APPENDIX: HASH FUNCTIONS
(From Kleinberg &Tardos Book: Algorithm Design

Dictionary data type

Dictionary. Given a universe U of possible elements, maintain a subset $S \subseteq U$ so that inserting, deleting, and searching in S is efficient.

Dictionary interface.

- *create*(): initialize a dictionary with $S = \emptyset$.
- *insert*(u): add element $u \in U$ to S.
- *delete*(u): delete u from S (if u is currently in S).
- *lookup*(u): is u in S ?

Challenge. Universe U can be extremely large so defining an array of size $|U|$ is infeasible.

Applications. File systems, databases, Google, compilers, checksums, P2P networks, associative arrays, cryptography, web caching, etc.

HASH FUNCTIONS

Hashing

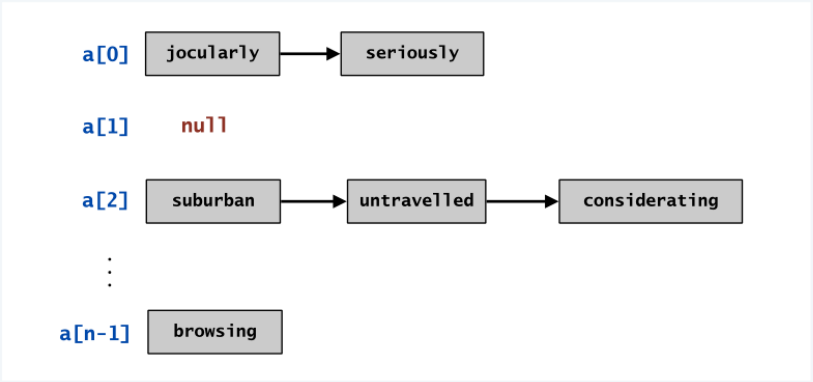
Hash function. $h : U \rightarrow \{ 0, 1, \ldots, n-1 \}$.

Hashing. Create an array a of length n. When processing element u, access array element $a[h(u)]$.

Collision. When $h(u) = h(v)$ but $u \neq v$.

birthday paradox

- A collision is expected after $\Theta(\sqrt{n})$ random insertions.
- Separate chaining: $a[i]$ stores linked list of elements u with $h(u) = i$.



34

HASH FUNCTIONS

Hashing performance $h: U \to [n]$; $|U| \gg n$; $\forall S \subseteq U$ s.t. $|S| = m$

Ideal hash function. Maps m elements uniformly at random to n hash slots.

- Running time depends on length of chains.
- Average length of chain = $\alpha = m / n$.
- Choose $n \approx m \Rightarrow$ expect $O(1)$ per insert, lookup, or delete.

Challenge. Hash function h that achieves $O(1)$ per operation.

Approach. Use randomization for the choice of h.

adversary knows the randomized algorithm you're using, but doesn't know random choice that the algorithm makes

37

Universal Family of Hash Functions

Universal hashing (Carter–Wegman 1980s)

A universal family of hash functions is a set of hash functions H mapping a universe U to the set $\{ 0, 1, \ldots, n-1 \}$ such that

- For any pair of elements $u \neq v$: $\Pr_{h \in H}\left[h(u) = h(v) \right] \leq 1/n$
- Can select random h efficiently.
- Can compute $h(u)$ efficiently.

chosen uniformly at random

Ex. $U = \{ a, b, c, d, e, f \}$, $n = 2$.

	a	b	c	d	e	f
$h_1(x)$	0	1	0	1	0	1
$h_2(x)$	0	0	0	1	1	1

$H = \{h_1, h_2\}$

$\Pr_{h \in H}[h(a) = h(b)] = 1/2$

$\Pr_{h \in H}[h(a) = h(c)] = 1$

$\Pr_{h \in H}[h(a) = h(d)] = 0$

. . .

not universal

	a	b	c	d	e	f
$h_1(x)$	0	1	0	1	0	1
$h_2(x)$	0	0	0	1	1	1
$h_3(x)$	0	0	1	0	1	1
$h_4(x)$	1	0	0	1	1	0

$H = \{h_1, h_2, h_3, h_4\}$

$\Pr_{h \in H}[h(a) = h(b)] = 1/2$

$\Pr_{h \in H}[h(a) = h(c)] = 1/2$

$\Pr_{h \in H}[h(a) = h(d)] = 1/2$

$\Pr_{h \in H}[h(a) = h(e)] = 1/2$

$\Pr_{h \in H}[h(a) = h(f)] = 0$

. . .

universal

38

Universal Family of Hash Functions

Universal hashing: analysis

Proposition. Let H be a universal family of hash functions mapping a universe U to the set $\{ 0, 1, \ldots, n-1 \}$; let $h \in H$ be chosen uniformly at random from H; let $S \subseteq U$ be a subset of size at most n; and let $u \notin S$. Then, the expected number of items in S that collide with u is at most 1.

Pf. For any $s \in S$, define random variable $X_s = 1$ if $h(s) = h(u)$, and 0 otherwise. Let X be a random variable counting the total number of collisions with u.

$$E_{h \in H}[X] = E[\textstyle\sum_{s \in S} X_s] = \sum_{s \in S} E[X_s] = \sum_{s \in S} \Pr[X_s = 1] \leq \sum_{s \in S} \frac{1}{n} = |S| \frac{1}{n} \leq 1$$

linearity of expectation — X_s is a 0–1 random variable — universal

Q. OK, but how do we design a universal class of hash functions?

Universal Family of Hash Functions

Designing a universal family of hash functions

Modulus. We will use a prime number p for the size of the hash table.

Integer encoding. Uniquely identify each element $u \in U$ with a base-p integer of r digits: $x = (x_1, x_2, \ldots, x_r)$.

distinct elements have different encodings

Hash function. Let A = set of all r-digit, base-p integers. For each $a = (a_1, a_2, \ldots, a_r)$ where $0 \le a_i < p$, define

$$h_a(x) = \left(\sum_{i=1}^{r} a_i x_i \right) \bmod p$$

maps universe U to set $\{ 0, 1, \ldots, p-1 \}$

Hash function family. $H = \{ h_a : a \in A \}$.

Universal Family of Hash Functions

$U \subseteq [p]^r$

Designing a universal family of hash functions

$h_a(x) = \sum_{i=1}^{r} a_i x_i \mod p$

Theorem. $H = \{ h_a : a \in A \}$ is a universal family of hash functions.

Pf. Let $x = (x_1, x_2, \ldots, x_r)$ and $y = (y_1, y_2, \ldots, y_r)$ encode two distinct elements of U.

We need to show that $\Pr[h_a(x) = h_a(y)] \le 1/p$.

- Since $x \neq y$, there exists an integer j such that $x_j \neq y_j$.
- We have $h_a(x) = h_a(y)$ iff

$$(1) \qquad a_j \underbrace{(y_j - x_j)}_{z} \equiv \underbrace{\sum_{i \neq j} a_i (x_i - y_i)}_{m} \mod p$$

Principle of Deferred D
This is fixed before
a_j is chosen i.u.a.r.

- Can assume a was chosen uniformly at random by first selecting all coordinates a_i where $i \neq j$, then selecting a_j at random. Thus, we can assume a_i is fixed for all coordinates $i \neq j$.
- Since p is prime, $a_j z \equiv m \mod p$ has at most one solution among p possibilities. ← see lemma on next slide
- Thus $\Pr[h_a(x) = h_a(y)] \le 1/p$. ▪ → it is the a_j value s.t. (1) is satisfied

41

Universal Family of Hash Functions

Number theory fact

Fact. Let p be prime, and let $z \not\equiv 0 \bmod p$. Then $\alpha z \equiv m \bmod p$ has at most one solution $0 \le \alpha < p$.

Pf.

- Suppose $0 \le \alpha_1 < p$ and $0 \le \alpha_2 < p$ are two different solutions.
- Then $(\alpha_1 - \alpha_2)\, z \equiv 0 \bmod p$; hence $(\alpha_1 - \alpha_2)\, z$ is divisible by p.
- Since $z \not\equiv 0 \bmod p$, we know that z is not divisible by p.
- It follows that $(\alpha_1 - \alpha_2)$ is divisible by p.
- This implies $\alpha_1 = \alpha_2$. ▪

here's where we use that p is prime

Bonus fact. Can replace "at most one" with "exactly one" in above fact.

