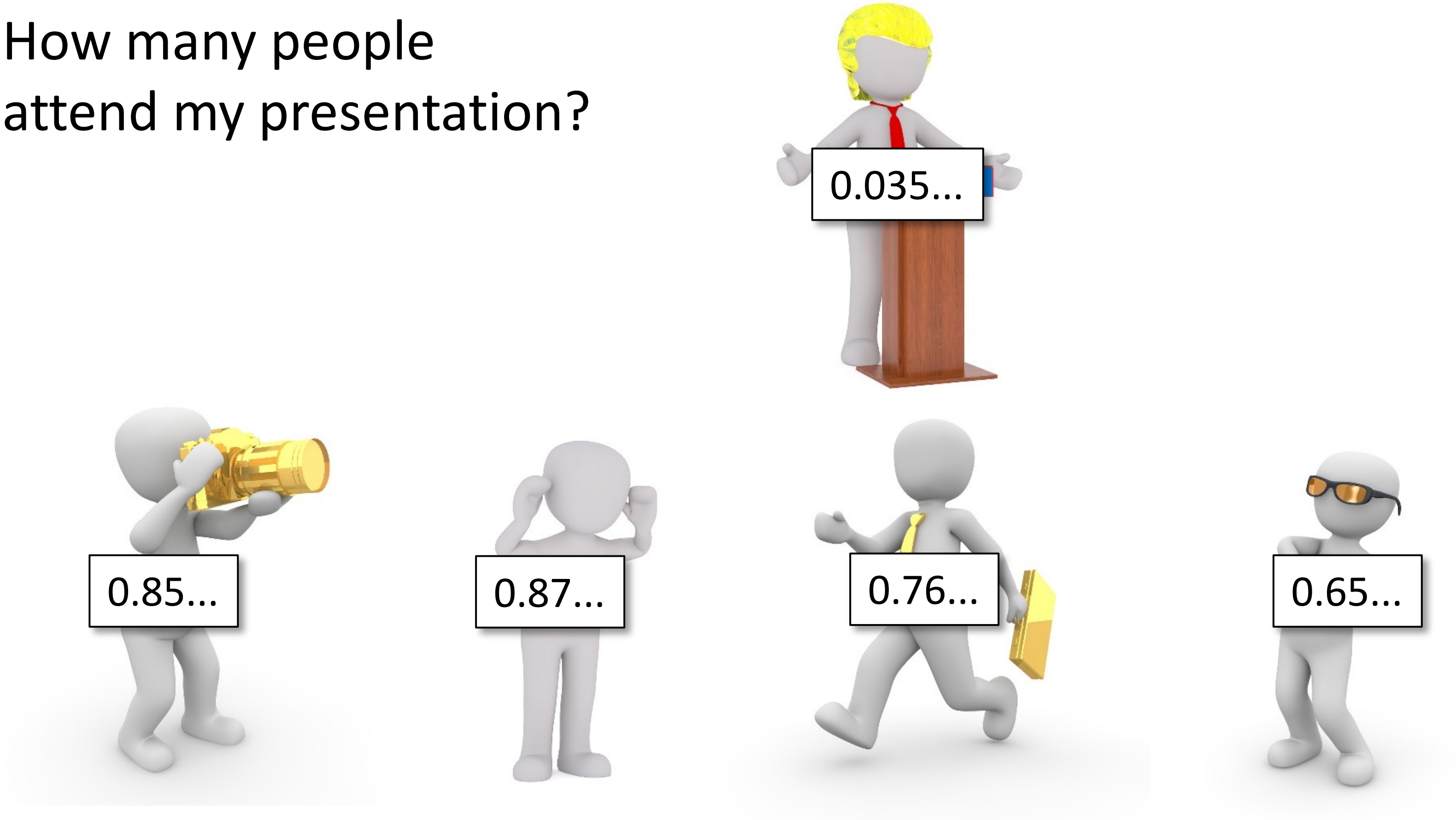
Flajolet-Martin sketches: main idea

How many people
attend my presentation?

- Pick, at random, a hash function that assigns to every person on earth a number between 0 and 1.

Flajolet-Martin sketches: main idea

How many people
attend my presentation?



- Compute the number for everyone entering the room
- Maintain the maximum over all numbers seen, call this S

Flajolet-Martin sketches: main idea

- What do we know about this largest number S?
- It only depends on the *number* of elements, not on how many times they entered:

  $\max\{h(P_1),h(P_1),h(P_1),h(P_2)\} = \max\{h(P_1), h(P_2)\}$

- The higher the number of elements N, the higher S will be *in expectation*

$$P(S \leq x) = x^N$$

$$E[S] = \int_0^1 x\, N\, x^{N-1}\, dx = \frac{N}{N+1}$$

- We can reverse-engineer, and estimate N from seeing S as:

$$N = \frac{S}{1-S}$$

Flajolet-Martin sketches: main idea

- There are still a number of issues, though:
- **First issue: S can be far away from E[S]**
  - Accidentally having one person with a high hash number may lead to huge overestimations
  - Use multiple hash functions instead
    - k independent hash functions $h_1, \ldots, h_k$ give $S_1, \ldots, S_k$
    - compute the mean; this has lower variance

