

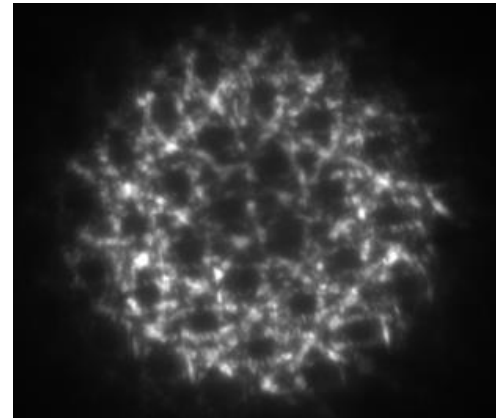
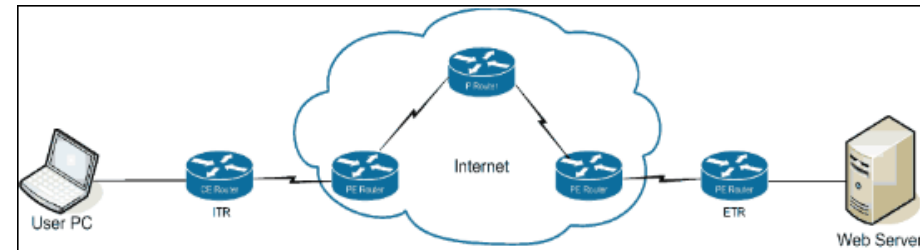
# Tools for a data scientist's toolbox Data Stream Processing and others

Toon Calders – Dept. of Mathematics and Computer Science

# Introduction

Many applications generate huge amounts of data

- Need to be processed on the fly
- Data too abundant to store all



# 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

# Tools for your data scientist's toolbox



What is hot?  
Tracking heavy hitters



Extreme Counting



Anyone like me?  
Similarity search



# Heavy Hitters

# Warming up problem: Heavy Hitters

“Given a stream  $S$ , identify all items (*heavy hitters*) that occur more than  $\theta N$  times”



Items arrive one by one; what we do not store will be inaccessible later on.

How much space needed for an *exact* solution?

# Heavy Hitters

Counting every item is impossible

- E.g., all pairs of people that phone to each other

We do not know on beforehand which combinations will be frequent

Example:



30 items; ●:8, ●:6, ●:5

All others are 3

If frequency is 20%: ● and ● need to be output

# Heavy Hitters – Worst Case

How much space needed for an *exact* solution?

Answer: worst case at least  $O(n \log(N))$

Proof:

Let  $\theta = 50\%$ , and we have already seen  $N/2-1$  symbols

Configuration: bag (no order) of symbols seen so far

For any two configurations the system needs to be in a different *state*

Hence, we need at least  $\log(\#\text{configurations})$  space





# Heavy Hitters – Alternative Solution

“Given a stream, identify all items (*heavy hitters*) that occur more than  $\theta N$  times”



Algorithm by *Karp et al*:

- $O(\log(N)/\theta)$  space
- Concession: may have **false positives**  
(*Can be made exact with 2 passes*)

Karp, Richard M., Scott Shenker, and Christos H. Papadimitriou. "A simple algorithm for finding frequent elements in streams and bags." *ACM Transactions on Database Systems (TODS)* 28.1 (2003): 51-55.



# Heavy Hitters – Set Solution

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



Remove 5 elements of a different color to get  $S'$ :

If ● was a heavy hitter, it still is!

Hence, removing 5 elements of different color gives us a smaller set, but we keep all heavy hitters.

# Heavy Hitters – Set Solution

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



Remove 5 elements of a different color to get  $S'$ :

If ● was a heavy hitter, it still is!

Hence, removing 5 elements of different color gives us a smaller set, but we keep all heavy hitters.

- Can be done repeatedly

# Heavy Hitters – Set Solution

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



Remove 5 elements of a different color to get  $S'$ :

If ● was a heavy hitter, it still is!

Hence, removing 5 elements of different color gives us a smaller set, but we keep all heavy hitters.

- Can be done repeatedly

# Heavy Hitters – Set Solution

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



Remove 5 elements of a different color to get  $S'$ :

If ● was a heavy hitter, it still is!

Hence, removing 5 elements of different color gives us a smaller set, but we keep all heavy hitters.

- Can be done repeatedly

# Heavy Hitters – Set Solution

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



Remove 5 elements of a different color to get  $S'$ :

If ● was a heavy hitter, it still is!

Hence, removing 5 elements of different color gives us a smaller set, but we keep all heavy hitters.

- Can be done repeatedly
- Until no longer possible to remove 5

Answer is a subset of the remaining (at most 4) colors:



# Heavy Hitters – Stream Implementation

“Given a stream, identify all items that occur more than  $\theta N$  times”

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



# Heavy Hitters - Summary

Algorithm by *Karp et al.*

Problem:

- Find all items exceeding frequency  $\theta N$

Space:  $O(1 / \theta)$  counters

Time per update:  $O(1)$  \*

**Concession:**

- **False positives or 2-pass**

\* Karp et al. propose a datastructure 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 al.

*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  $\theta$ -frequent items:

- Initialization: none of the items has a counter
- Item  $\bullet$  enters at time  $t$ :
  - If  $\bullet$  has a counter:  $\text{count}(\bullet) ++$
  - Else:
    - $\text{count}(\bullet) = 1$
    - $\text{start}(\bullet) = t$
  - For all other items  $\blacktriangle$  with a counter do:
    - If  $\text{count}(\blacktriangle) / (t - \text{start}(\blacktriangle) + 1) < \theta$  :
      - Delete Counter for  $\blacktriangle$

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%)
●	5	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)
●	<u>2</u>	1 (25%)
●	<u>17</u>	4 (29%)
●	<u>27</u>	1 (25%)
●	<u>8</u>	6 (26%)
●	<u>19</u>	3 (25%)



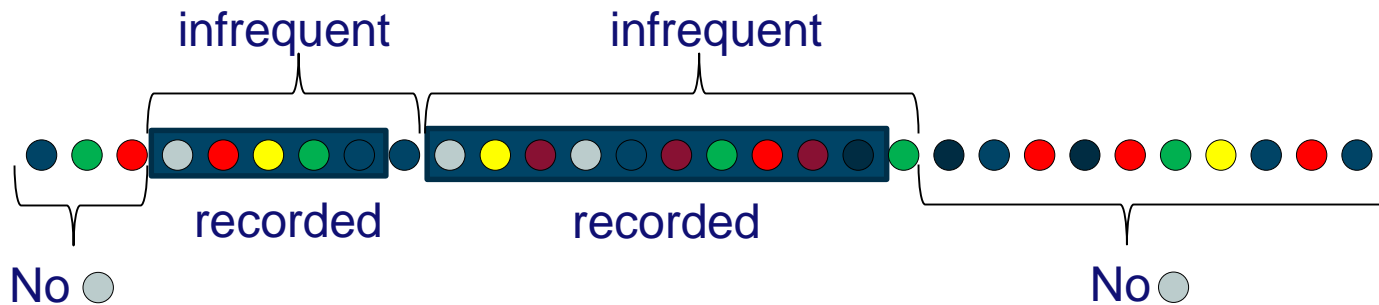
# Lossy Counting - Correctness

Why does it work?

- If  $\circ$  is not recorded,  $\circ$  is not frequent in the stream

Imagine marking when  $\circ$  was recorded:

- If  $\circ$  occurs, recording starts
- Only stopped if  $\circ$  becomes infrequent since start recording

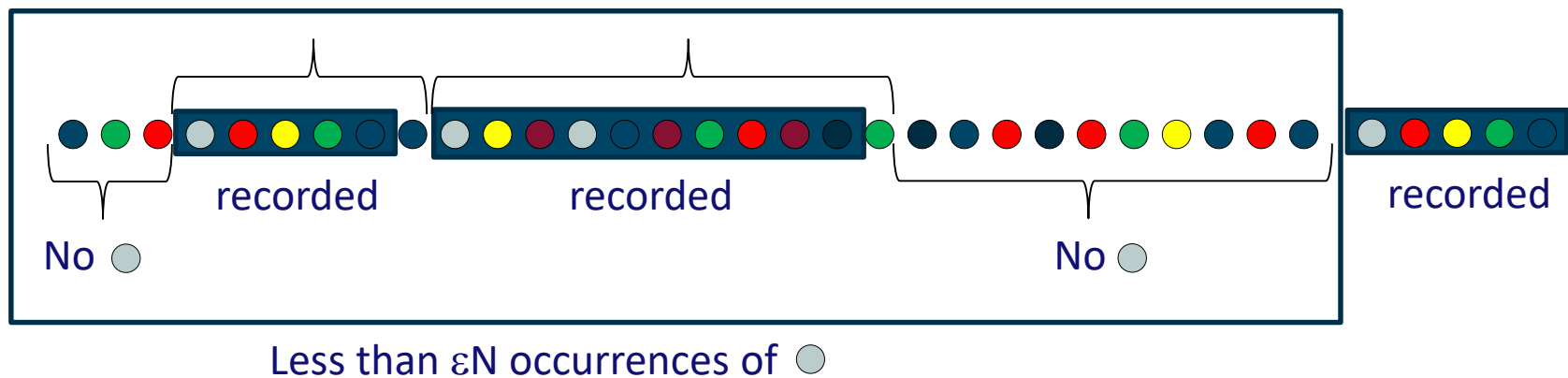


Whole stream can be partitioned into parts in which  $\circ$  is not frequent  $\rightarrow$   $\circ$  is not frequent in the whole stream

# Lossy Counting – Guarantee

Run the algorithm with  $\epsilon$  as threshold

Guaranteed: at any point in time, the true frequency of  $\circ$  is in the interval  $[ \text{count}(\circ)/N , \text{count}(\circ)/N + \epsilon ]$



Report all items  $\circ$  with  $\text{count}(\circ) \geq (\theta - \epsilon) N$

- All items in result have frequency at least  $\theta - \epsilon$
- All items with frequency  $\theta$  are in the result

# Heavy hitters - Summary



Karp's algorithm:

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

Lossy Counting:

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

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

# Heavy Hitters - Applications

- Automatically block IP-traffic between pairs of addresses taking up more than 1% of the bandwidth
  - using lossy counting:
    - $1000 \log(N/1000)$  counters worst case
      - 30Kb for 1,000,000,000 items
    - Set threshold to 1.1%
    - 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



# Extreme Counting

# Counting items

Problem; give the number of unique items in a stream.