Pf idea. Euclid's algorithm.

Universal Family of Hash Functions

Universal hashing: summary

Goal. Given a universe U, maintain a subset $S \subseteq U$ so that insert, delete, and lookup are efficient.

Universal hash function family. $H = \{ h_a : a \in A \}$.

$$h_a(x) = \left(\sum_{i=1}^{r} a_i x_i \right) \bmod p$$

- Choose p prime so that $m \le p \le 2m$, where $m = |S|$.
- Fact: there exists a prime between m and $2m$. ← can find such a prime using another randomized algorithm (!)

Consequence.

- Space used = $\Theta(m)$.
- Expected number of collisions per operation is ≤ 1

  $\Rightarrow$ $O(1)$ time per insert, delete, or lookup.

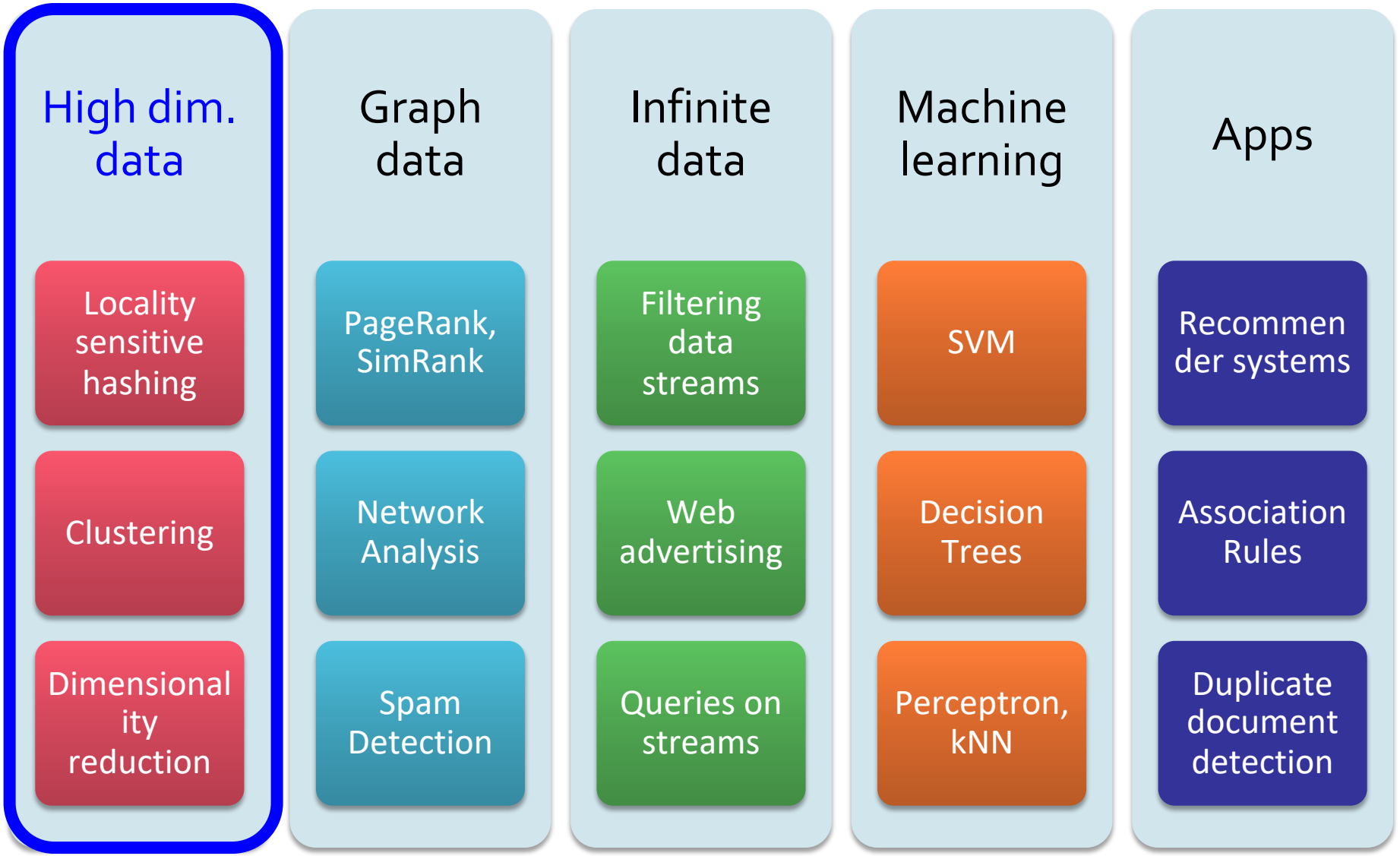
43

Finding Similar Items in Large Data Sets

Lecturer: **Andrea Clementi**
University of Rome *Tor Vergata*

Note: These slides are an adaptation, with more details, of the slides by Jure Leskovec, Anand Rajaraman, Jeff Ullman available at http://www.mmds.org

New thread: High dim. data

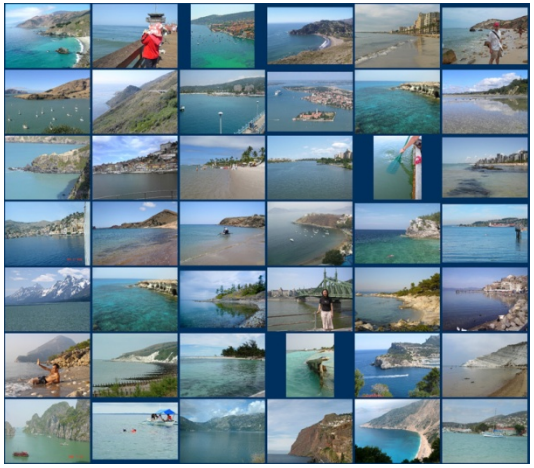


[Hays and Efros, SIGGRAPH 2007]

Application: Scene Completion Problem

target item

many items



selection of *similar* items

closest item

A Common Metaphor

Many problems can be expressed as:

finding "similar" sets

Examples:

- **Pages with a large fraction of similar words**
  - For duplicate detection, classification by topic
- **Customers who purchased a large fraction of similar products**
  - Products with similar customer sets
- **Images with similar features**
  - Users who visited similar websites

High-Dimensional Data and Distance

Input: *High-Dimensional* data points $\boldsymbol{x_1, x_2, \ldots, x_N}$

- Image x is a *long* vector of pixel colors

$$x = \begin{bmatrix} 1 & 4 & 1 \\ 0 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} \rightarrow [1\ 4\ 1\ 0\ 2\ 1\ 0\ 1\ 0] \in \{0,1,\ldots,c\}^h \text{ , } (h = \text{data dimension})$$

Introduce a distance function $\boldsymbol{d(x_1, x_2)}$

- Which quantifies the "distance" between any pair of data points $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$

Goal: Find ***all* pairs of data points** $\boldsymbol{(x_i, x_j)}$ that are within some *distance threshold* $\boldsymbol{d(x_i, x_j) \leq s}$

High-Dimensional Data and Distance

Goal: Find **all pairs of data points** (x_i, x_j) that are within some distance threshold $d(x_i, x_j) \leq s$

Naïve solution would take $O(N^2 * h)$ ☹

where N is the number of data points and h is the data dimension.

MAGIC: **This can be done in $O(N * h')$ with $h' << h$, *with high confidence* !!**

Why? How?