$$Var\left(\frac{\sum_{i=1}^{k} S_i}{k}\right) = \frac{Var(S)}{k}$$

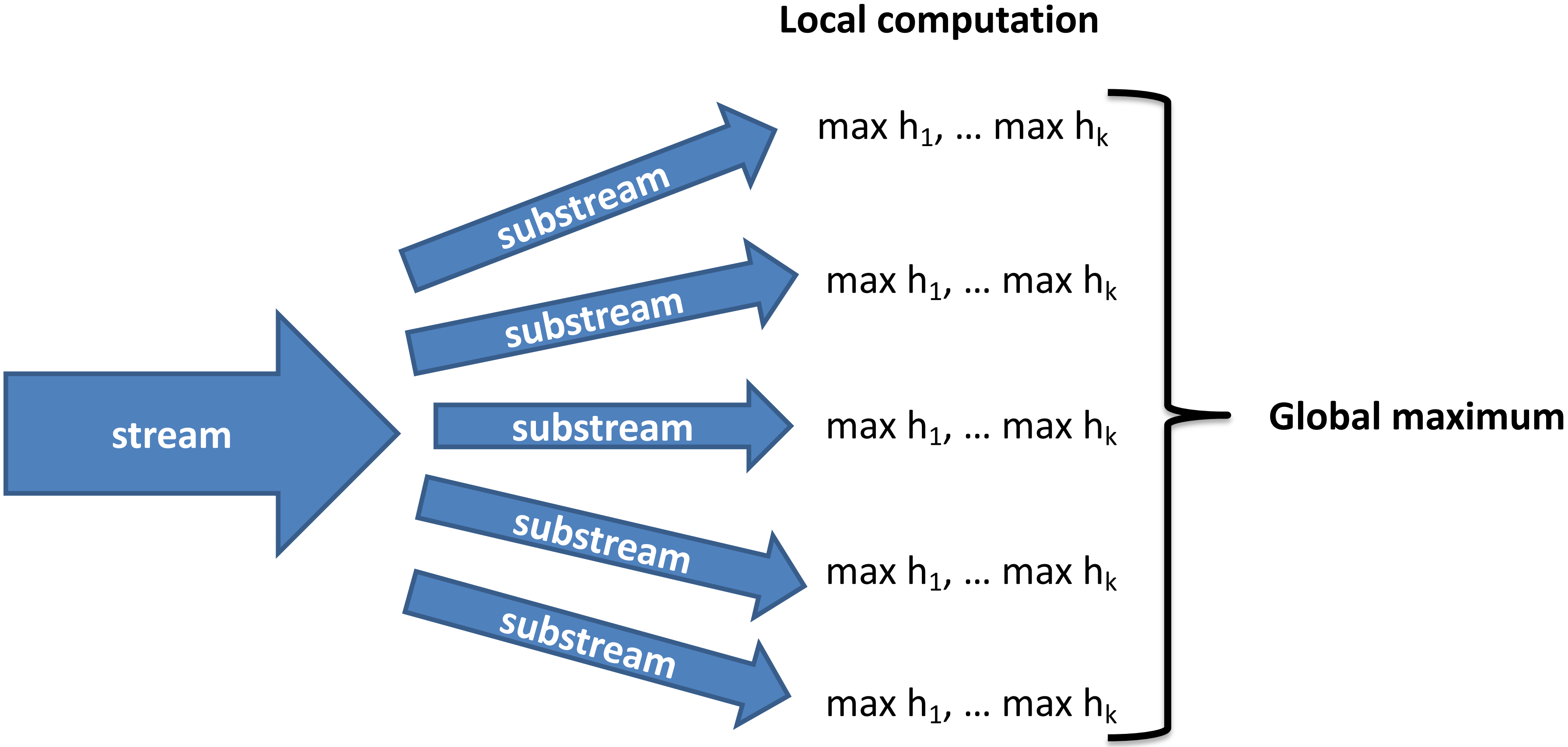
Flajolet-Martin sketches: main idea

- **Second issue: for large N, the quantity S will quickly become indistinguishable from 1**
- Flajolet-Martin uses the following solution:
  - Take the binary representation of h(x) 110101000
  - Look at the number of 0's in the tail 3
  - Keep the *maximum R* of the number of 0s in the tail

- Probability of finding a tail of *r* zeros:
  - Goes to 1 if $\mathrm{N} \gg 2^r$
  - Goes to 0 if $\mathrm{N} \ll 2^r$
- Thus, 2^R will almost always be around *N*

FM-Sketches are easy to parallelise

- No problem; easily parallelizable
  - max (max(A), max(B)) = max(A$\cup$B)



Local computation

stream

substream

substream

substream

substream

substream

max h_1, … max h_k

max h_1, … max h_k

max h_1, … max h_k

max h_1, … max h_k

max h_1, … max h_k

Global maximum

Variant: HyperLogLog Sketch

- Workhorse when it comes to cardinality counting
- Avoids the need for many hash-functions to reduce error
  - Use first bits of hash-function to split stream
  - Use last bits to maintain FM-sketch of substream
- Standard error $\frac{\mathbf{1.04}}{\sqrt{\boldsymbol{m}}}$ (m is size of summary)
  - Independent of stream size
  - Hence, log(log(N)) dependence on stream length

Demo at: http://content.research.neustar.biz/blog/hll.html

Similarity search

Similarity Search

A very common operation:

- Find similar customers
- Find similar documents (e.g. Plagiarism checker)

- Locality Sensitive Hashing is a well-known technique to quickly find near-duplicates
- We will illustrate the principle for the Jaccard-Coefficient which measures the distance between sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Jaccard Coefficient

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Indicates how similar the sets A and B are.

Example:

J({a,b,c},{c,d}) = 1/4

J({a,b,c},{b,c,d}) = 2/4

Used, e.g., to detect near duplicates

A set of n-grams in document 1

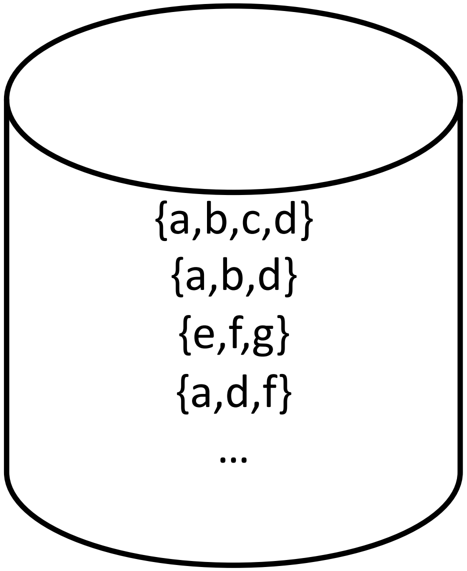
B set of n-grams in document 2

Similarity Search – Jaccard Index

Example:

Recommendation needs to be made for a user U

- We characterize users by the set of items they bought
- Find users who bought similar items
- Recommend items that were bought by these users



{a,b,c,d}
{a,b,d}
{e,f,g}
{a,d,f}
…

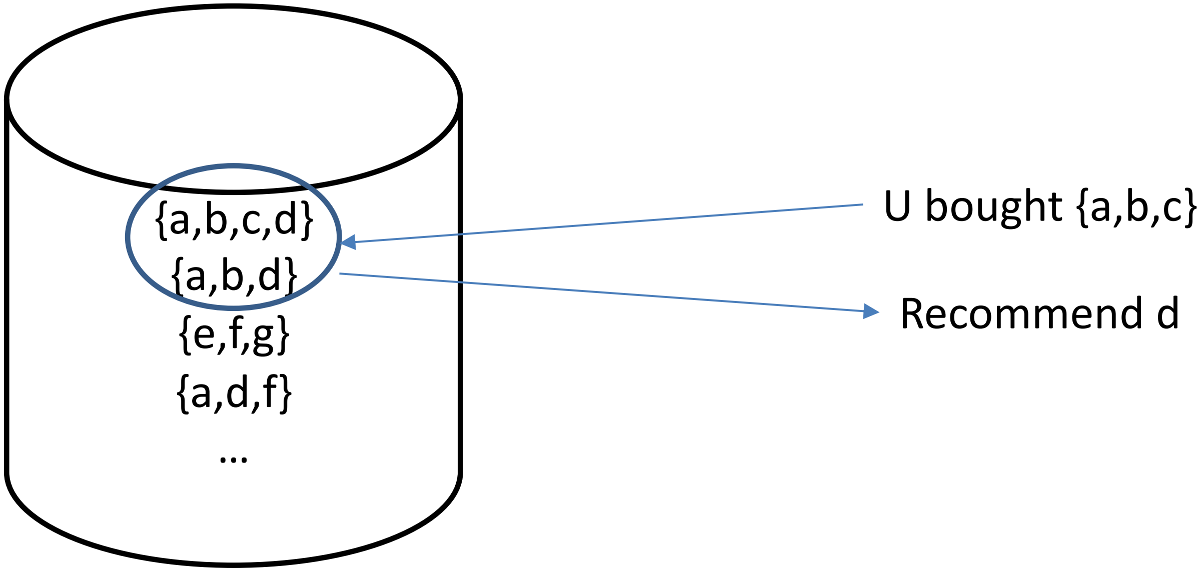
U bought {a,b,c}

Similarity Search – Jaccard Index

Example:

Recommendation needs to be made for a user U

- We characterize users by the set of items they bought
- Find users who bought similar items
- Recommend items that were bought by these users