Highly useful property:

- Number of unique visitors to a website
- Estimate cardinality of projecting a relation onto a subset of its attributes

Heavy hitters are not suited for counting the number of unique items in a stream



# Extreme Counting

How many people attend my presentation?



# Extreme Counting: Attempt 1a

$S = \{\}$

$N = 0$

**Whenever** a person  $P$  enters the room:

**if**  $P$  not in  $S$ :

$$S = S \cup \{ P \}$$

$$N += 1$$

How many people  
attend my presentation?



**Exact** algorithm

**Complexity:**

**Space**  $O( N \text{ len}(\text{identifier of } P) )$

**Time**  $\log(N)$



# Extreme Counting: Attempt 1b

$S = \{\}$

$N = 0$

**Whenever** a person  $P$  enters the room:

**if**  $h(P)$  **not in**  $S$ :

$S = S \cup \{ h(P) \}$

$N += 1$

$h(P)$  denotes hash of identifier of  $P$

**Near exact** algorithm if  $\text{range}(h)$  large enough

**Complexity:**

**Space**  $O( N \log(N) )$

**Time**  $O( \log(N) )$

How many people  
attend my presentation?



# Extreme Counting

This solution is not satisfactory at all

- Space  $N \log(N)$  is completely unacceptable
- Time  $\log(N)$  is barely acceptable

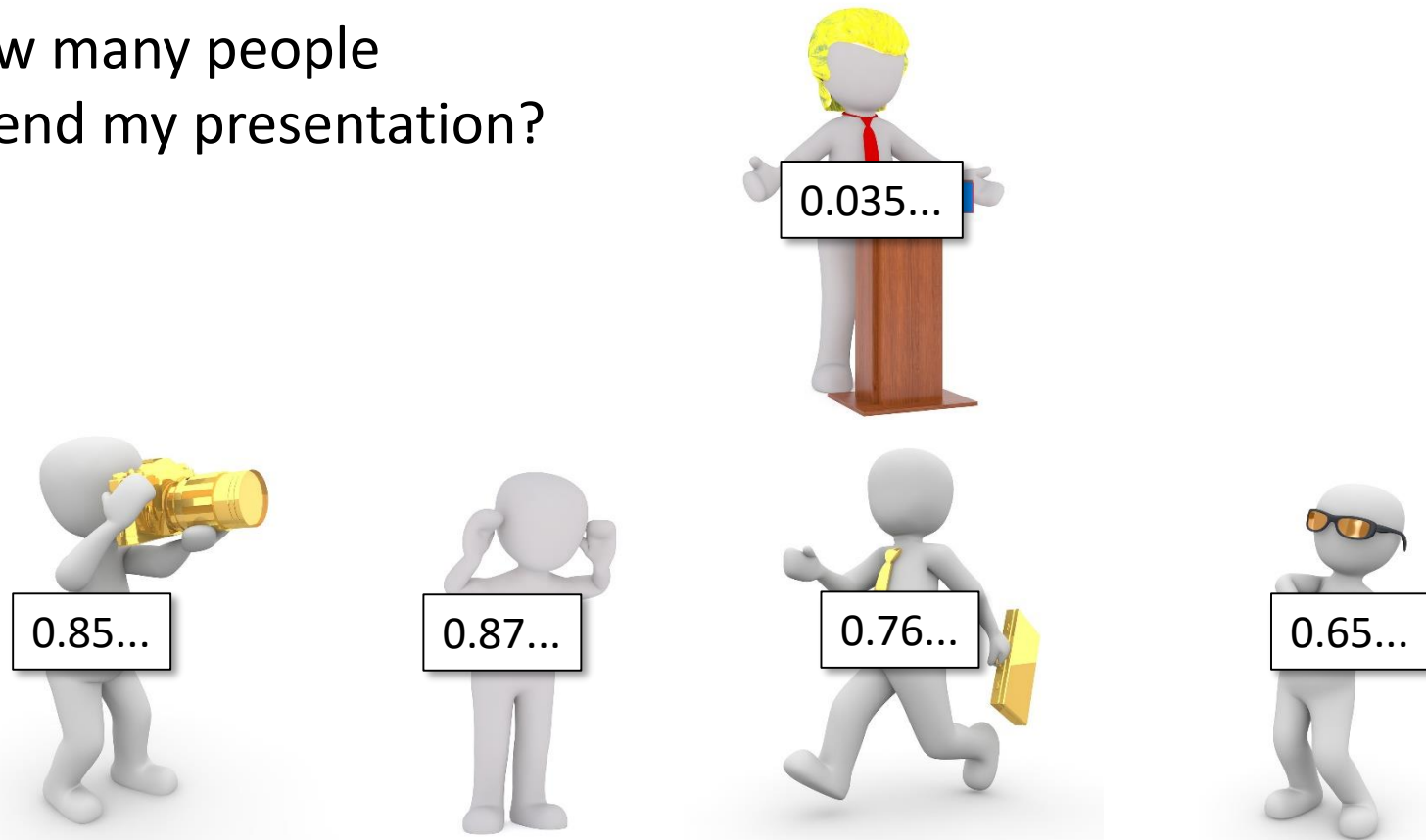
We will introduce an alternative: Hyperloglog sketch

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

HLL is based on the idea of Flajolet-Martin sketches

# Main idea behind Flajolet-Martin sketches

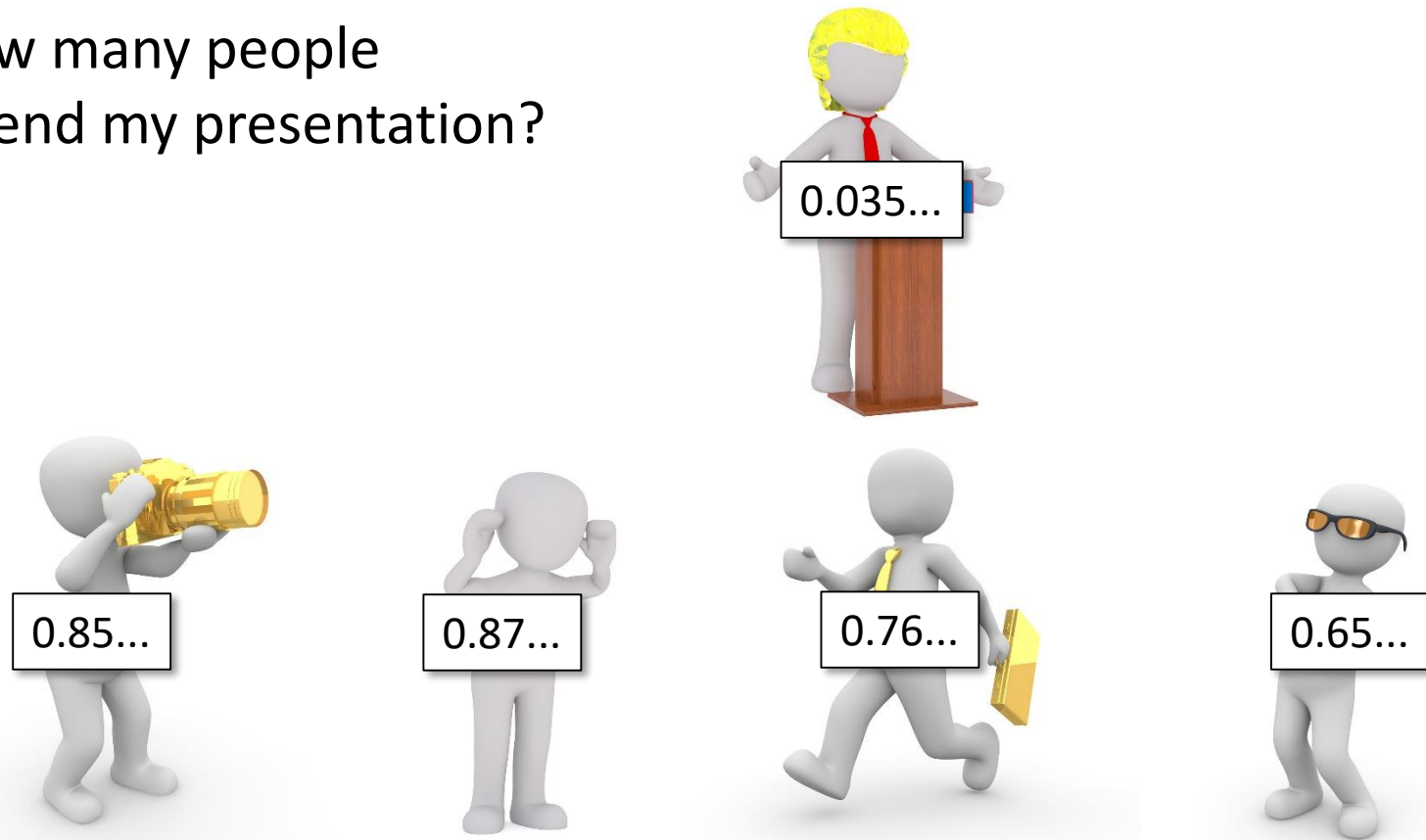
How many people attend my presentation?