Our Application: Finding Similar Documents

Goal: Given a large number (N in the millions or billions) of Docs, find "near duplicate" pairs

Applications:

- Mirror websites, or approximate mirrors
  - Don't want to show both in search results
- Similar news articles at many news sites
  - Cluster articles by "same story"

Technical Problems:

- Many small *pieces* of one Doc can appear out of *order* in another
- Too many Docs to compare *all* pairs
- Docs are so large or so many that they cannot fit in main memory

Choosing the *right* Distance Measures

For each application, we first need to define what "**distance**" means

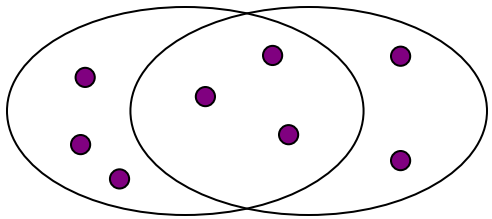
Our Choice: *Jaccard* distance/similarity

over a family of subsets

- The ***Jaccard* similarity** of two **sets** is:

$$sim(C_1, C_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$$

- **Jaccard distance: $d(C_1, C_2) = 1 - sim(C_1, C_2)$**



3 in intersection
8 in union
Jaccard similarity= 3/8
Jaccard distance = 5/8

Application: Documents

A First Important Question: How apply ***Jaccard*** *distance/similarity* to Docs (i.e. to finite-size *strings*)?

Doc ∈ U* where U is the Alphabet

J.Similarity $sim(C_1, C_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$

Docs ⇔ **Sets** **?** 🤔

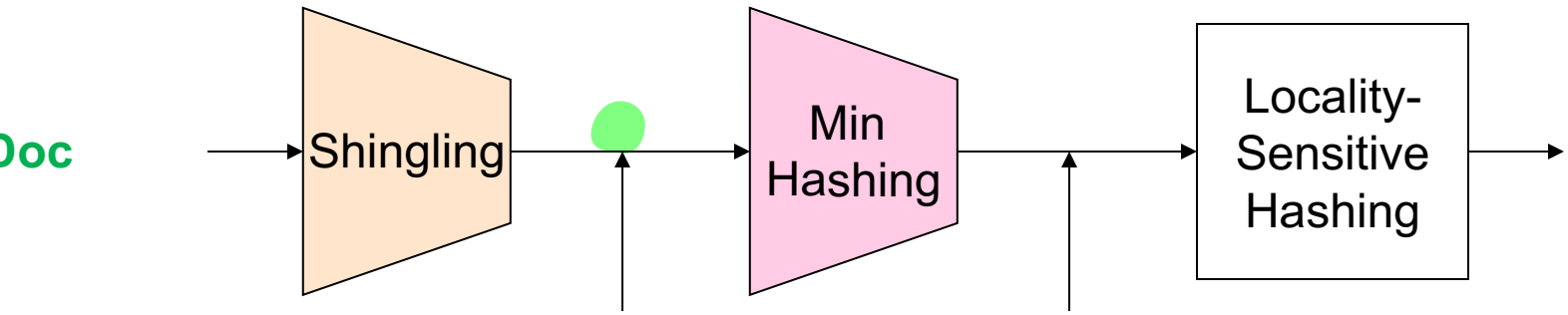
3 Essential Steps for Similar Docs

1. ***Input:*** A Huge *Universe* of *Docs*

2. ***Shingling:*** *Convert* Docs to (large) Sets
3. ***Min-Hashing:*** *Convert* large Sets to short Signatures, while preserving ***J.*** *Similarity*
4. ***Locality-Sensitive Hashing:*** Focus on pairs of Signatures likely to be from J.similar Docs (overcome the Θ(N^2) barrier!!)

5. **Output: Candidate *Doc* pairs!**

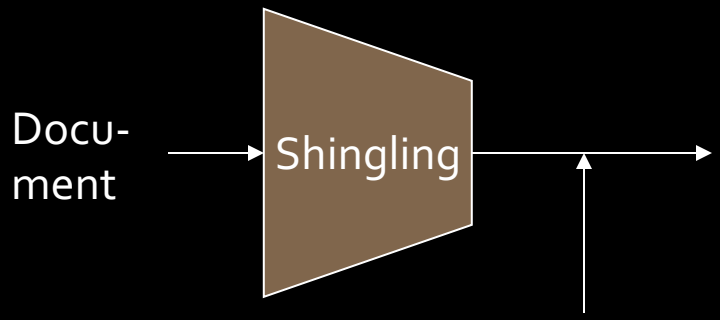
The Big Picture



The **Set** of strings of length ***k*** that appear in the Doc

Signatures: short integer vectors that represent the **Sets**, and reflect their J.similarity

Candidate pairs: those pairs of signatures that we need to test for similarity



Shingling

Step 1: ***Shingling:*** Convert documents to sets

Documents as High-Dim Data

Step 1: *Shingling:* Convert Documents to Sets

Simple approaches:

- Document = set of *words* appearing in document
- Document = set of "important" *words*

Don't work well for this application.....😟

Why?

Need to account for ordering of words!

A different approach: Shingles!

Definition of Shingles

A *k*-shingle (or *k*-gram) for a Doc is a sequence of ***k*** **tokens** that appears in the Doc

- **Tokens** can be ***characters***, *words* or something else, depending on the application:
  - **U** ≡ Universe of ***all*** possible Tokens
- **Example**: Assume Tokens = **characters and k=2**
  - Then for Doc $\mathbf{D_1}$ = abcab → Set of 2-shingles is $\mathbf{S(D_1)}$ = {ab, bc, ca}

- **Option:** Shingles as a bag (multiset). Count ab twice: $\mathbf{S'(D_1)}$ = {ab, bc, ca, ab}

Doc Representation by Shingles

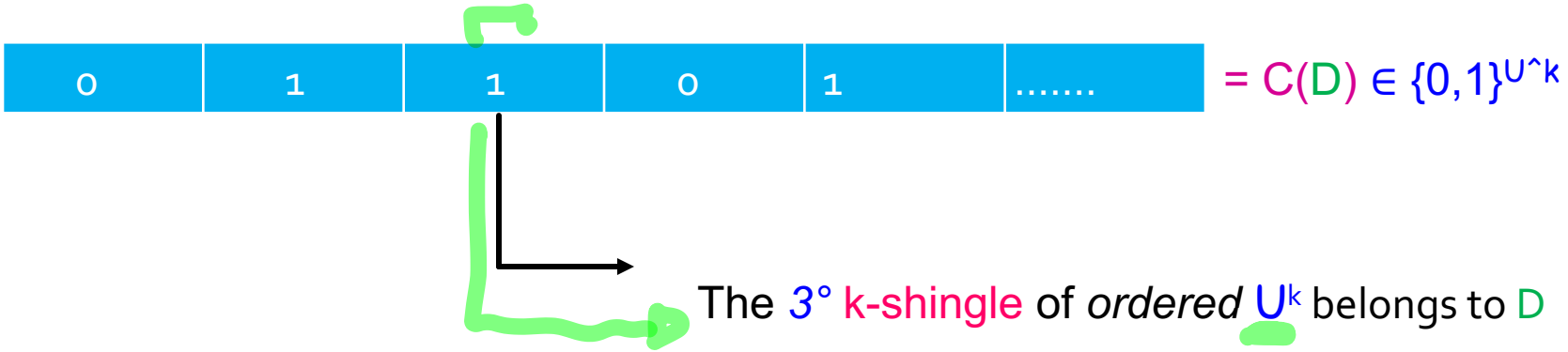
Each Doc D ∈ U^* is represented as the **set** of its **k-shingles C=S(D)**

Equivalently:

each Doc is represented as a *long binary vector* **C=S(D)** having one component for each element of the set U^k of *all* possible *k*-shingles

- Each unique shingle is a *dimension* (*component*) of **C=S(D)**
- ***Note:*** *Vectors* have several dimensions but are very sparse

Example:



Similarity Metric for Shingles

Our strategy:

(a) Doc D is represented as the **subset** of its **k-shingles C_j=S(D)**

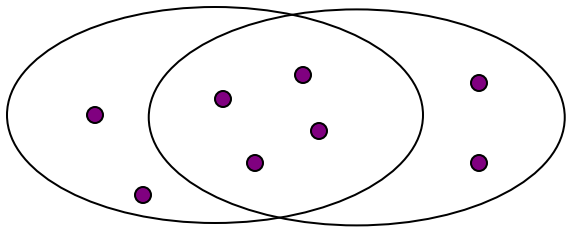
Equivalently, each **Doc** is a **binary vector** in the space of all possible *k*-shingles $\cup^k$

(b) A natural ***similarity*** measure **(=1- distance)** is the ***Jaccard similarity*** (and, thus, ***distance***):