{a,b,c,d}
{a,b,d}
{e,f,g}
{a,d,f}
…

U bought {a,b,c}

Recommend d

Similarity Search: Naïve Algorithm

```
For each user U’ in DB:
    compute sim := J(items(U),items(U’))
    if sim ≥ threshold:
        return items(U’)
```

Complexity: |DB|

For high-dimensional queries indexing methods such as inverted indices are no longer efficient

We need another indexing mechanism

Jaccard Coefficient

Let A, B be subsets of universal set V

h is a function mapping elements of V to {1,2,…,|V|}

Example: d → 1, c→ 2, a → 3, b → 4

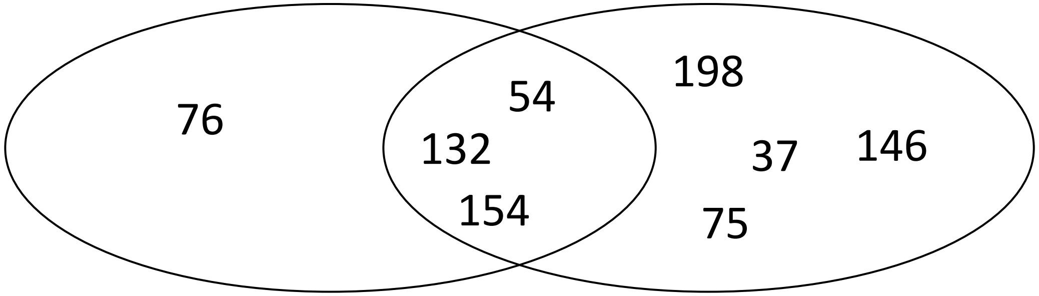
Let *minhash* of A be $\min_h(A) := \min_{a \in A} h(a)$

$$\Pr[\ \min_h(A) = \min_h(B)\ ]$$

$$= \ |A \cap B| \,/\, |A \cup B|$$

$$= \ J(A,B)$$

* We implicitly assume range(h) >> |A|, |B|



Locality-Sensitive Hashing

We call such a function $\min_h$ *locality-sensitive* for Jaccard

Independent locality-sensitive functions can be combined

Independent functions $h_1, \ldots, h_m$

"signature" of set A:

$|A|$ and vector ($\min_{h1}(A), \min_{h2}(A), \ldots, \min_{hm}(A)$)

Estimating J(A,B):

$(a_1, \ldots, a_m)$ vector for A $(b_1, \ldots, b_m)$ vector for B

Let $e = \# \{ i \mid a_i=b_i \}$

e / m is an estimator for J(A,B)

Jaccard Coefficient

Example: V = { a, b, c, d, e }

A = { a, b }

B = { b, c, d }

C = { a, b, c, e }

	h_1	h_2	h_3	h_4
a	1	2	5	2
b	2	5	2	4
c	3	1	4	5
d	4	4	1	3
e	5	3	3	1

A	1	2	2	2
B	2	1	1	3
C	1	1	2	1

J(A,B) = 1/4 ; estimate: 0

J(A,C) = 1/2 ; estimate: 1/2

J(B,C) = 2/5 ; estimate: 1/4

Locality-Sensitive Hashing

We will first illustrate the principle for Jaccard Index

MinHashing to create signatures of sets

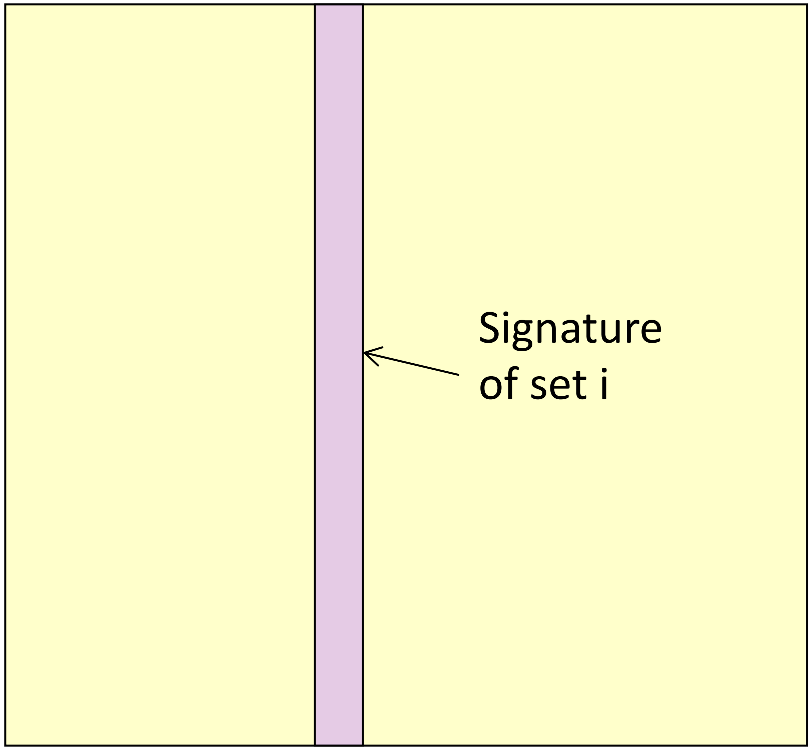
A → [123, 235, 576, 67, 56]

B → [123, 3456, 56, 67, 867]

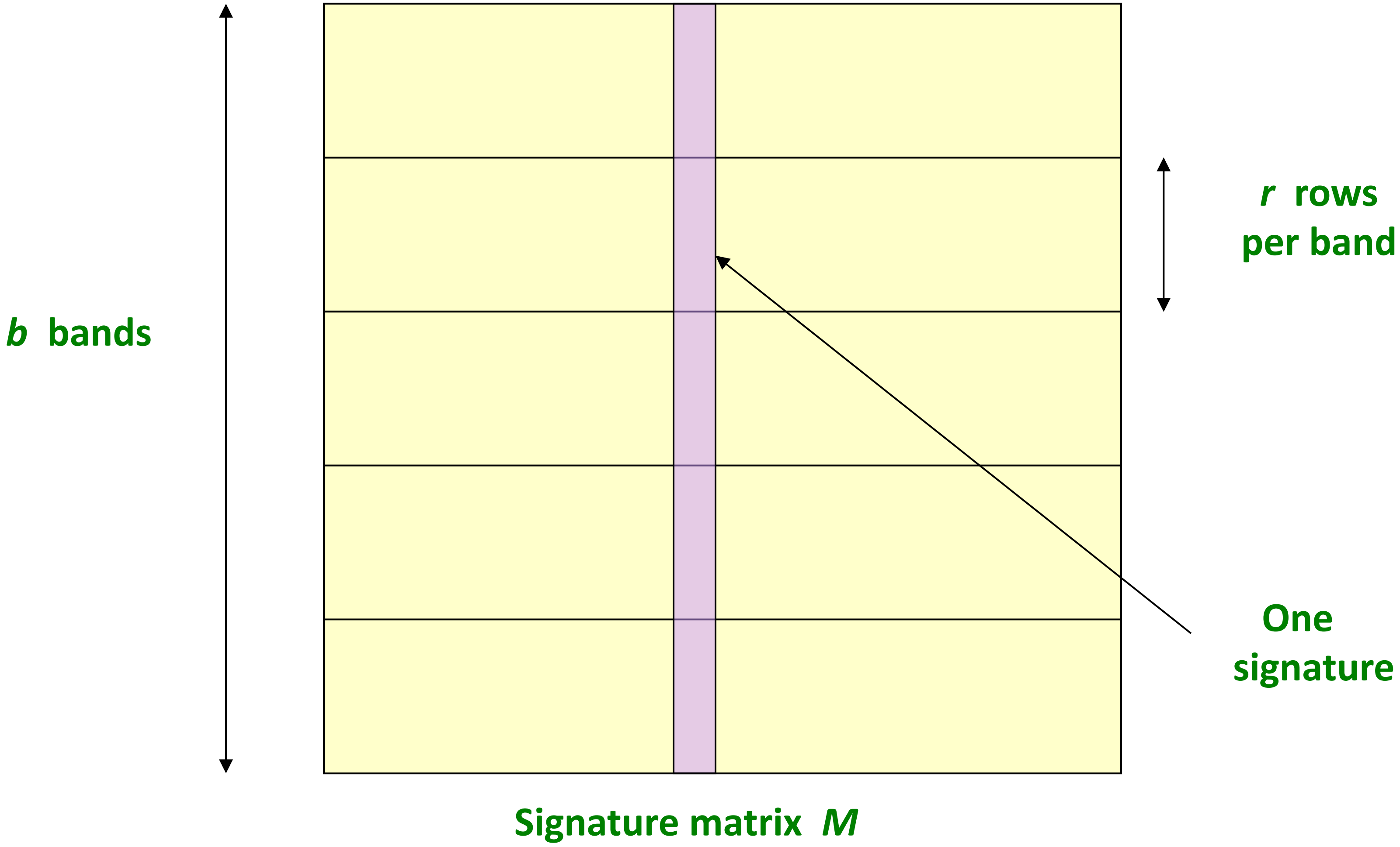
J(A,B) estimated by number of entries on which their signature corresponds