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

# Main idea behind Flajolet-Martin sketches

How many people attend my presentation?



Compute the number for everyone entering the room  
Maintain the maximum over all numbers seen

# Main idea behind Flajolet-Martin sketches

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:

$$N = \frac{E[S]}{1-E[S]}$$

# Main idea behind Flajolet-Martin sketches

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  
 $h_1, \dots, h_k$  give  $L_1, \dots, L_k$

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



# Main idea behind Flajolet-Martin sketches

**Second issue: for large  $N$ , the quantity  $S$  will quickly become indistinguishable from 1**

- Flajolet-Martin use 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* of the number of 0s in the tail

**Probability of finding a tail of  $r$  zeros:**

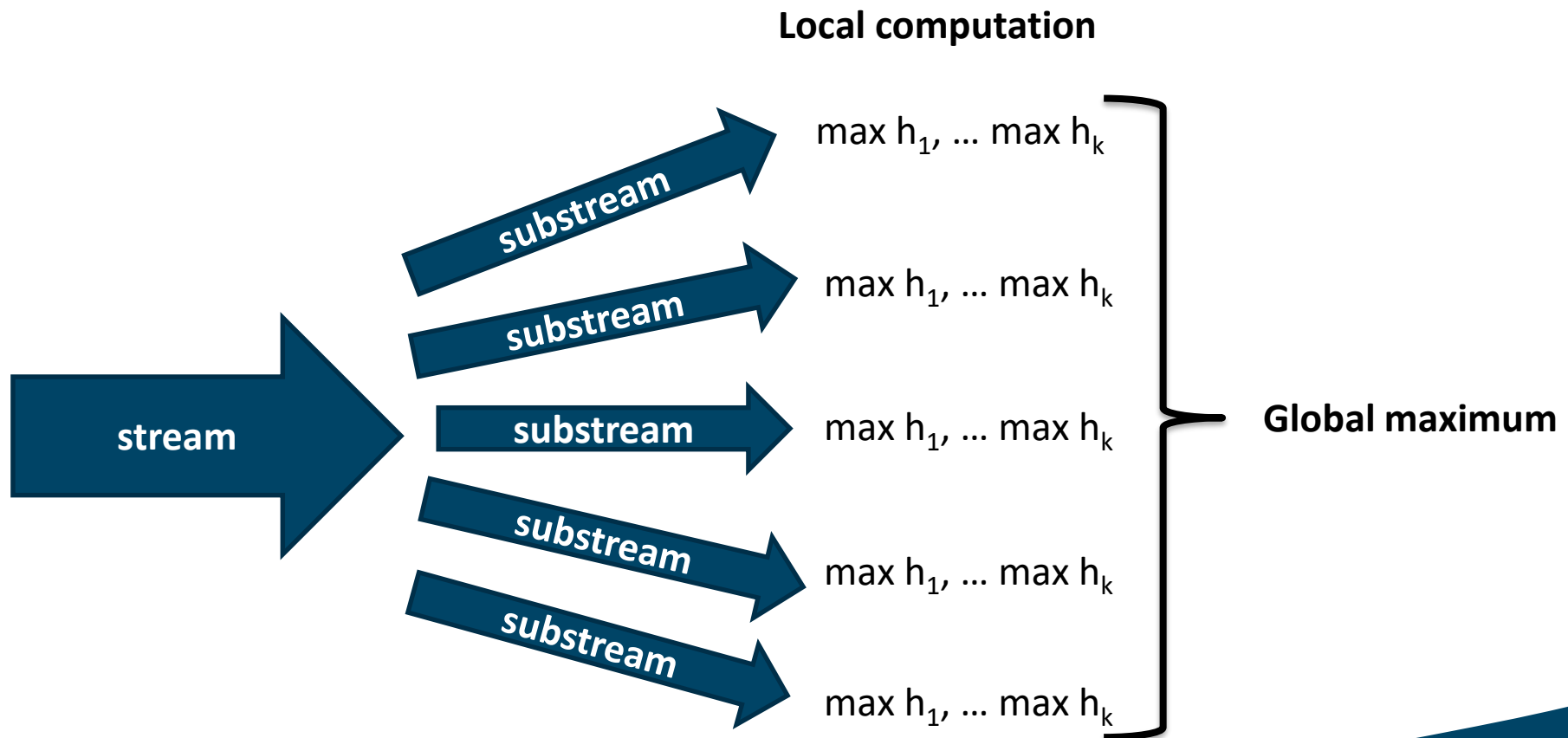
- Goes to **1** if  $N \gg 2^r$
- Goes to **0** if  $N \ll 2^r$

**Thus,  $2^R$  will almost always be around  $m$ !**

# FM-Sketches are easy to parallelise

No problem; easily parallelizable

- $\max(\max(A), \max(B)) = \max(A \cup B)$





# 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  $\frac{1.04}{\sqrt{m}}$  (m is size of summary)

- Independent of stream size
- Hence,  $\log(\log(N))$  dependence on stream length

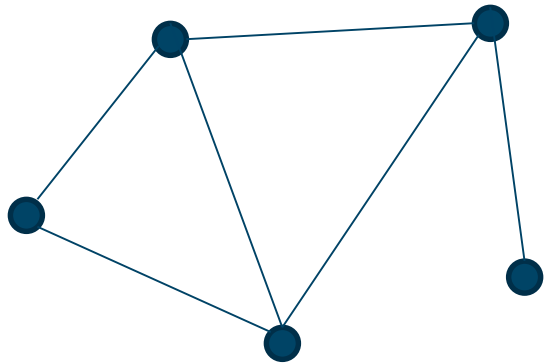
Great demo at:

<http://content.research.neustar.biz/blog/hll.html>



# Application: Neighborhood Function

Count the number of pairs of nodes at distance 1, 2, 3, ...



1: 6  
2: 3  
3: 1

Important statistics; allows to compute average degree, diameter, effective diameter.

Boldi, Paolo, Marco Rosa, and Sebastiano Vigna. "HyperANF: Approximating the neighbourhood function of very large graphs on a budget." *Proceedings of the 20th international conference on World wide web*. ACM, 2011.

# Application: Neighborhood Function

## Straightforward algorithm

Set  $N_0(v) = \{v\}$

For  $i = 1$  to  $r$ :

For all  $v$  in  $V$ :

$$N_i(v) = N_{i-1}(v)$$

For  $\{v, w\}$  in  $E$ :

$$N_i(v) \leftarrow N_i(v) \cup N_{i-1}(w)$$

$$N_i(w) \leftarrow N_i(w) \cup N_{i-1}(v)$$

Return  $\text{avg}(|N_1(v)|), \text{avg}(|N_2(v)| - |N_1(v)|), \dots$

Time:  $O(r |V| |E|)$

Space:  $O(|V|^2)$



# Application: Neighborhood Function

Observation: we can replace every set by a *summary*

- Take union, cardinality, add an element

Size of set:  $V$  versus size of summary:  $k \ll \ll |V|$

- $|V|$  versus  $\log(\log(|V|))$

With the summary we achieve:

- Time  $O(r k |E|)$
- Space  $O(k |V|)$

Speedup is enormous





# Anyone like me? 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-Index 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 (Altavista)