$J.sim(D_1, D_2) \equiv |C_1 \cap C_2| / |C_1 \cup C_2| = Pr(C_1 \cap C_2 \mid C_1 \cup C_2)$

$dist(D_1, D_2) \equiv 1 - sim(D_1, D_2)$

PROBABILISTIC INTERPRETATION



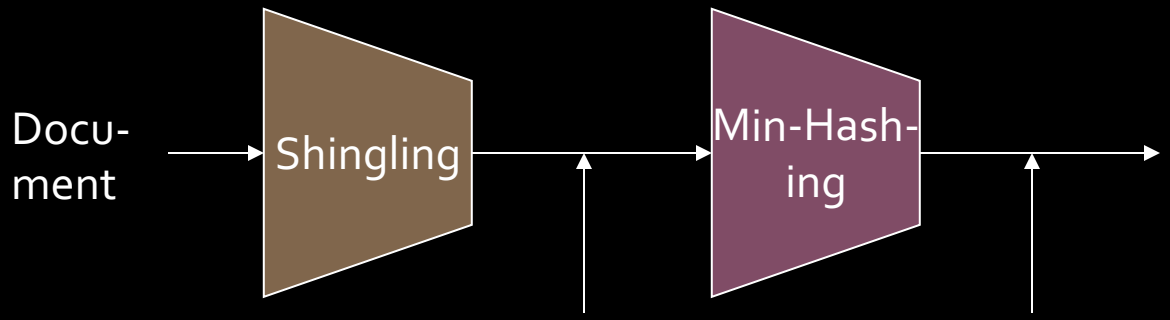
Working Hypothesis

Docs that have lots of k-shingles in common have *similar* text, *even* if the text appears in different order

Practice: You must pick ***k*** large enough, or most Docs will have most shingles → large intersection → large (not relevant) ***J. Sim.***

Parameter tuning from Real World:

- ***k*** = 5 is **OK** for *short* Docs
- ***k*** = 10 is better for *long* Docs



The set of strings of length *k* that appear in the doc-ument

Signatures: short integer vectors that represent the sets, and reflect their similarity

MinHashing

Step 2: ***Min-hashing:*** Convert **large sets** (i.e. long binary sparse *vectors*) to **short signatures**, while **preserving similarity**

Recall: Motivation for Min-Hashing

Suppose we need to find near-duplicate **Docs** among $N = 10^6$ **Docs** and, hence, among 10^6 *k−shingles C=S(D)*

Then....

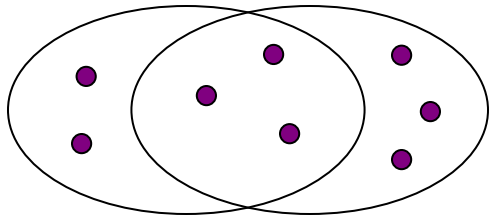
Min-Hashing does not avoid $\Theta(N^2) \simeq 5*10^{11}$ pair *comparisons!*

However ...

It makes *each* pair comparison (***J.*** *similarity* computations) much more **efficient**

Note: if **C** $\in \{0,1\}^{U\hat{}k}$ → cost(Naïve-Comparison) = $\Theta(|U|^k)$. For alphabet size $|U| = 27$ and $k = 10$ → .....😢

Encoding Sets as Binary Vectors



Many ***similarity*** problems can be formalized as **finding subsets** that have significant **intersection (recall *J. Sim.*):**

(a) Encode sets using binary vectors

- One dimension (component) per *element* (shingle) in the *Universe* U^k

(b) Interpret: (i) *set intersection* as bitwise ***AND***

(ii) *set union* as bitwise ***OR***

Example: C_1 = 10111; C_2 = 10011

- Size of intersection **= 3**; size of union **= 4**,
- **Jaccard similarity** (not distance) **= 3/4**
- **Distance: $d(C_1,C_2) = 1 – (J.Sim(C_1,C_2)) = 1/4$**

From Sets to Large, Sparse Boolean Matrices

Rows = elements (shingles) = vector components (**n.** of raws = **m** = $|U|^k$)

Columns = Sets = Docs (N col)

- **1** in row ***s*** and column ***e*** *iff* ***s*** $\in$ ***e***
- Column similarity is the **J.sim** of the corresponding sets
- **Remark:** Typical matrix **M** is sparse!

Each Doc is a column:

- **Example: sim(C_1 ,C_2) = ?**
  - Size of intersection = 3; size of union = 6, *Jaccard similarity* (not distance) = 1/2
  - **d(C_1,C_2) = 1 – (J.Sim (C_1,C_2)) = 1/2**

Matrix **M:**

|Docs| = N

Shingles				
	1	1	1	0
	1	1	0	1
	0	1	0	1
	0	0	0	1
	1	0	0	1
	1	1	1	0
	1	0	1	0

$m = |U|^k$

Outline: Finding Similar Columns

So far:

- Docs → ***Subsets*** of Shingles
- Represent *Subsets* as *binary columns* of a *matrix* **M**

Next goal: Find similar columns by computing *small* column ***signatures*** (**Min-Hashing**)

Technical goal: define a ***good*** signature *algorithm* so that

J.Sim of Columns ≈ **"Similarity" of Signatures**

Outline: Finding Similar Columns

Next Goal: Find similar columns using small signatures

Our Approach:

- **1) Signatures of columns =** small *summaries* of columns
- **2) Examine pairs of signatures** to find similar columns
  - **Essential:** Similarities of signatures and columns must be related
- **3) Optional:** Check that columns with similar signatures are really *similar*

Warnings:

- Comparing *all pairs* may take too much time: **Task for LSH**
- These methods can produce *false negatives*, and even *false positives* (if the optional check Step (3) is not made)

Hashing Columns (Signatures)

Key idea: "hash" each m-size column ***C*** to a small ***signature h(C)***, such that: $|h(C)| << |C| = m$

- **(1)** ***h(C)*** is small enough that the signature fits in RAM (Fast Memory)
- **(2)** ***sim(C_1, C_2)*** is "*close*" to the *equivalence* of signatures ***h(C_1)*** and ***h(C_2):***

Goal: Find a hash function *h(·)* such that:

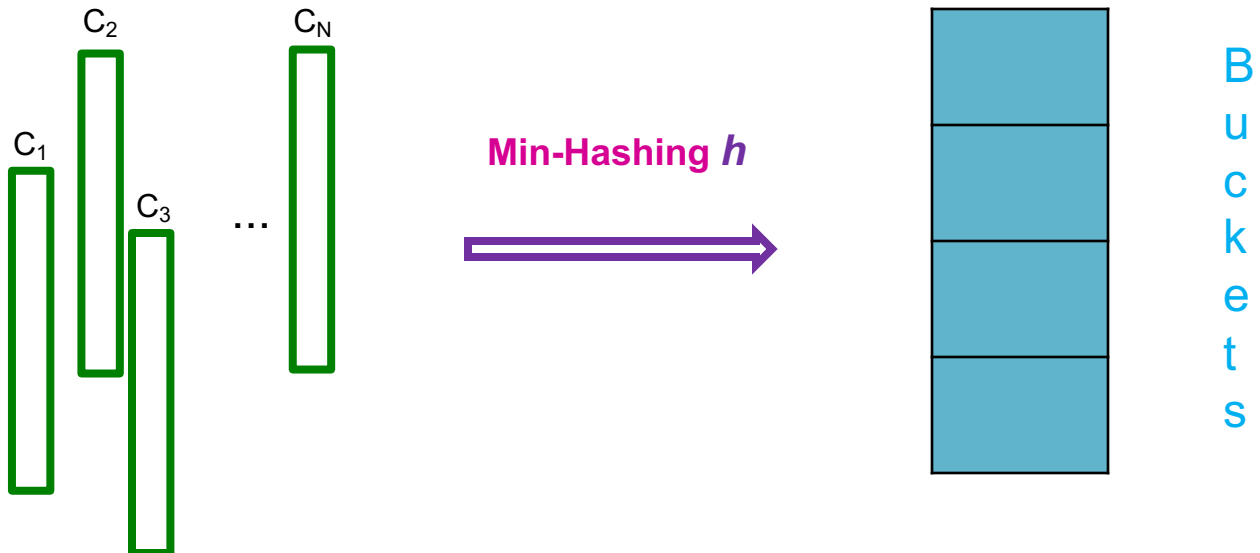
- If ***sim(C_1,C_2)*** is high, then **with high prob.** ***h(C_1) = h(C_2)***
- If ***sim(C_1,C_2)*** is low, then **with high prob.** ***h(C_1) ≠ h(C_2)***

Hashing Columns (Signatures)

Goal: Find a hash function *h(·)* such that:

- If ***sim(C_1,C_2)*** is high, then **with high prob.** ***h(C_1) = h(C_2)***
- If ***sim(C_1,C_2)*** is low, then **with high prob.** ***h(C_1) ≠ h(C_2)***

Intuition: Hash Docs into ***Buckets***. Expect that "most" pairs of near duplicate Docs hash into the same **Bucket**!