Signature matrix

- Each column represents a set
- Each rows represents a different hash



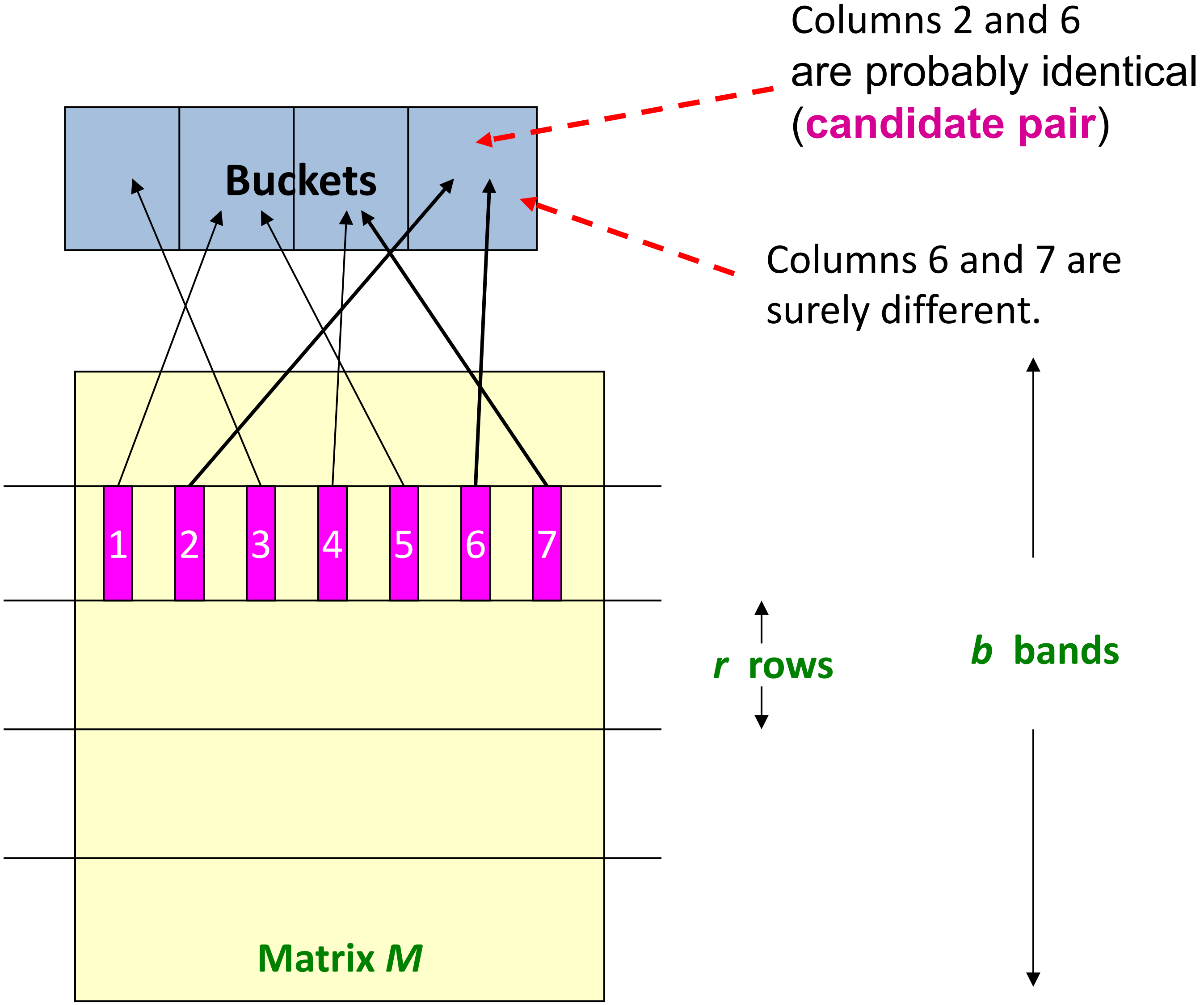
Partition *M* into *b* Bands



Partition *M* into *b* Bands

- Divide matrix ***M*** into ***b*** bands of ***r*** rows
- For each band, hash its portion of each column to a hash table with ***k*** buckets
  - Make ***k*** as large as possible
- ***Candidate*** column pairs are those that hash to the same bucket for ≥ **1** band
- Tune ***b*** and ***r*** to catch most similar pairs,
  but few non-similar pairs

Hashing Bands



Columns 2 and 6 are probably identical (**candidate pair**)

Columns 6 and 7 are surely different.

Simplifying Assumption

- There are **enough buckets** that columns are unlikely to hash to the same bucket unless they are **identical** in a particular band

- Hereafter, we assume that “**same bucket**” means “**identical in that band**”

- Assumption needed only to simplify analysis, not for correctness of algorithm

Example of Bands

Assume the following case:

- Suppose 100,000 columns of ***M*** (100k docs)
- Signatures of 100 integers (rows)
- Choose ***b*** = 20 bands of ***r*** = 5 integers/band

- **Goal:** Find pairs of documents that are at least ***s*** *= 0.8* similar

C_1, C_2 are 80% Similar

- **Find pairs of** ≥ ***s***=0.8 similarity, set **b**=20, **r**=5
- **Assume:** sim(C_1, C_2) = 0.8
  - Since sim(C_1, C_2) ≥ **s**, we want C_1, C_2 to be a **candidate pair**: We want them to hash to at **least 1 common bucket** (at least one band is identical)
- **Probability C_1, C_2 identical in one particular band:** $(0.8)^5 = 0.328$
- Probability C_1, C_2 are ***not*** similar in all 20 bands: $(1\text{-}0.328)^{20} = 0.00035$
  - i.e., about 1/3000th of the 80%-similar column pairs are **false negatives** (we miss them)
  - **We would find 99.965% pairs of truly similar documents**