A set of n-grams in document 1

B set of n-grams in document 2

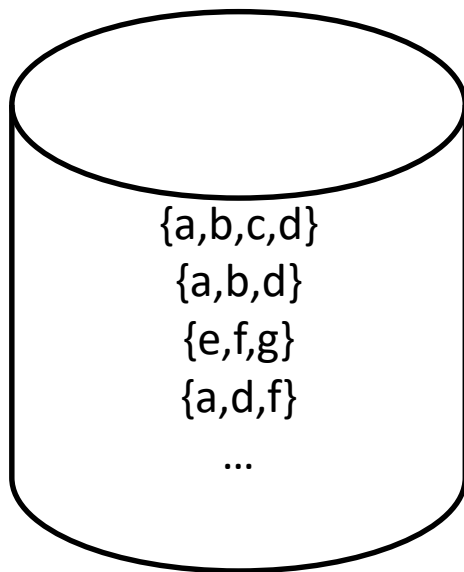


# Similarity Search – Jaccard Index

## Example:

Recommendation needs to be made for a user U

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



U bought {a,b,c}

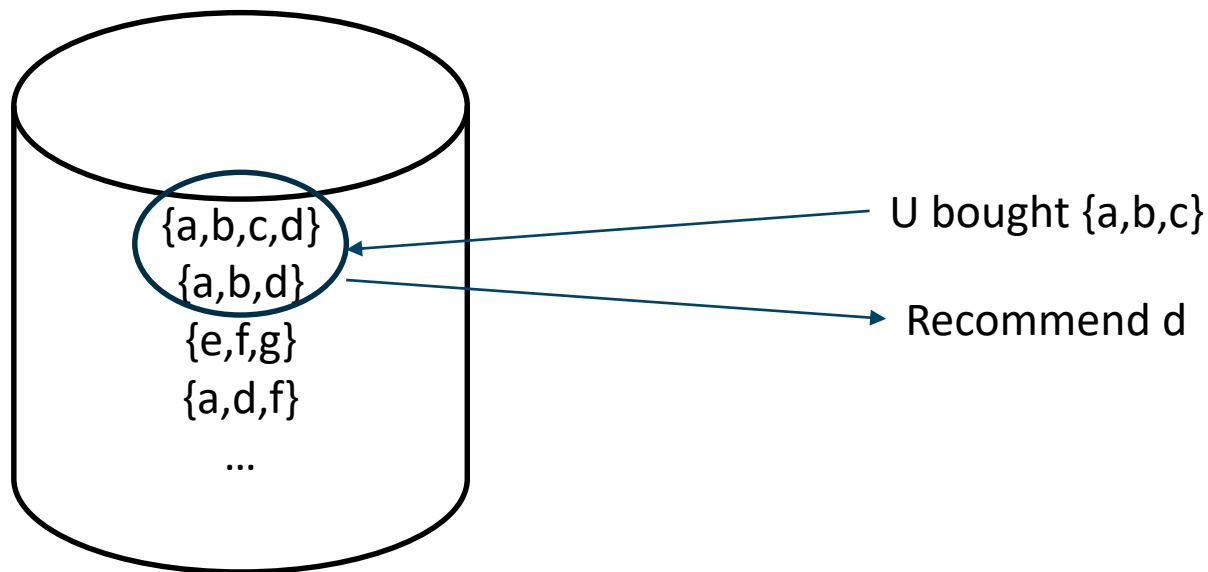


# Similarity Search – Jaccard Index

## Example:

Recommendation needs to be made for a user U

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



# Similarity Search: Naïve Algorithm

For each user  $U'$  in DB:

    compute  $Idx := J(\text{items}(U), \text{items}(U'))$

    if  $Idx \geq \text{threshold}$ :

        return  $\text{items}(U')$

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  $U$

$h$  is a function mapping elements of  $U$  to  $\{1, 2, \dots, |U|\}$

Example:  $d \rightarrow 1, c \rightarrow 2, a \rightarrow 3, b \rightarrow 4$

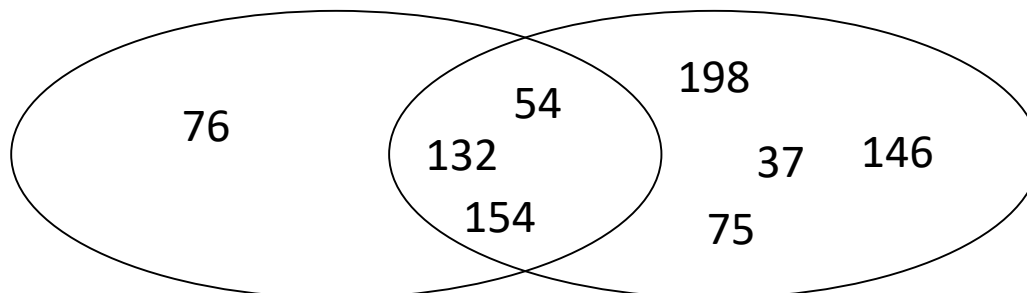
Let  $\min_h(A) := \min_{a \in A} h(a)$

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

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

$$= J(A, B)$$

\* We implicitly assume  $\text{range}(h) \gg |A|, |B|$



# Locality-Sensitive

We call such a function  $\min_h$  *locality-sensitive* for Jaccard  
Independent locality-sensitive functions can be combined

Independent functions  $h_1, \dots, h_m$

“signature” of set A:

$|A|$  and vector  $(\min_{h_1}(A), \min_{h_2}(A), \dots, \min_{h_m}(A))$

Estimating  $J(A,B)$

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

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

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

# Jaccard Coefficient

Example:  $U = \{ a, b, c, d, e \}$

$A = \{ a, b \}$

$B = \{ b, c, d \}$

$C = \{ a, b, c, e \}$

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:  $1/4$

	$h_1$	$h_2$	$h_3$	$h_4$
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

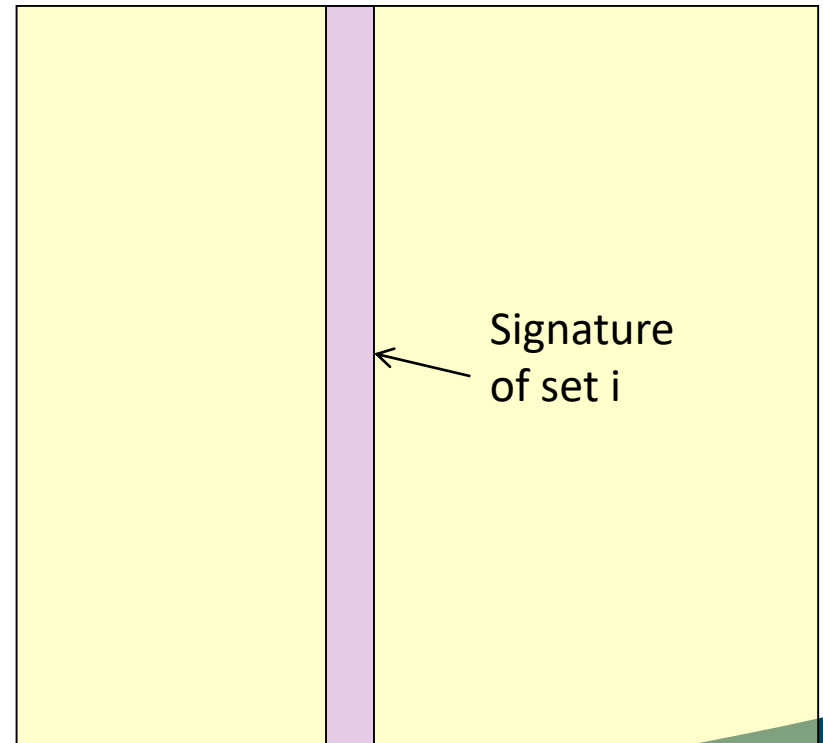
# Locality-Sensitive Hashing

We will first illustrate the principle for Jaccard Index  
MinHashing to create signatures of sets

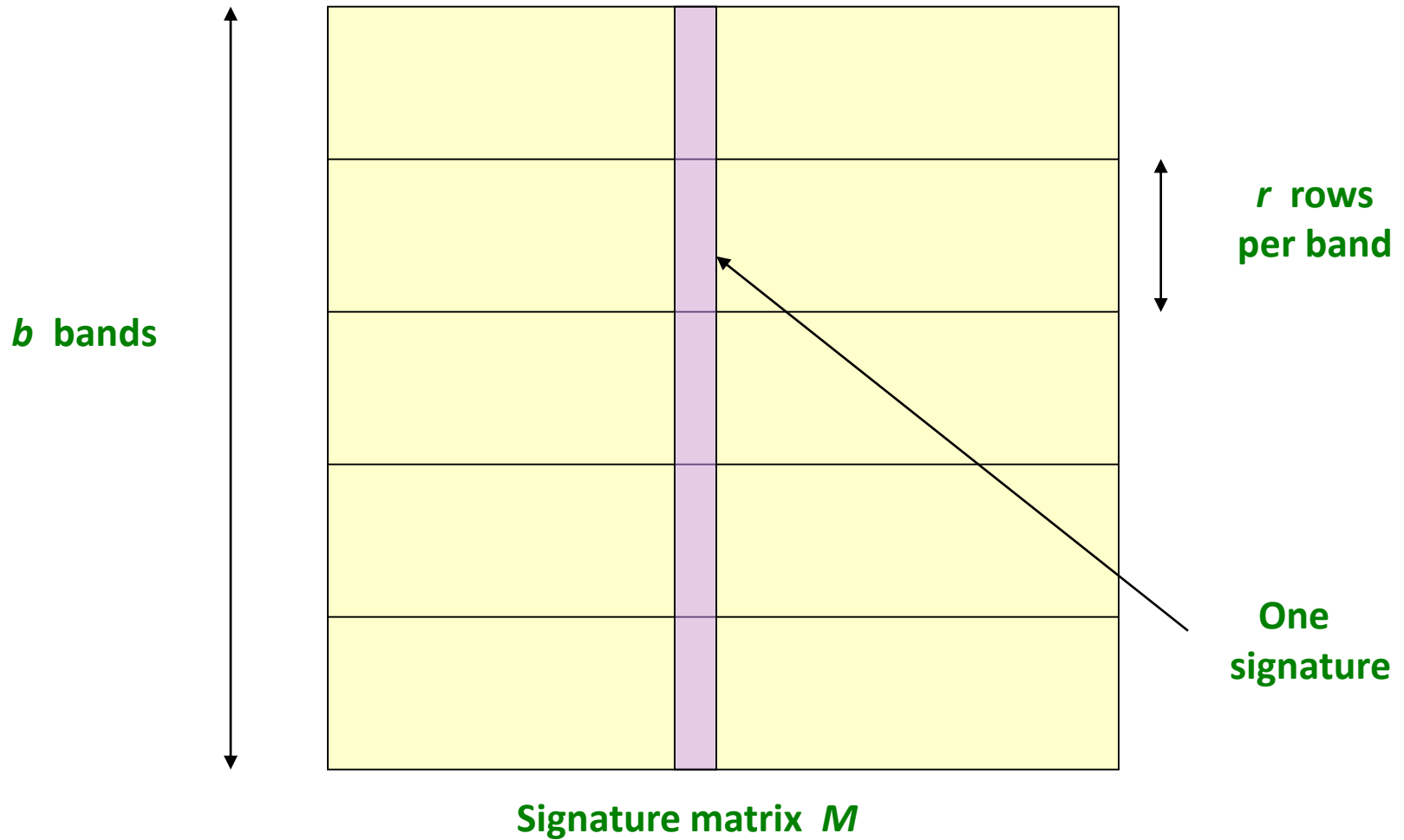
- $A \rightarrow [123, 235, 576, 67, 56]$
- $B \rightarrow [123, 3456, 56, 67, 867]$