Min-Hashing

Goal: Find a hash function $h(\cdot)$ such that:

- If $sim(C_1,C_2)$ is high, then **with high prob.** $h(C_1) = h(C_2)$
- If $sim(C_1,C_2)$ is low, then **with high prob**. $h(C_1) \neq h(C_2)$

Clearly, the hash function depends on the *similarity* metric:

- Not all similarity metrics have a suitable hash function

Good News: There is a suitable hash function for the Jaccard similarity: It is called **Min-Hashing**

Min-Hashing

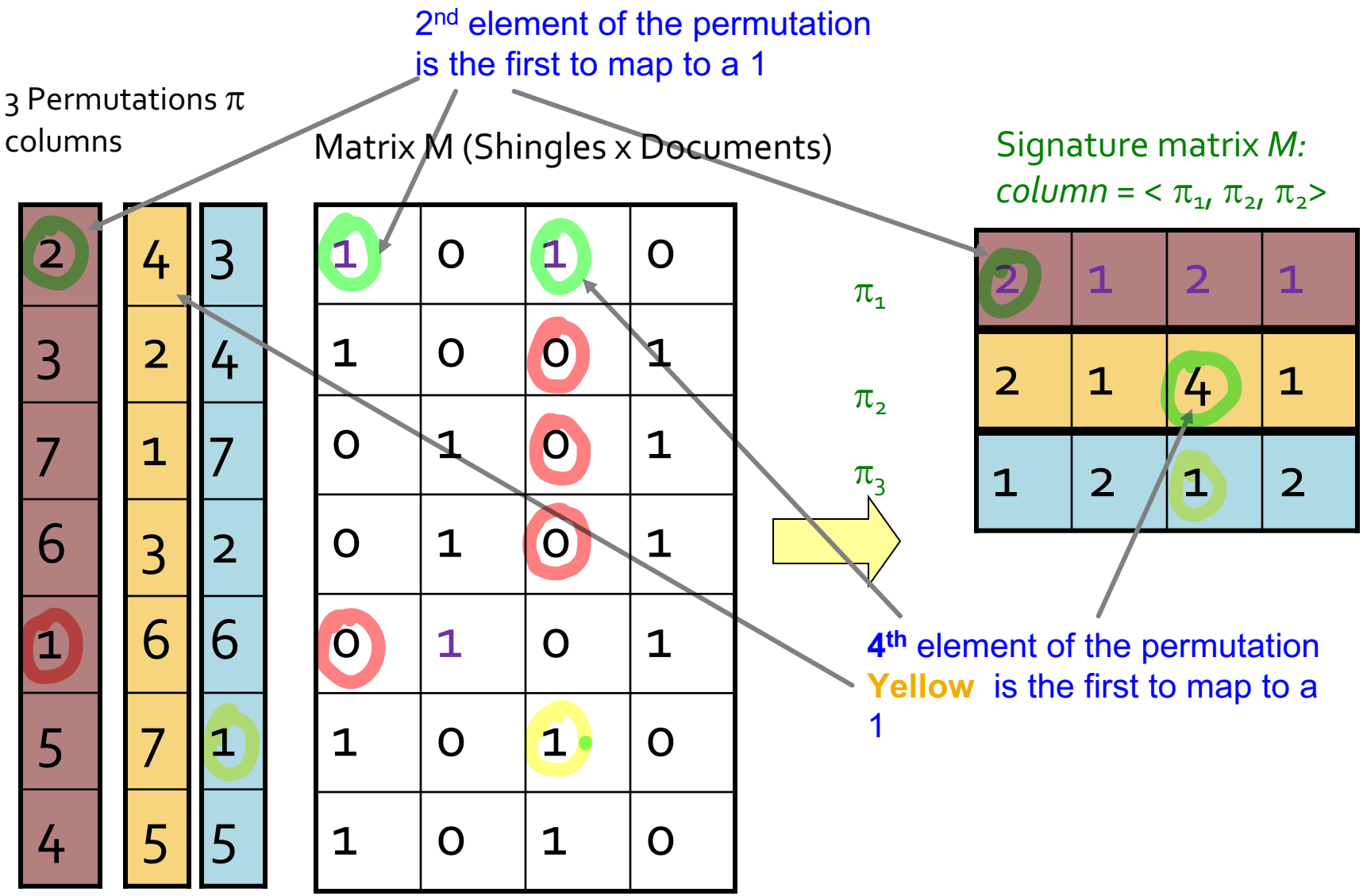
Imagine the *rows* of matrix **M** permuted under ***random permutation*** π

Def. For each column C, define: **Min-Hash function $h_\pi(C)$** ≡ the **index** of the **first** (in the permuted order π) row in which column **C** has value **1** (we named it as **min(π(C))**)

Practical Strategy: Use several (e.g., 100) independent rnd hash functions (that is, rnd permutations) to create a *signature* of **C**

see later....

Min-Hashing Example with N=4, m= 7



MEMORY - COST (MIN-HASH)

$h_\pi(C) \longrightarrow [m] \equiv \{1, 2, \ldots, m\}$

where $|C| \equiv |U|^k \equiv m$

and SPACE $(h_\pi(C)) \equiv \Theta(\lg m)$

$m \xrightarrow{h_\pi} \Theta(\lg m)$

$\sqcup$ m

it is just a row-index of M

The Min-Hash Property

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

J-SIMILARITY PRESERVING

THM : Fix C_1, C_2. Choose *u.a.r.* a permutation π. Then,

- $\Pr_\pi [h_\pi(C_1) = h_\pi(C_2)] = J.sim(C_1, C_2)$

1/3

Proof.

- Let **X** be a Doc (column of shingles), $y \in X$ is one of its shingle. Then: $\Pr[\pi(y) = \min(\pi(X))] = 1/|X|$ since:
  - It is equally likely that any row $y \in X$ is mapped to the ***min*** index

bit = 1

- Let y be s.t. $\pi(y) = \min(\pi(C_1 \cup C_2))$ π is rnd uniq
- **Then either:** $\pi(y) = \min(\pi(C_1))$ if $y \in C_1$, **or** → UNION
  $\pi(y) = \min(\pi(C_2))$ if $y \in C_2$

One of the two cols had to have 1 at position ***y***

- So the prob. that **both** are **true** is the prob. of $y \in C_1 \cap C_2$
- $\Pr[\min(\pi(C_1)) = \min(\pi(C_2))] = |C_1 \cap C_2| / |C_1 \cup C_2| = sim(C_1, C_2)$

$h(C_1)$ $h(C_2)$

$$= \frac{|C_1 \cap C_2| \cdot 1/|X|}{|C_1 \cup C_2| \cdot 1/|X|}$$

Similarity for Signatures

Given any two columns C_1 and C_2 , **Main THM** → the Prob. that C_1 and C_2 get the same Min-Hash signature, i.e. $h_\pi(C_1) = h_\pi(C_2)$, **equals** the value of **J.*sim*(C_1, C_2)**

Increase the "confidence"!? How?

Simple Idea from *Statistics*: Exploit *concentration* on the *Expected Value*: use multiple, mutually-independent Min-Hash **signatures *SIG*(C) = <** $h_{\pi 1}(C)$, $h_{\pi 2}(C)$,..., $h_{\pi t}(C)$> for some fixed **t >>1** and set:

DEF. *Similarity* Sign-Sim(C_1, C_2) of two (vector) signatures *SIG*(C_1) and ***SIG*(C_2) = the fraction of the chosen** Min-Hash signatures **in which they *agree***

AVERAGE $t \to t\infty$ EXPECTATION

Formally,

Def. binary r.v. **Z = 1 iff "min($\pi(C_1)$) = min($\pi(C_2)$)"** .