C_1, C_2 are 30% Similar

- **Find pairs of ≥ *s*=0.8** similarity, set **b**=20, **r**=5
- **Assume:** sim(C_1, C_2) = 0.3
  - Since sim(C_1, C_2) < **s** we want C_1, C_2 to hash to **NO common buckets** (all bands should be different)
- **Probability C_1, C_2 identical in one particular band:** $(0.3)^5$ = 0.00243
- Probability C_1, C_2 identical in at least 1 of 20 bands: $1 - (1 - 0.00243)^{20} = 0.0474$
  - In other words, approximately 4.74% pairs of docs with similarity 0.3% end up becoming **candidate pairs**
    - They are **false positives** since we will have to examine them (they are candidate pairs) but it will turn out their similarity is below threshold **s**

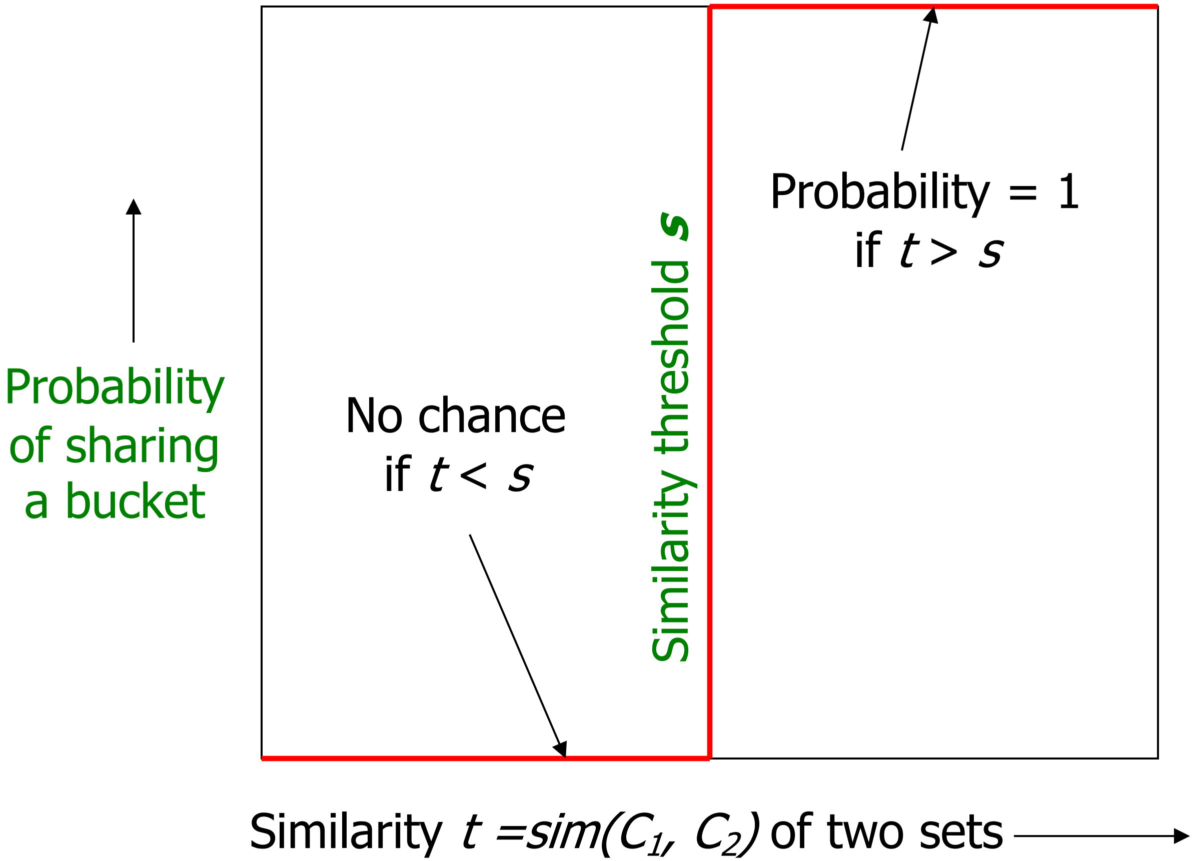
LSH Involves a Tradeoff

- **Pick:**
  - The number of Min-Hashes (rows of ***M***)
  - The number of bands ***b***, and
  - The number of rows ***r*** per band

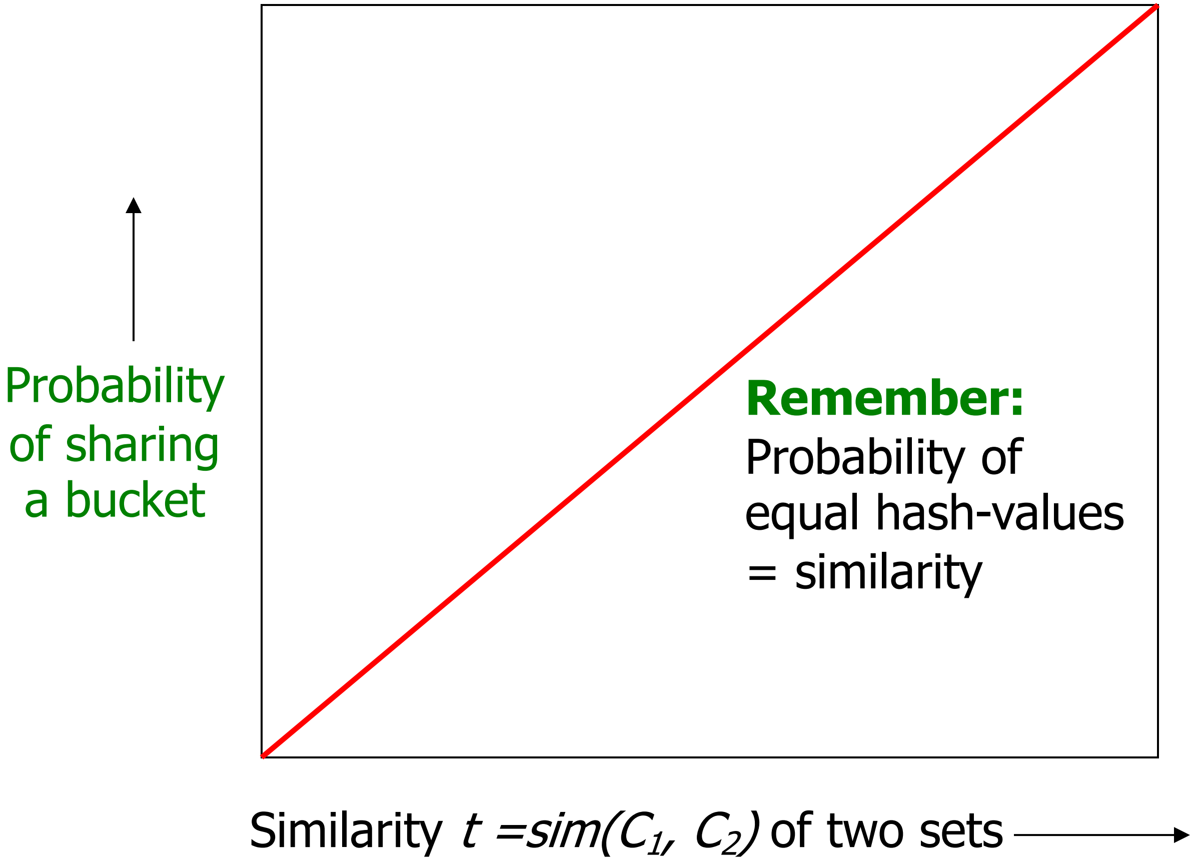
  to balance false positives/negatives

- **Example:** If we had only 15 bands of 5 rows, the number of false positives would go down, but the number of false negatives would go up