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

Signature matrix



# Partition $M$ into $b$ Bands



# Partition $M$ into 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

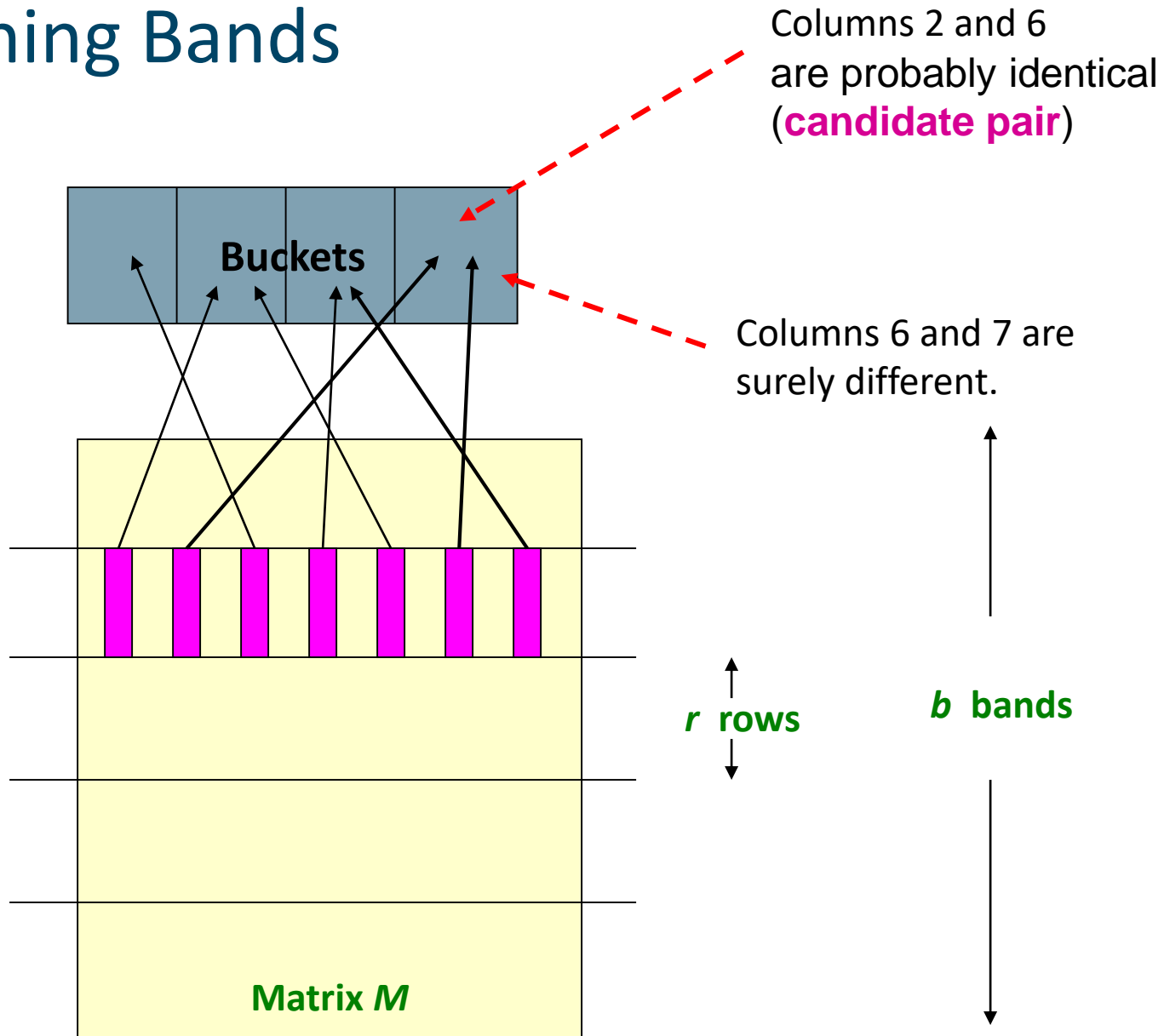
- Make  $k$  as large as possible

**Candidate** column pairs are those that hash to the same bucket for  $\geq 1$  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.

**$r$  rows**

**$b$  bands**

**Matrix M**

# Example: Bands

## Sets

$A = \{ a, b \}$

$B = \{ b, c, d \}$

$C = \{ a, b, c, e \}$

## Signatures

$(1, 2, 2, 2, 3, 1)$

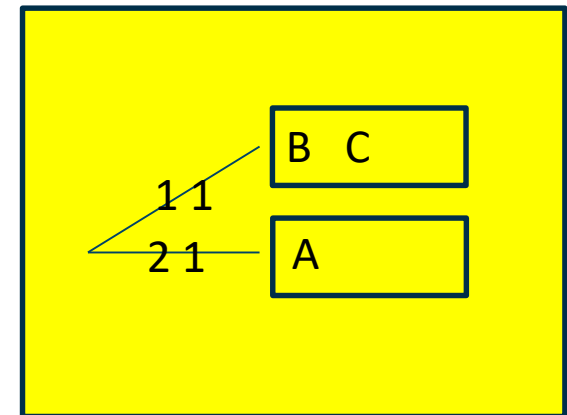
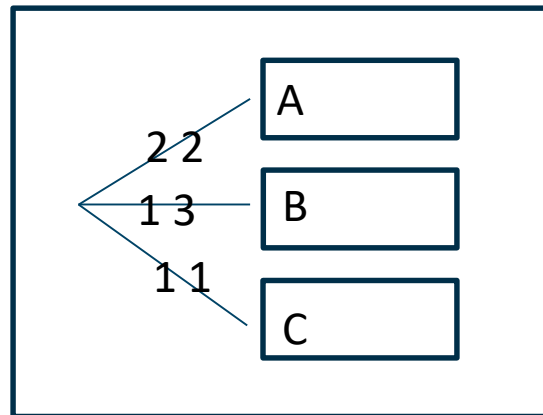
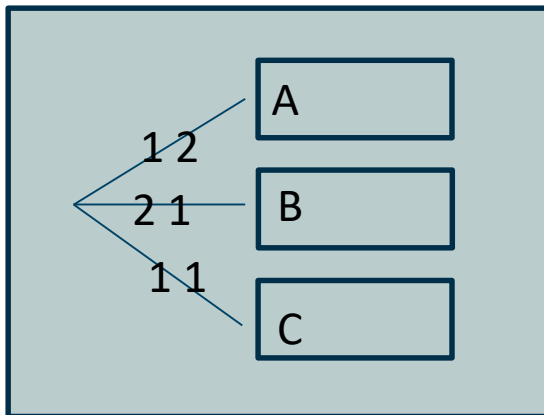
$(2, 1, 1, 3, 1, 1)$