Goal. Estimate **Pr[Z=1]**

THM 1

we KNOW Pr[Z=1] III S.SIM(C_1,C_2)

Min-Hashing Example

Permutation π

2	4	3
3	2	4
7	1	7
6	3	2
1	6	6
5	7	1
4	5	5

Input matrix (Shingles x Documents)

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

Signature Matrix *SIG(π,C)*

2	1	2	1
2	1	4	1
1	2	1	2

Similarities: $|C_i \cap C_j| / |C_i \cup C_j|$

	1-3	2-4	1-2	3-4
Col/Col	6/7	0.75	0	0
Sig/Sig	2/3	1.00	0	0

Min-Hash: The Algorithm

Rnd Algorithm ***Doc-Pair Check***

Input: Columns $C_1, C_2 \in \{0,1\}^h$; Confidence Parameter **t**

For j=1 to t

- Choose u.a.r. a raw permutation $\pi_j \in \Pi_m$
- Compute hash $h_{\pi j}(C_1)$ and $h_{\pi j}(C_2)$

Return **Sign-Sim(C_1, C_2) = $|\{j : h_{\pi j}(C_1) = h_{\pi j}(C_2)\}| / t$**

Note: J.*Sim*(C_1, C_2) ≠ Sign-Sim(C_1, C_2) but… Because of the Min-Hash property (Main THM): J.*Sim*(C_1,C_2) ≡ E_π(Sign-Sim(C_1, C_2)), so

Coroll. 1 (of Main THM). Sign-Sim(C_1,C_2) → J.*Sim*(C_1,C_2) as t → ∞

Important Note: since the **t** hash functions are *mut. indep.* → we have *concentration results* on | Sign-Sim(C_1,C_2) - J.*Sim*(C_1,C_2) |

Min-Hash: Space Complexity

Pick t=100 random permutations of the **m rows**

Think of ***SIG*(C)** as a *short* column vector of a signature Matrix

- ***SIG*(i,C) =** the index of the first row that has a 1 in column **C** according to the *i*-th rnd permutation,

$$SIG(i,C) \text{ is an index!} \quad \Theta(\log |C|) = \Theta(\log m))$$

Note: The sketch (signature) ***SIG*(*,C)** of **C** is **small: ~100 bytes (in general: $\Theta(t*\log m)$)!**

Recall: $m = |U|^k$ is the number of *all* possible **k-shingles**

We achieved our goal: **We "compressed" long-bit vectors C into short signatures *SIG*(*,C) (m → t log m)**

Implementation Trick for Min-Hash Signatures

Min-Hash function $h_\pi(C)$ requires to generate several independent rnd permutations π of the raw set U^k ($|U|^k = m$)

Permuting m rows even once is too expensive !

Idea: Replace **t Row Permutations** with **t Row Hash Functions**

Ordering under an ***Hash Function* f** gives an (*almost*) random π

Pick **t = 100 hash** functions ***f_i : [m]→[m]*** ***(there might be few collisions: neglect them!)***

raw index of the first 1 "after" hashing f_i

$\Theta(\lg m)$ bits

Original Binary Matrix M

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

m raws = k-shingles

N docs = columns

Min-Hash using f_i

Signature Matrix *SIG*(i,C)

f_1						
f_2						
…						
f_t						

N docs = columns

Algorithm for *Signature* Matrix *SIG(i,C)*

FOR each column ***C=1…N*** and (hash-function) **i=1…t** DO
- Initialize the signature matrix ***SIG(i,C) = ∞***

FOR each row **j=1…m** of the original matrix **M** DO
- FOR each column ***C=1…N*** DO
  - IF ***M(j,C) = 1*** THEN (**)
    - FOR each (hash-function) **i=1…t** DO
    - Compute the i-hash of raw **j**: $f_i(j)$
    - IF $f_i(j) < SIG(i,C)$ THEN update $SIG(i,C) \leftarrow f_i(j)$

MIN - Computation

SEE PREVIOUS LESSONS!

Performance Analysis: Θ(mN) iterations

Each iteration requires **t** hash computations $f_i(j)$ *iff* **Condition** (**) holds.

Good News: i) No full computations of rnd permutations π_m ;
ii) Original Matrix M is sparse → (**) is unlike !

Bad News: Hash may generate collisions (no permutation)

How to pick a random hash function h(x)?

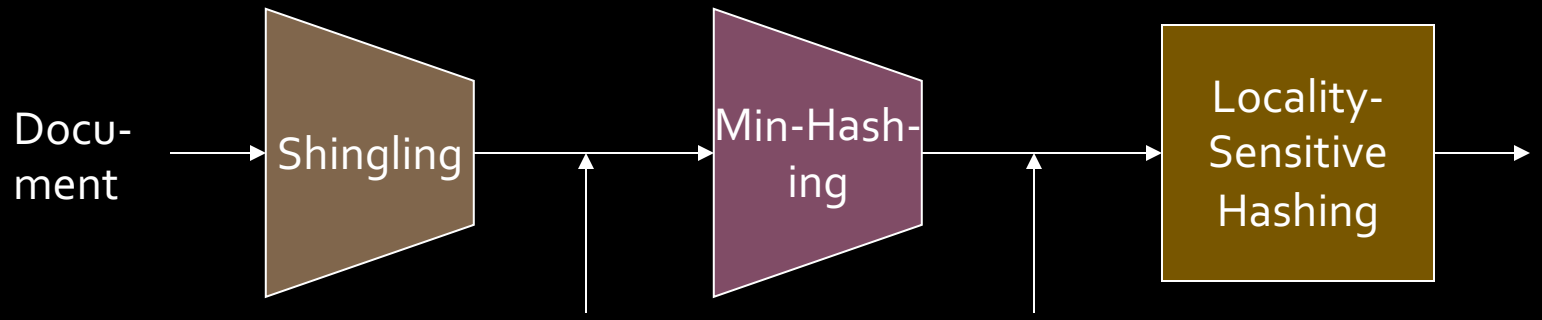
Universal hashing:

$h_{a,b}(x)=((a \cdot x+b) \bmod p) \bmod m$

where:

a,b … random integers

p … prime number (p > m)



Locality Sensitive Hashing

Step 3: ***Locality-Sensitive Hashing:***
Focus on pairs of signatures likely to be from similar documents

Motivation for LSH

Suppose we need to find *near*-duplicates among $N = 10^6$ Docs

Naïvely, we would have to compute pairwise

J. similarities for **every pair of Docs**

$N(N-1)/2 \approx 5*10^{11}$ comparisons !!!

- At 10^5 *secs/day* and 10^6 comparisons/sec, it would take **5 days**
- For $N = 10^7$, it takes more than a year…!

Remark: **Min-Hashing** does not avoid $\Theta(N^2)$ pair *comparisons*. It makes *each* pair comparison (*J. similarity* computations) much more **efficient:**

Original Matrix M(k,C) → **Signature Matrix *SIG*(i,C)**

m raws → **t log m raws**

index size

hash functions

Local Sensitive Hashing (LSH)

Goal: Find Doc pairs with similarity at least ***s*** (for some ***similarity threshold s***, e.g., ***s*** = 0.8)

└ INPUT PARAMETER

LSH – General idea: Use a function ***f(x,y)*** that tells whether ***x*** and ***y*** is a ***candidate*** *Doc Pair:* a pair of Docs whose *similarity* must be evaluated in a deeper way.

General Idea of LSH: Hashing Docs Signature

Define **Candidate Function *f(x,y):***

- **Consider (Min-Hash) Signature Matrix *SIG*(i,C) (i=1…t; C=1…N)**

SIG(i,C)*

2	1	4	1
1	2	1	2
2	1	2	1

NOTE: From now on, we "forget" matrix M, and we only work with matrix SIG(i,C)*

- Hash columns (Docs) of *SIG*(i,C) to *many* Buckets
- Each pair of Docs that hashes into the same Bucket is a candidate pair: check it!