Analysis of LSH – What We Want



What 1 Band of 1 Row Gives You



Remember: Probability of equal hash-values = similarity

b bands, *r* rows/band

Columns C_1 and C_2 have similarity t

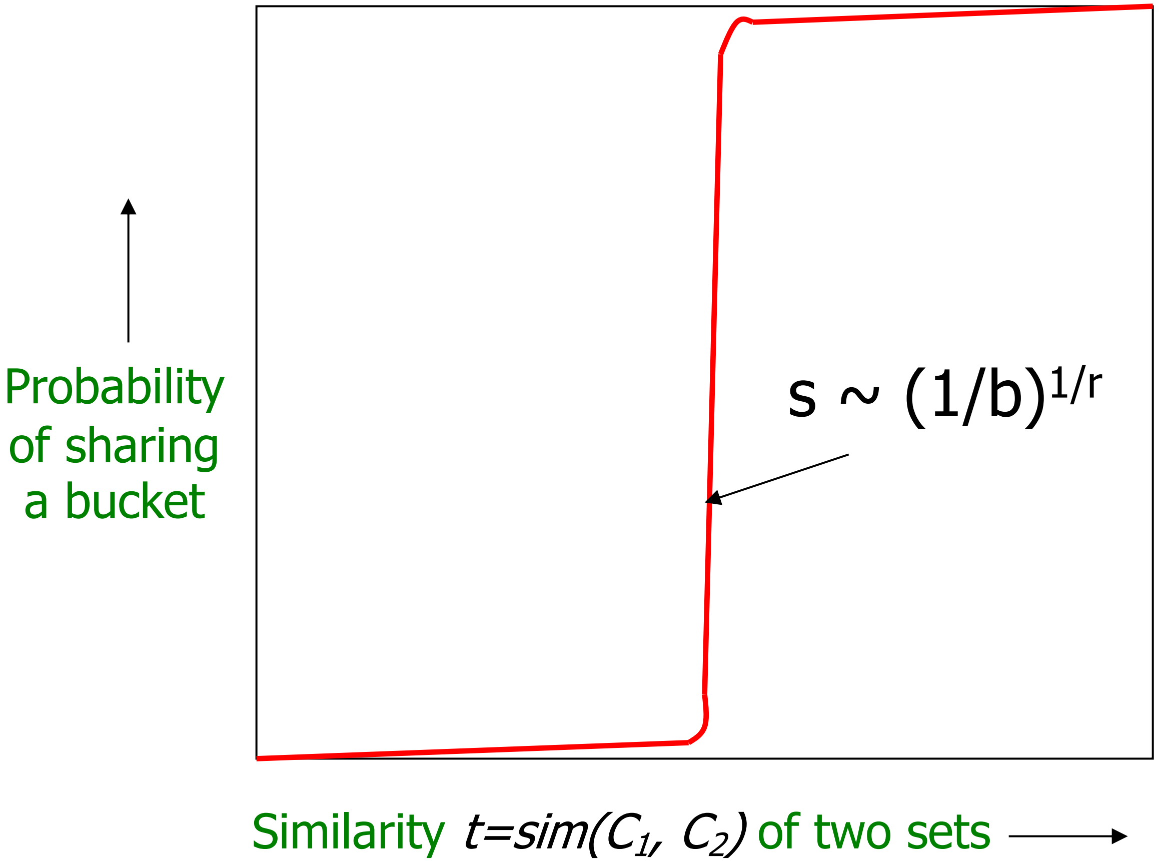
Pick any band (r rows)

- Prob. that all rows in band equal = t^r
- Prob. that some row in band unequal = $1 - t^r$

Prob. that no band identical = $(1 - t^r)^b$

Prob. that at least 1 band identical = $1 - (1 - t^r)^b$

What *b* Bands of *r* Rows Gives You



At least one band identical

No bands identical

$$\mathbf{1 - (1 - \boldsymbol{t}^{\boldsymbol{r}})^{\boldsymbol{b}}}$$

Some row of a band unequal

All rows of a band are equal

Example: $b = 20$; $r = 5$

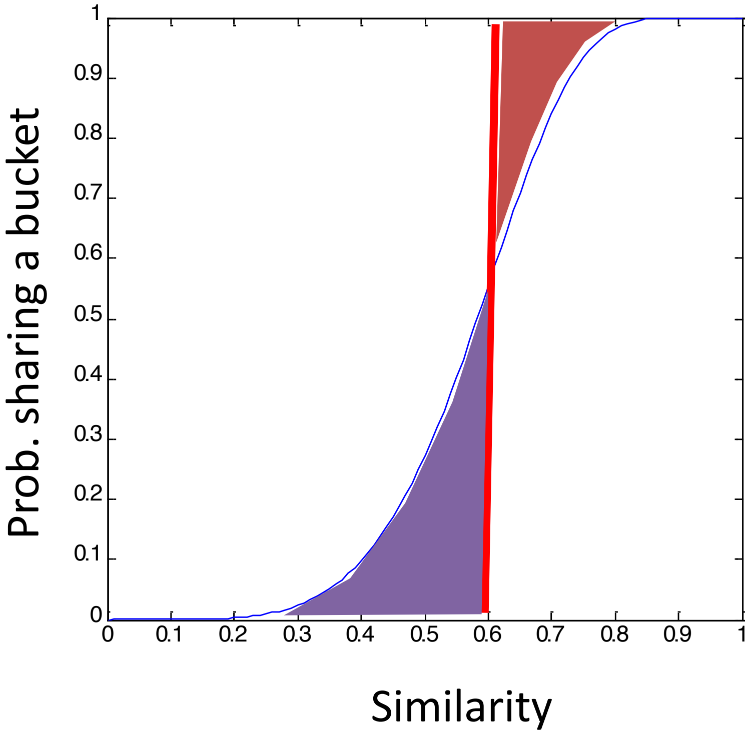
- **Similarity threshold s approx. 0,55**
- **Prob. that at least 1 band is identical:**

t	$1-(1-t^r)^b$
.2	.006
.3	.047
.4	.186
.5	.470
.6	.802
.7	.975
.8	.9996

Note the steep gap around the threshold

Picking *r* and *b*: The S-curve

- **Picking *r* and *b* to get the best S-curve**
  - 50 hash-functions (r=5, b=10)



Red area: False Negative rate

Blue area: False Positive rate

LSH Summary

- Tune ***M, b, r*** to get almost all pairs with similar signatures, but eliminate most pairs that do not have similar signatures