$(1, 1, 2, 1, 1, 1)$

## Matrix M

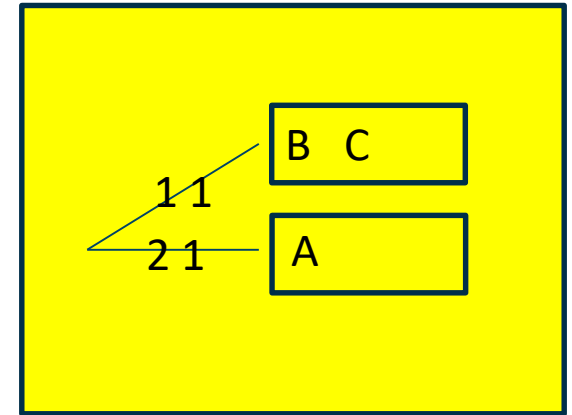
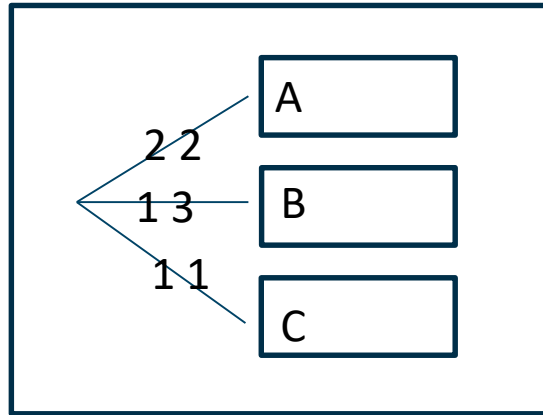
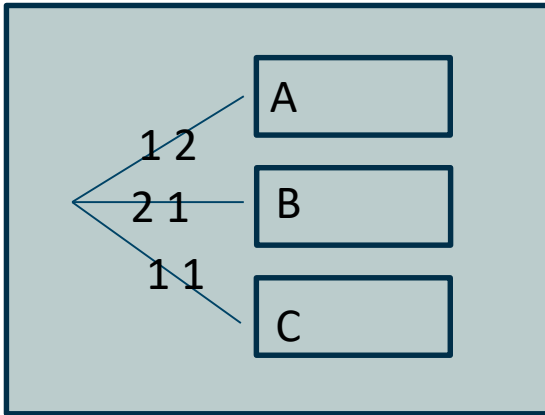
1	2	1
2	1	1
2	1	2
2	3	1
3	1	1
1	1	1

## Partitions



# Example: Bands

## Partitions

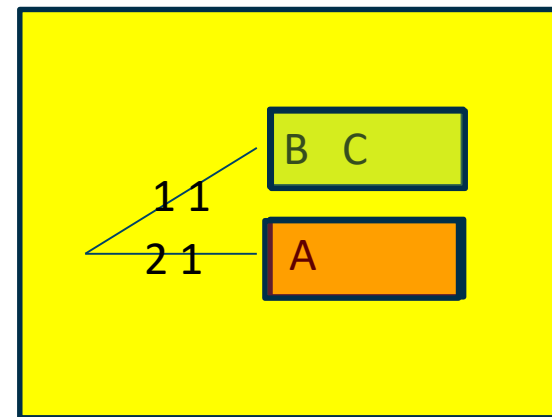
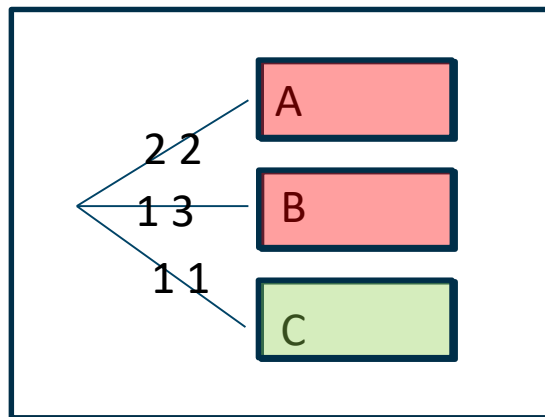
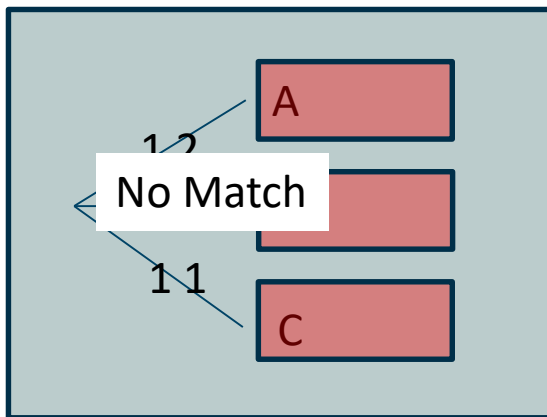


New item  $D = \{ b, c, e \}$  arrives; to which element do I need to compare?

1. Compute signature of  $D$ :  $(2,2,1,1,1,1)$
2. Split signature into bands:  $(2,2,1,1,1,1)$

# Example: Bands

## Partitions



New item  $D = \{ b, c, e \}$  arrives; to which element do I need to compare?

1. Compute signature of D: (2,2,1,1,1,1)
2. Split signature into bands: (2,2,1,1,1,1)
3. For each band, lookup candidates with that key
4. B and C match; do an in-depth comparison with these elements only

# 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 items)

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 to a query  $Q$

# C, Q are 80% Similar

Find pairs of  $\geq s=0.8$  similarity, set  $b=20$ ,  $r=5$

**Assume:**  $\text{sim}(C, Q) = 0.8$

- Since  $\text{sim}(C, Q) \geq s$ , we want C to be a **candidate**

**Probability  $\text{sig}(C)$  and  $\text{sig}(Q)$  identical in one particular band:**

$$(0.8)^5 = 0.328$$

**Probability  $\text{sig}(C)$  and  $\text{sig}(Q)$  not identical in any band:**

$$(1-0.328)^{20} = 0.00035$$

- i.e., about 1/3000th of the 80%-similar items in DB are **false negatives** (we miss them)
- **We would find 99.965% pairs of truly similar items**

# C, Q are 30% Similar

Find pairs of  $\geq s=0.8$  similarity, set  $b=20$ ,  $r=5$

**Assume:**  $\text{sim}(C, Q) = 0.3$

- Since  $\text{sim}(C, Q) < s$ , we do NOT want C to be a **candidate**

**Probability  $\text{sig}(C)$  and  $\text{sig}(Q)$  identical in one particular band:**

$$(0.3)^5 = 0.00243$$

**Probability  $\text{sig}(C)$  and  $\text{sig}(Q)$  identical in at least one band:**

$$1 - (1 - 0.00243)^{20} = 0.0474$$

- In other words, approximately 4.74% items in the DB with similarity 30% end up becoming **candidate**
  - They are **false positives** since we will have to examine them (they are candidates) but then it will turn out their similarity is below threshold  $s$