LSH STRATEGY

LSH: Step I (Candidates from Min-Hash Signatures)

Choose a *suitable* **similarity threshold** s ($0 < s < 1$)

CRITERION

DEF. Columns x and y of $SIG(*,*)$ are a **candidate pair** if their Min-Hash signatures $SIG(*,x)$ and $SIG(*,y)$ agree on at least fraction s of their rows:
$|\{i \in [t] : SIG(i,x) = SIG(i,y)\}| \; / \; t \geq s$

Working Assumption: Docs x and y have the same ***similarity*** as their **Signatures** (thanks to **Corollary 1** of **Main THM**)

Note: Similarity of two column **signatures** does not come from *small* differences in some/several of their row values: there must be several/most **identical** values!

index gap $|i-j|$ has NO meaning !!

LSH: Step II (Hash Signatures to Buckets)

Main idea: Hash Columns of ***SIG*(*,*)** to a set **B** of Buckets for several times

Key Fact: (only) **candidate** Column sets are likely to hash to the same Bucket, *with high probability. (Key Property of Hash Functions)*

Hence:

Candidate pairs are those that hash to the same Bucket

Excercise: ***LSH*** overcomes the $\Theta(N^2)$ *barrier*: Why? When?

min-hash
DOCS with SAME INDEX will be hashed to the same BUCKET

Is this a 2nd level of HASHING in LSH

LSH. Step II (Details: Partition *SIG*(*,*) into Bands)

Technical Problem: Hashing the full signature of a Doc into a one Bucket may generate several false positive and false negative.

A Good Solution.

Partition each doc signature, i.e. each column of ***SIG*(*,*)** into a set of ***Bands*** (i.e. subsets of *adjacent* raws):

Divide matrix ***SIG*(*,*)** into ***b*** Bands, each of ***r*** rows

For each **Band**, hash its raw-portion of each column to a hash table with ***c*** Buckets

LSH

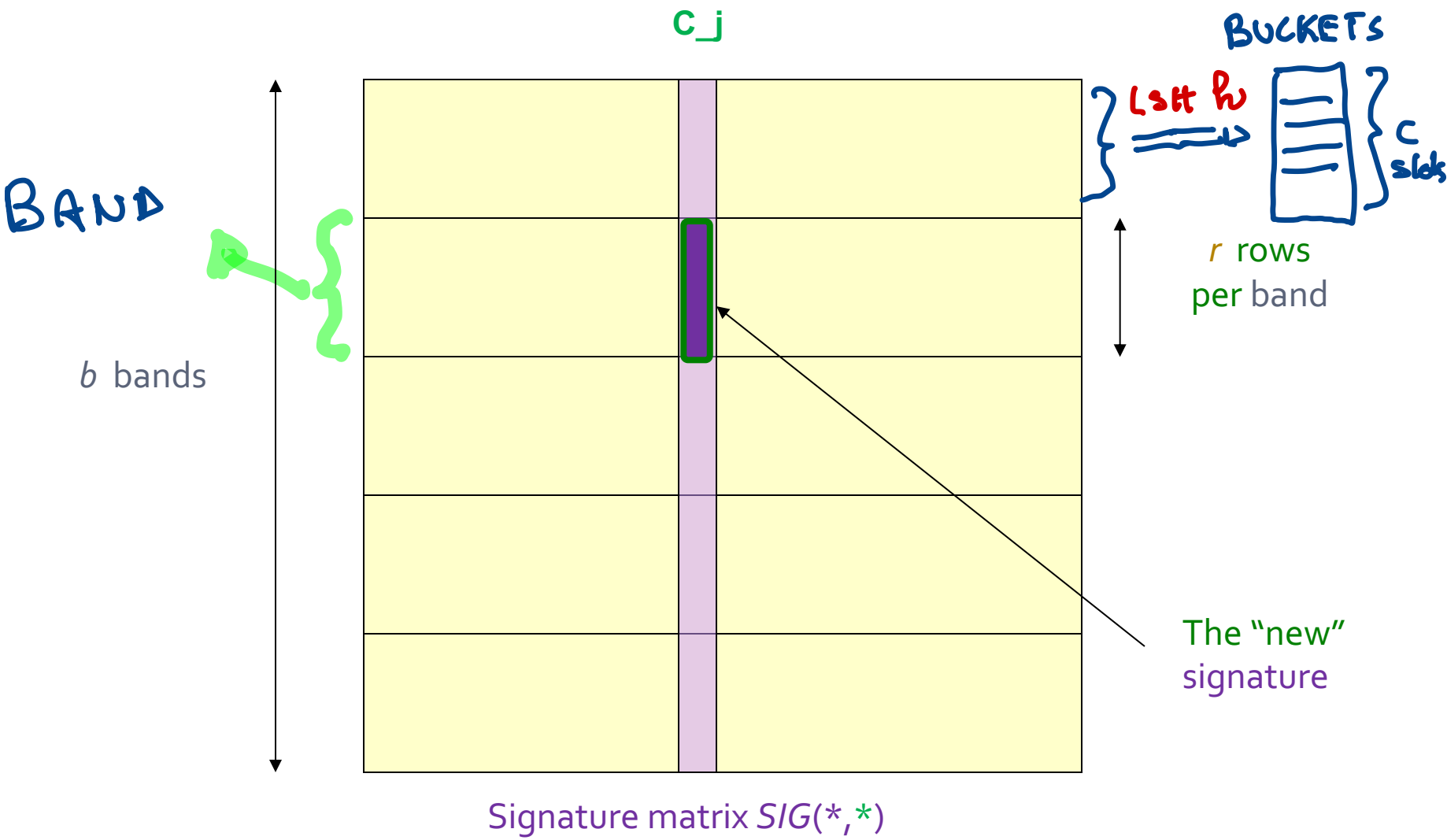
- Make ***c*** large enough (*w.r.t.* ***r***) to *whp* avoid **collisions** of different signatures

Candidate Criterium: Let's **candidate** column *pairs* those that hash to the same Bucket for **at least one Band**

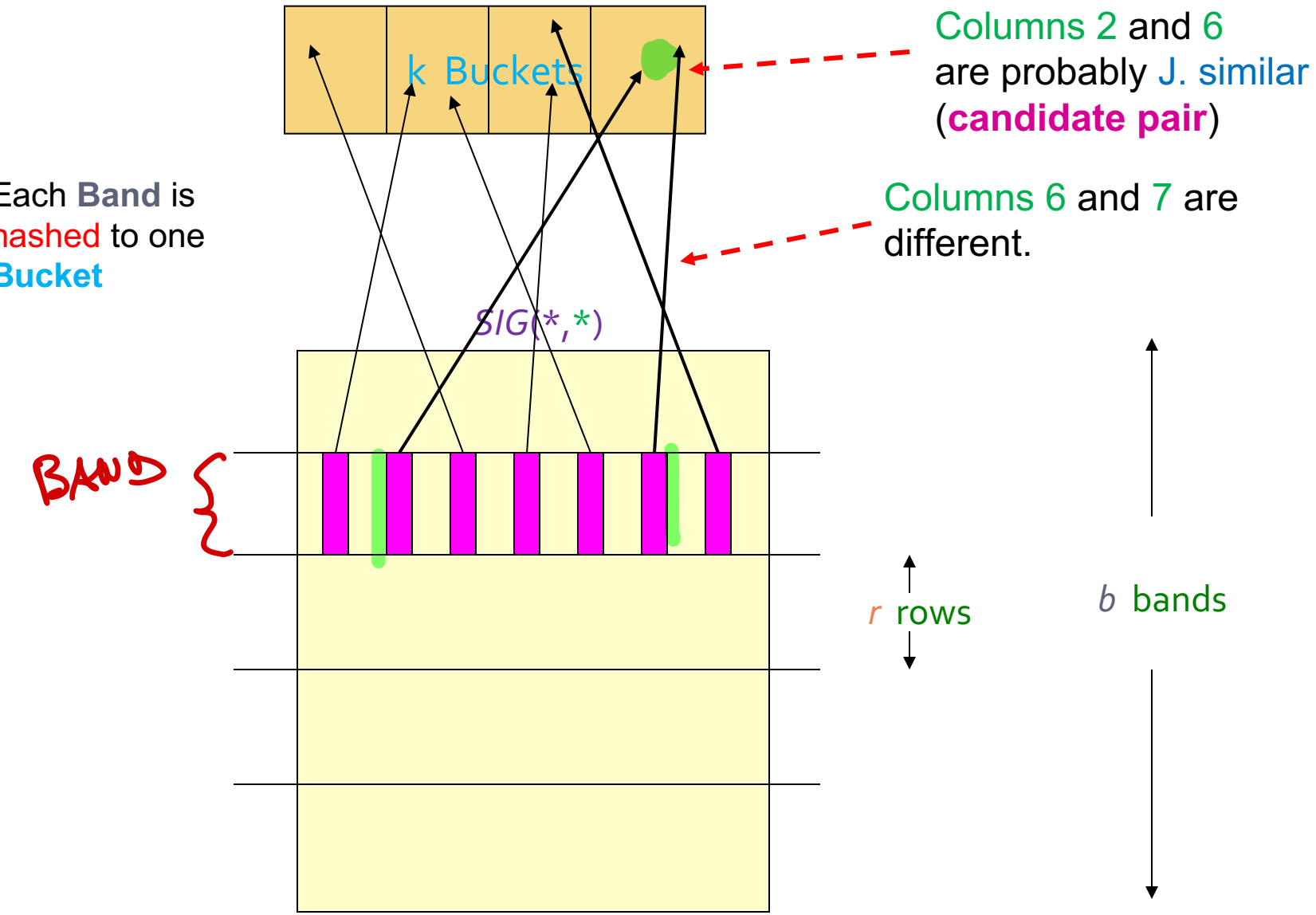
Practical Step: Tune ***b, r (and thus t= total n. of raws)*** to catch most similar *pairs*, but few non-similar *pairs*