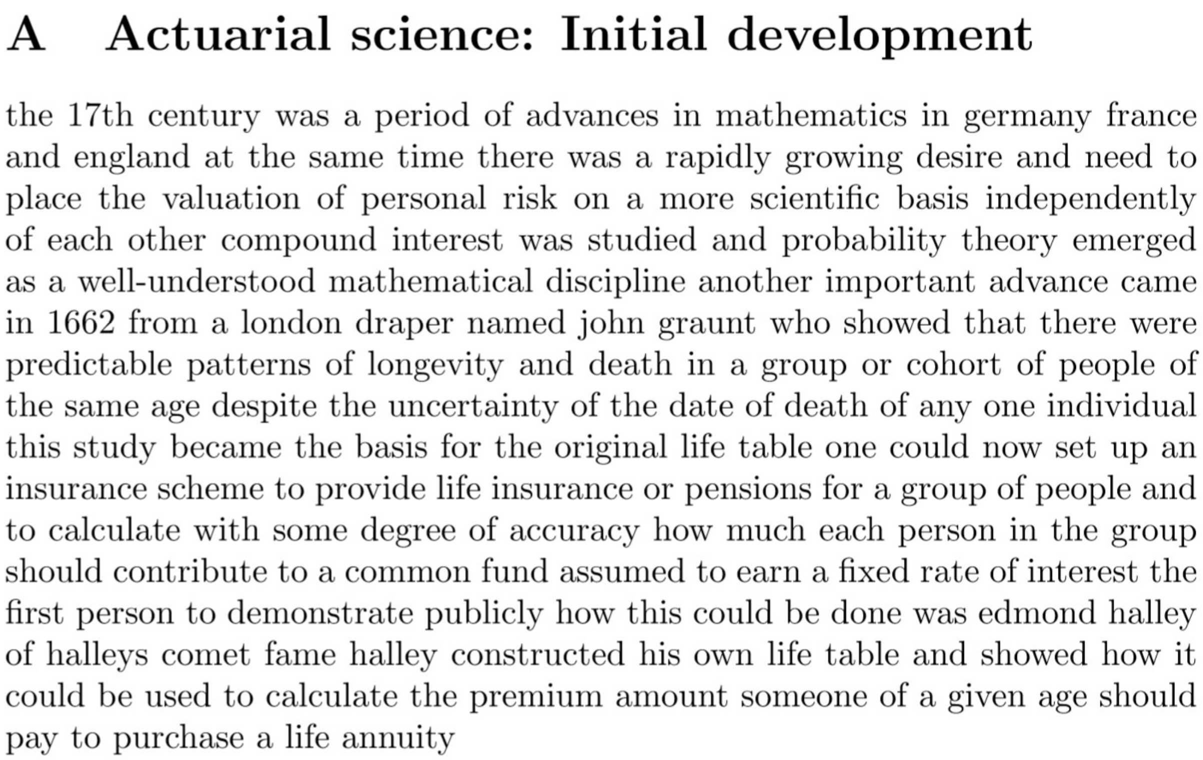
- Check in main memory that **candidate pairs** really do have **similar signatures**

- **Locality-sensitive** hashing reflects the fact that we want hash-functions that take locality into account
  - The more alike two points are, the more likely they hash into the same bucket

Case Study : Detecting Wiki-plagiarism

Make a near-duplicate detection system for Wikipedia articles

- Divide articles into chapters: 22,901,574 documents

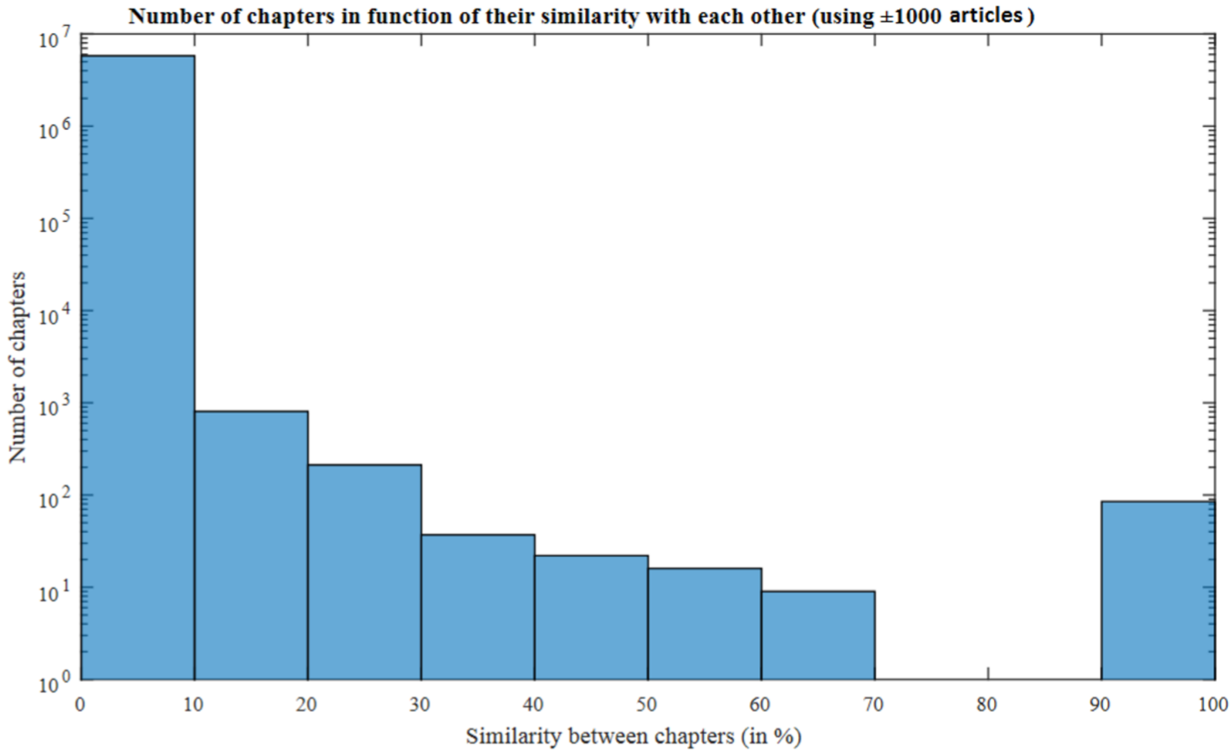


A Actuarial science: Initial development

the 17th century was a period of advances in mathematics in germany france and england at the same time there was a rapidly growing desire and need to place the valuation of personal risk on a more scientific basis independently of each other compound interest was studied and probability theory emerged as a well-understood mathematical discipline another important advance came in 1662 from a london draper named john graunt who showed that there were predictable patterns of longevity and death in a group or cohort of people of the same age despite the uncertainty of the date of death of any one individual this study became the basis for the original life table one could now set up an insurance scheme to provide life insurance or pensions for a group of people and to calculate with some degree of accuracy how much each person in the group should contribute to a common fund assumed to earn a fixed rate of interest the first person to demonstrate publicly how this could be done was edmond halley of halleys comet fame halley constructed his own life table and showed how it could be used to calculate the premium amount someone of a given age should pay to purchase a life annuity

Case Study : Detecting Wiki-plagiarism

- Create shingles and use MinHash to represent documents
  - Shingles of up to 4 consecutive words; hashed to 32 bits



Case Study : Detecting Wiki-plagiarism

- Create shingles and use MinHash to represent documents
  - Shingles of up to 4 consecutive words; hashed to 32 bits
  - Comparing two shingles takes about 0.91 miliseconds
  - Hence, finding a near duplicate by comparing all documents in the collection with the query document takes almost **6 hours**.