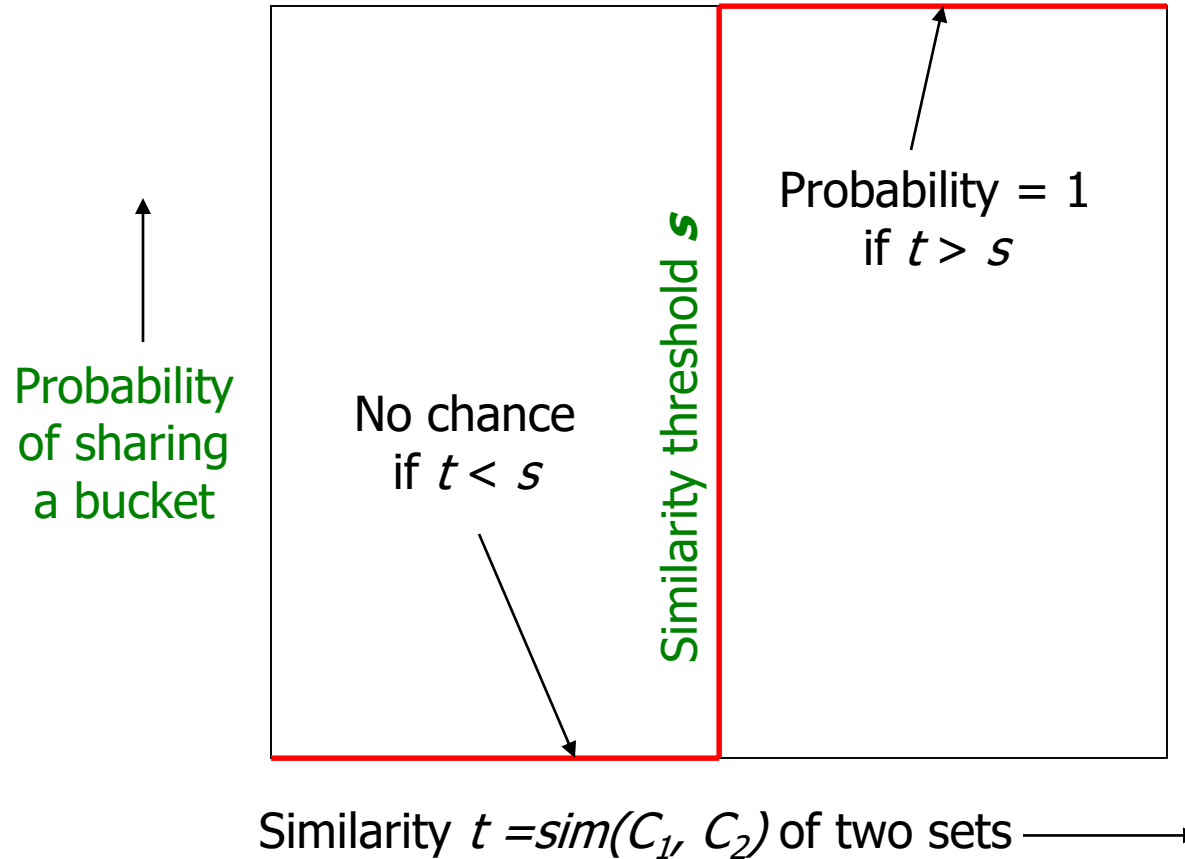
# LSH Involves a Tradeoff

## Pick:

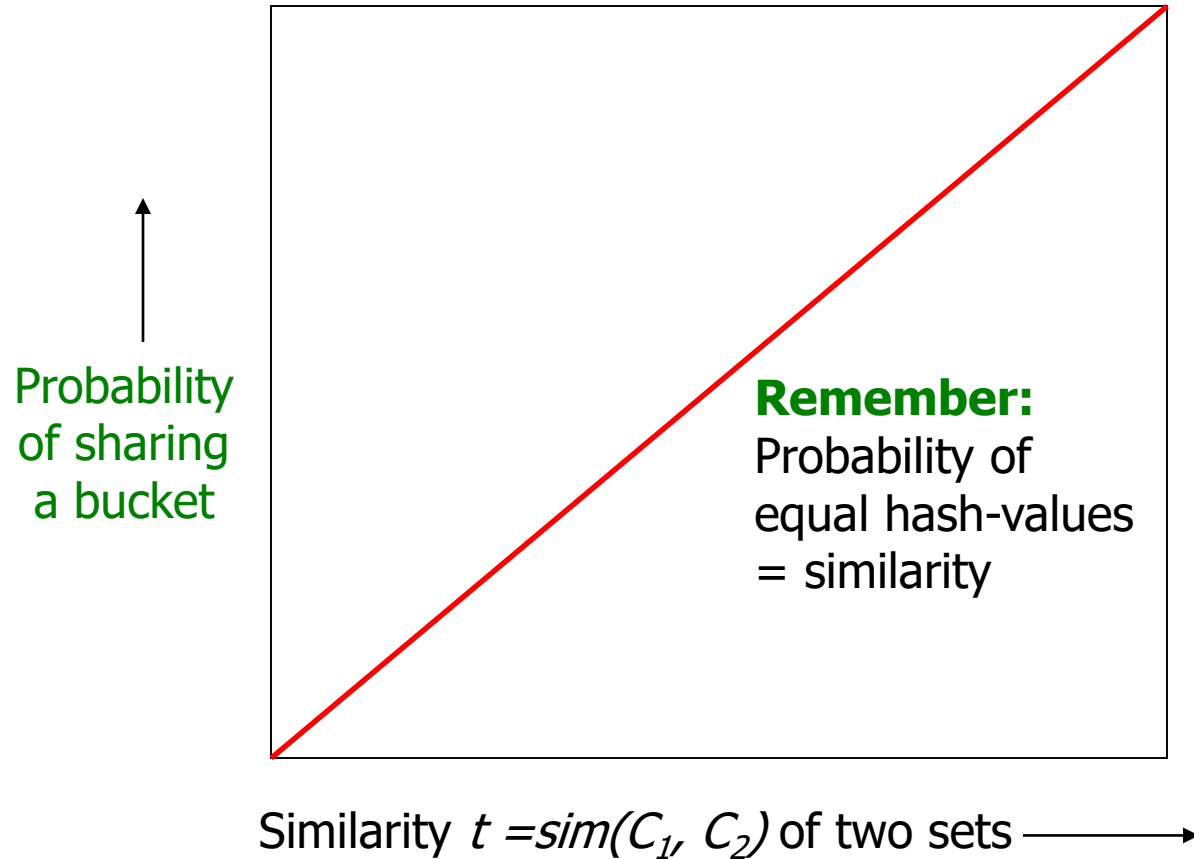
- The number of Min-Hashes (rows of  $M$ )
- The number of bands  $b$ , and
- The number of rows  $r$  per band  
to balance false positives/negatives

**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



# $b$ bands, $r$ rows/band

Columns  $C_1$  and  $C_2$  have similarity  $t$

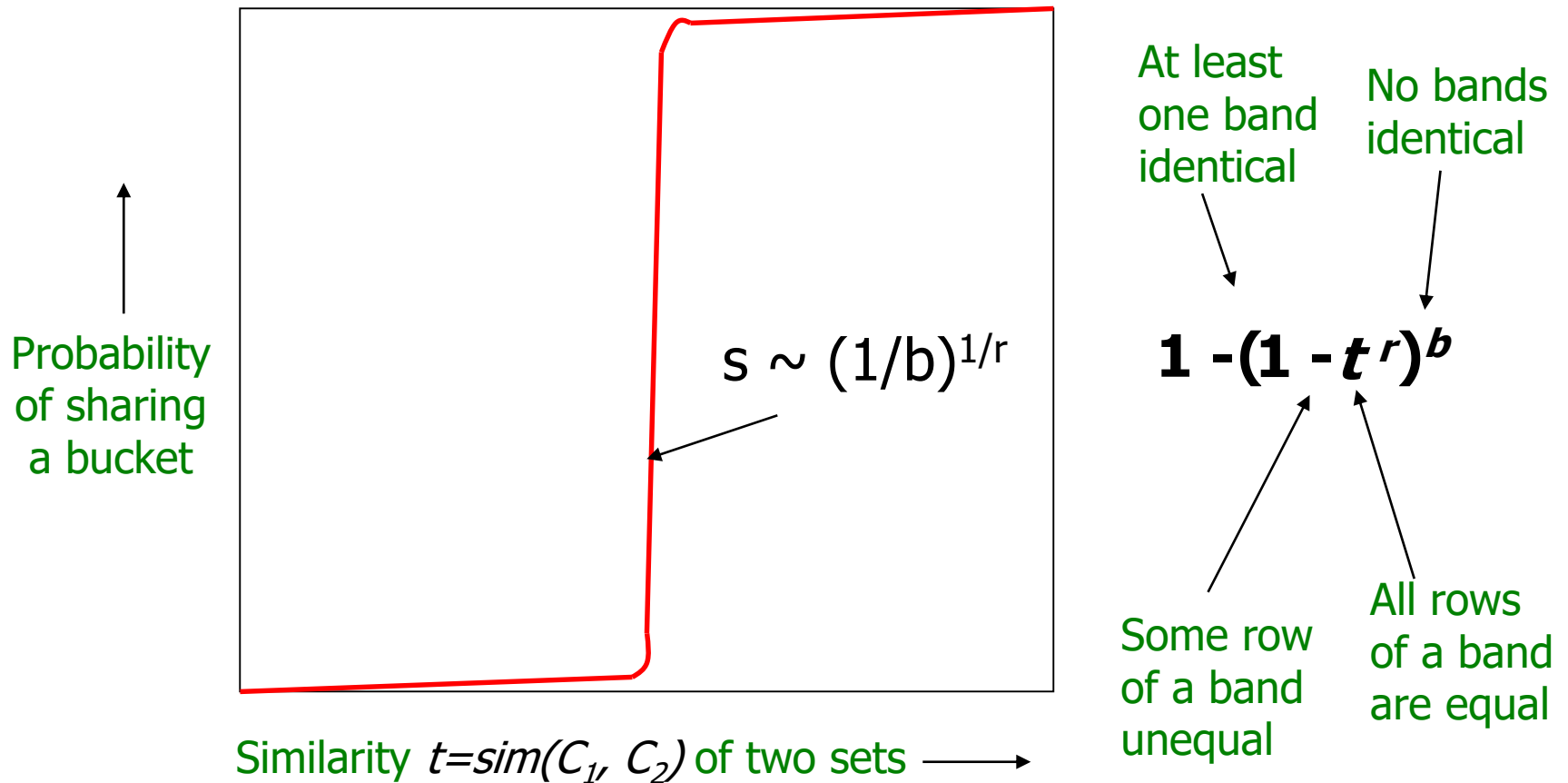
Pick any band ( $r$  rows)

- 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



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

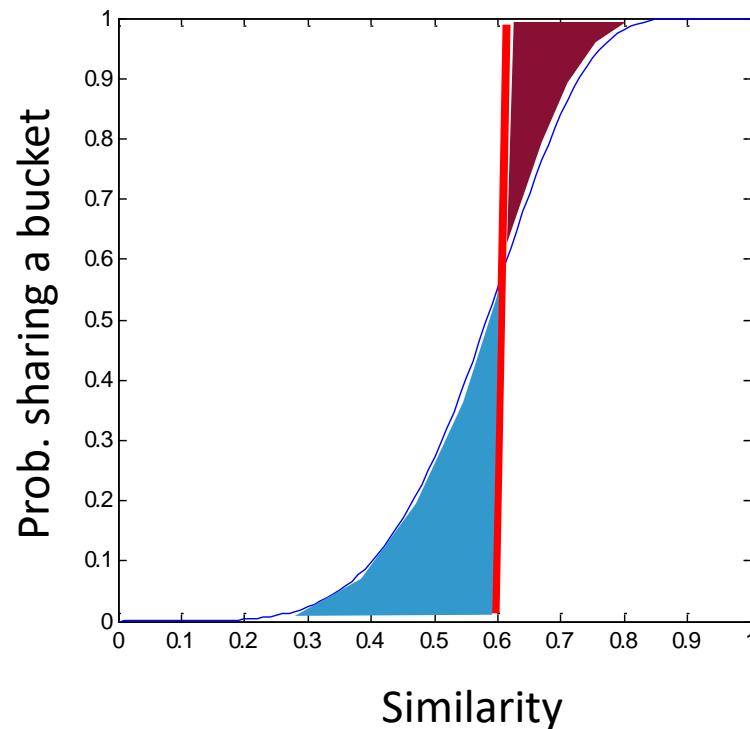
Similarity threshold  $s$

Prob. that at least 1 band is identical:

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

# 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

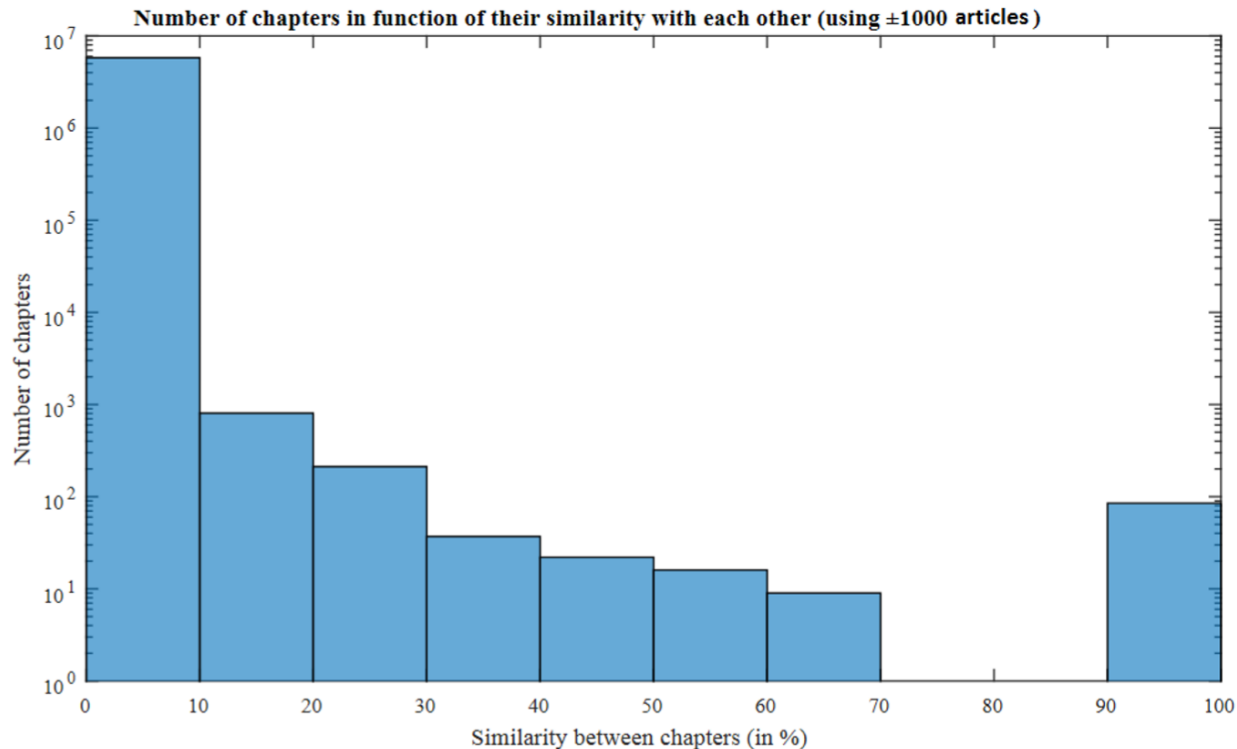
## **A Actuarial science: Initial development**

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 milliseconds
- 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:

$$\begin{aligned} P[ h(x)=h(y) \mid J(x,y) = 90\% ] &= 1-(1-(90\%)^4)^5 \\ &= 0,995\dots \end{aligned}$$

$$\begin{aligned} P[ h(x)=h(y) \mid J(x,y) = 70\% ] &= 1-(1-(70\%)^4)^5 \\ &= 0,75 \dots \end{aligned}$$

# Case Study : Detecting Wiki-plagiarism

Hence, LSH was used

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

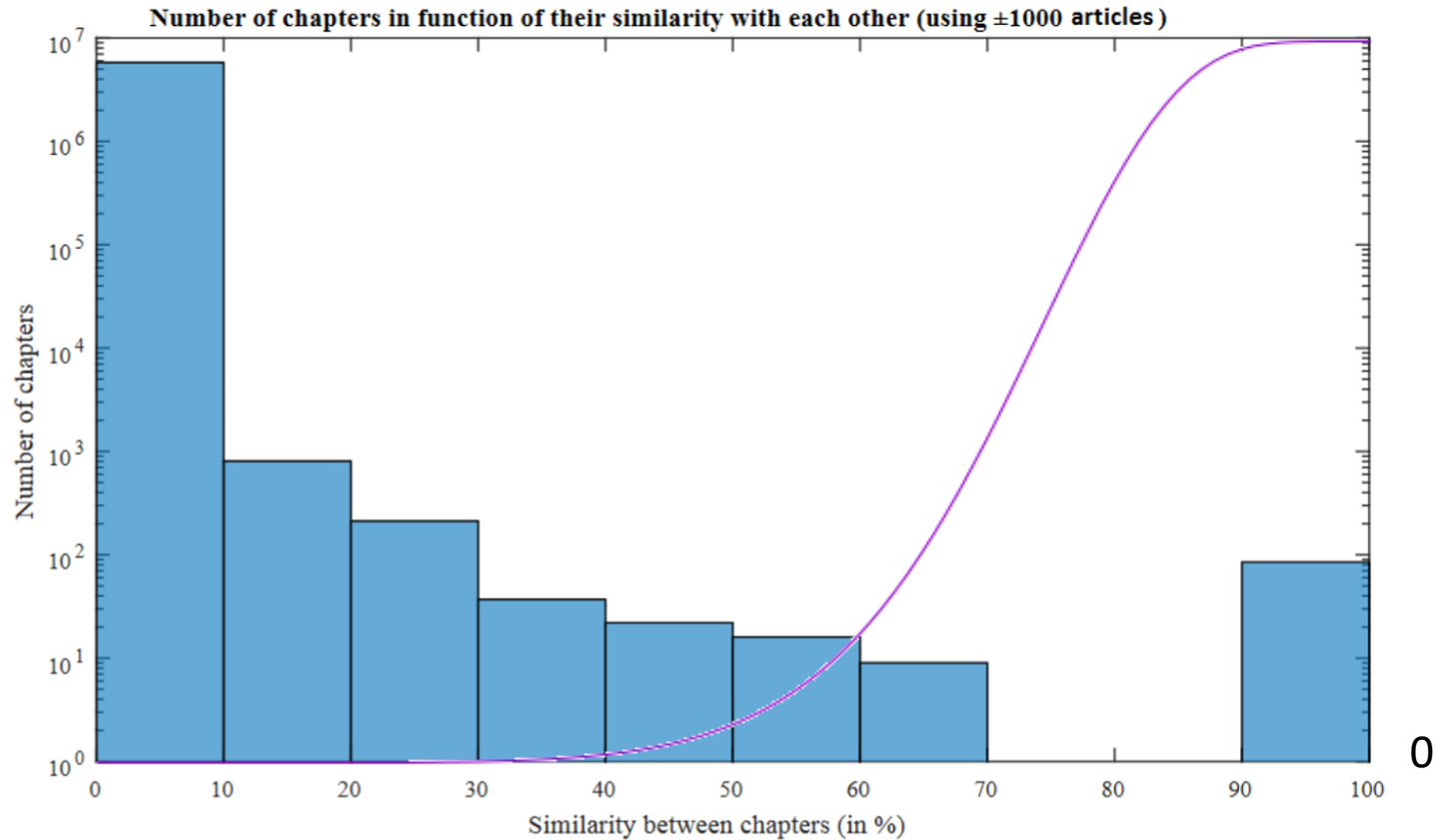
Some math:

7 bands, 7 rows per band:

$$\begin{aligned} P[ h(x)=h(y) \mid J(x,y) = 90\% ] &= 1-(1-(90\%)^7)^7 \\ &= 0,989\dots \end{aligned}$$

$$\begin{aligned} P[ h(x)=h(y) \mid J(x,y) = 70\% ] &= 1-(1-(70\%)^7)^7 \\ &= 0,45 \dots \end{aligned}$$

# Case Study : Detecting Wiki-plagiarism



Super-imposed:  $P( h(x)=h(y) \mid s(x,y)=\text{similarity} )$

# Case Study : Detecting Wiki-plagiarism

Index creation time:

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

Table 1: Execution times to compute the sketches



# Case Study : Detecting Wiki-plagiarism

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

Title <i>source-r-b-n</i>	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  $\text{sim}(x,y)$
- These hash-functions can be combined to reach the needed sensitivity

# Summary: Some new tools ...



What is hot?  
Tracking heavy hitters



Extreme Counting



Anyone like me?  
Similarity search

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