Case Study : Detecting Wiki-plagiarism

- Hence, LSH was used
  - Very few hashes were used: 20 to 50 hashes per sketch (different experiments)

Case Study : Detecting Wiki-plagiarism

- Hence, LSH was used
  - Very few hashes were used: 20 to 50 hashes per sketch (different experiments)

Some math:

5 bands, 4 rows per band:

$P[\ h(x)=h(y)\ |\ J(x,y) = 90\%\ ] \quad = \quad 1-(1-(90\%)^4)^5$

$= \quad 0{,}995\ldots$

$P[\ h(x)=h(y)\ |\ J(x,y) = 70\%\ ] \quad = \quad 1-(1-(70\%)^4)^5$

$= \quad 0{,}75\ \ldots$

Case Study : Detecting Wiki-plagiarism

- Hence, LSH was used
  - Very few hashes were used: 20 to 50 hashes per sketch (different experiments)

Some math:

7 bands, 7 rows per band:

$P[\ h(x)=h(y)\ |\ J(x,y) = 90\%\ ] \quad = \quad 1-(1-(90\%)^7)^7$

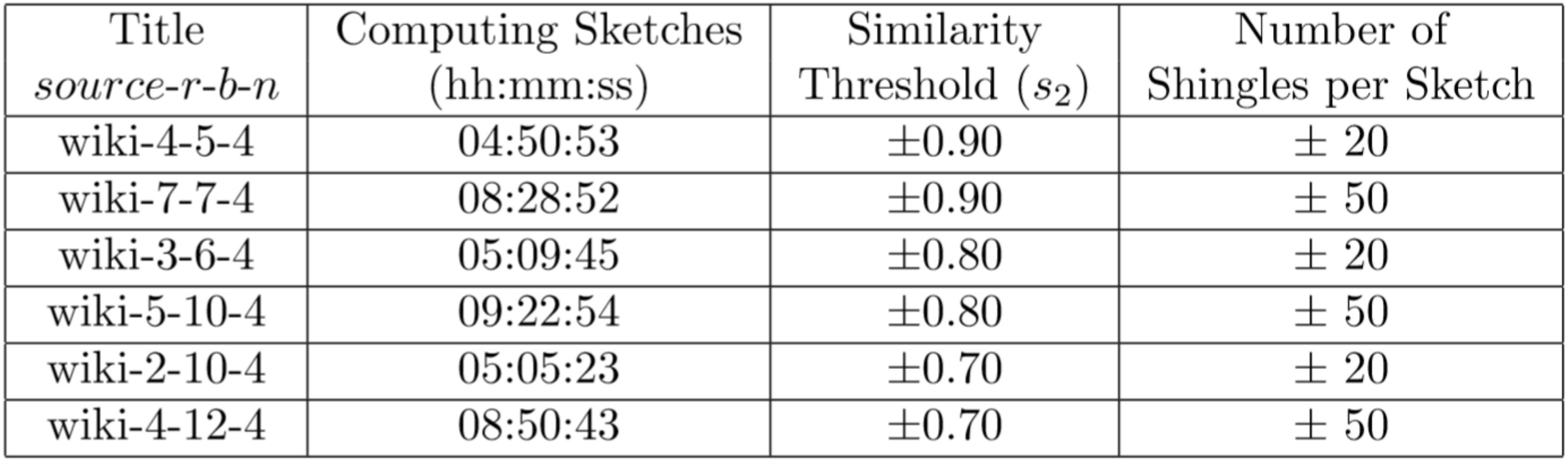
$= \quad 0{,}989\ldots$

$P[\ h(x)=h(y)\ |\ J(x,y) = 70\%\ ] \quad = \quad 1-(1-(70\%)^7)^7$

$= \quad 0{,}45\ \ldots$

Case Study : Detecting Wiki-plagiarism

- Index creation time:

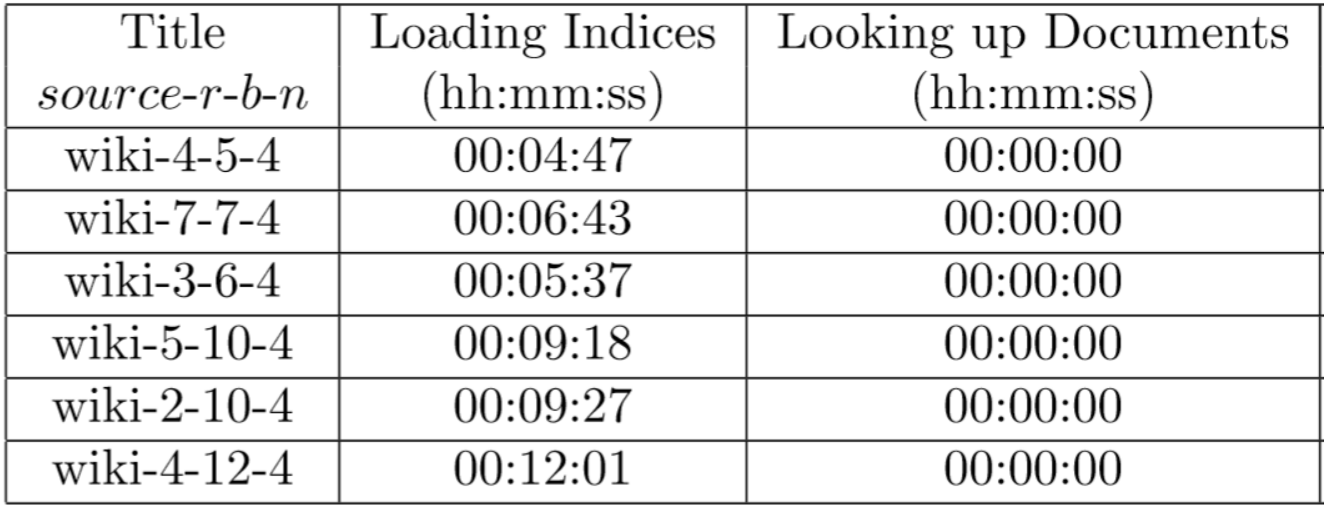


Title *source-r-b-n*	Computing Sketches (hh:mm:ss)	Similarity Threshold (s_2)	Number of Shingles per Sketch
wiki-4-5-4	04:50:53	±0.90	± 20
wiki-7-7-4	08:28:52	±0.90	± 50
wiki-3-6-4	05:09:45	±0.80	± 20
wiki-5-10-4	09:22:54	±0.80	± 50
wiki-2-10-4	05:05:23	±0.70	± 20
wiki-4-12-4	08:50:43	±0.70	± 50

Table 1: Execution times to compute the sketches

Case Study : Detecting Wiki-plagiarism

- Query time (recall: without index about **6 hours**)



Title *source-r-b-n*	Loading Indices (hh:mm:ss)	Looking up Documents (hh:mm:ss)
wiki-4-5-4	00:04:47	00:00:00
wiki-7-7-4	00:06:43	00:00:00
wiki-3-6-4	00:05:37	00:00:00
wiki-5-10-4	00:09:18	00:00:00
wiki-2-10-4	00:09:27	00:00:00
wiki-4-12-4	00:12:01	00:00:00

Summary: Similarity Search

LSH is an indexing technique for similarity search

Particularly useful for high-dimensional data

Can be extended to other similarity/distance measures

- Key ingredient: there must exist a hash-functions h such that P[h(x)=h(y)] increases with sim(x,y)
- These hash-functions can be combined to reach the needed sensitivity

Summary: Some new tools …

Frequent Items

Filtering

Distinct Counting

Similarity search

- If we are willing to rely on approximate results, many costly operations on big datasets can be executed very efficiently

Acknowledgement

Based on content created by Toon Calders.