Partition *SIG*(*,*) into *b* Bands

2	1	4	1
1	2	1	2
2	1	2	1



Hashing Bands



LSH: Simplifying Assumptions

There are **enough Buckets** (**c** >> **r**) that Columns are unlikely to hash to the same Bucket *unless* they are **identical** in a particular **Band**

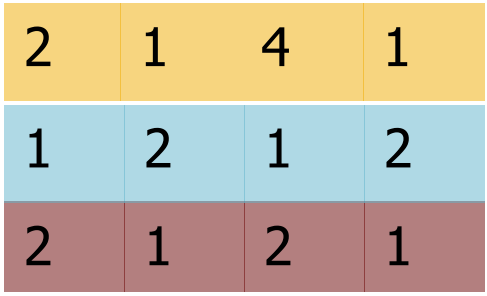
We assume that: "**same Bucket**" means "**identical in that Band**" (Perfect Hashing)

(real app = universal hashing)

Assumptions above needed only to simplify analysis, not for correctness of algorithm

Recall: **candidate** column pairs = those that hash to the same Bucket for **at least one Band**

Example of Bands



Assume the following case:

Suppose 100,000 columns of ***SIG*(*,*)** (100k Docs)

Signatures of 100 integers (rows)

Therefore, signatures take 40 Mb

Choose ***b*** = 20 bands, each of ***r*** = 5 integers.

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

Case I: C_1, C_2 are 80% similar

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

Assume: J.sim(C_1, C_2) = 0.8

- Since sim(C_1, C_2) $\geq$ **s**, we want C_1, C_2 to be a **candidate pair**: We want them to hash to **at least one common Bucket** (i.e. **at least one Band** is identical)

Fact 1. Probability C_1, C_2 get identical in one particular **Band is**

(i) $(0.8)^5 = 0.328$ {each of the 5 row pair must match}

Fact 2. Probability C_1, C_2 get ***not*** identical in *all* of the 20 **Bands** is

(ii) $(1-0.328)^{20} = 0.00035$ {each of the 20 Bands does not satisfy **(i)**} (false negative)

Confidence Rate: about 1/3000th of the 80%-**similar** column pairs are **false negatives** (we miss them) → **We would find 99.965% pairs of truly similar Docs**

Case II: C_1, C_2 are 30% similar

(ii)

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

Assume: J.sim(C_1, C_2) = 0.3

- Since J.sim(C_1, C_2) << **s** we want C_1, C_2 to hash to **NO common** Buckets (*all* **Bands** should be different)

Fact 1. Probability C_1, C_2 get identical in one particular **Band** is $(0.3)^5 = 0.00243$ r = 5

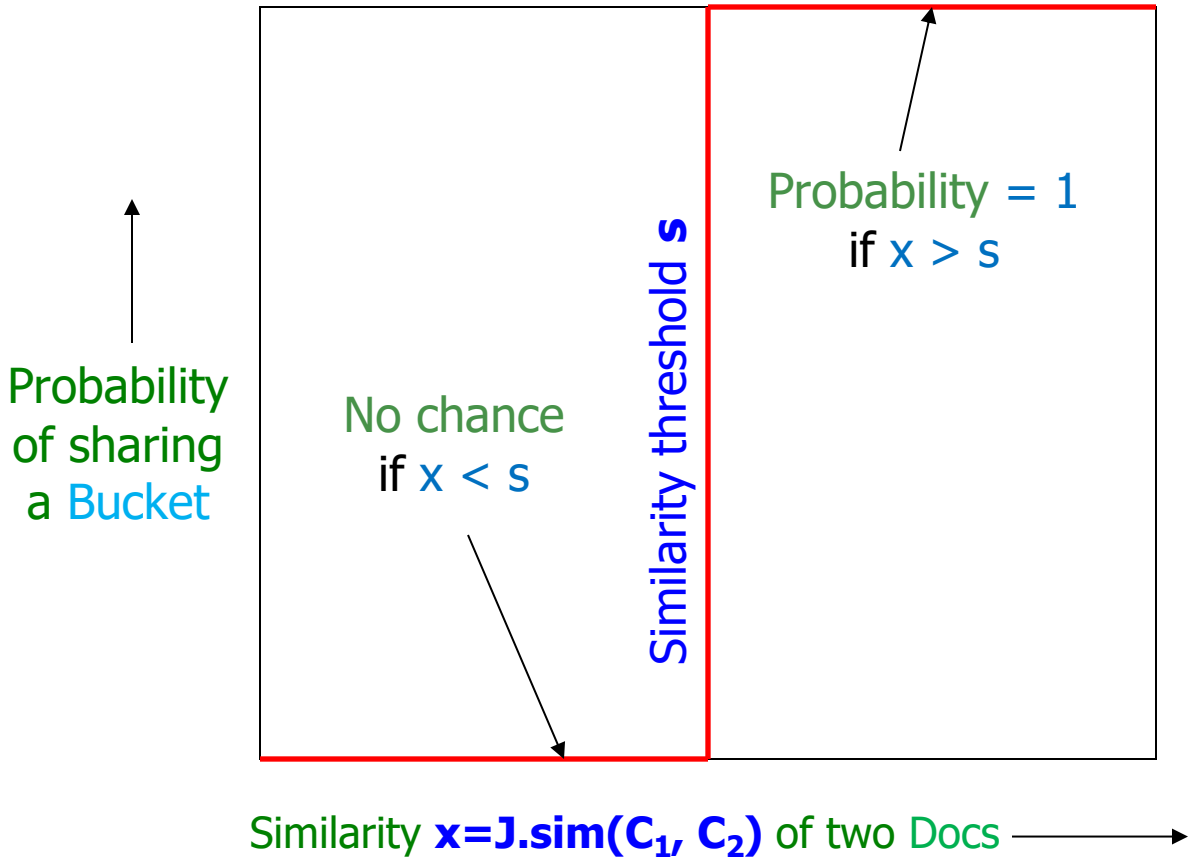
Fact 2. Probability C_1, C_2 get ***not***-identical in *all* of the 20 **Bands** is $1 - (1 - 0.00243)^{20} = 0.0474$ (false positive) → BAD EVENT

Confidence Rate: about 4.74% pairs of Docs with J.sim 0.3% end up becoming **candidate pairs**: They are **false positives.**

Since we will have to examine them (they are candidate pairs) but then it will turn out their similarity is below threshold **s**

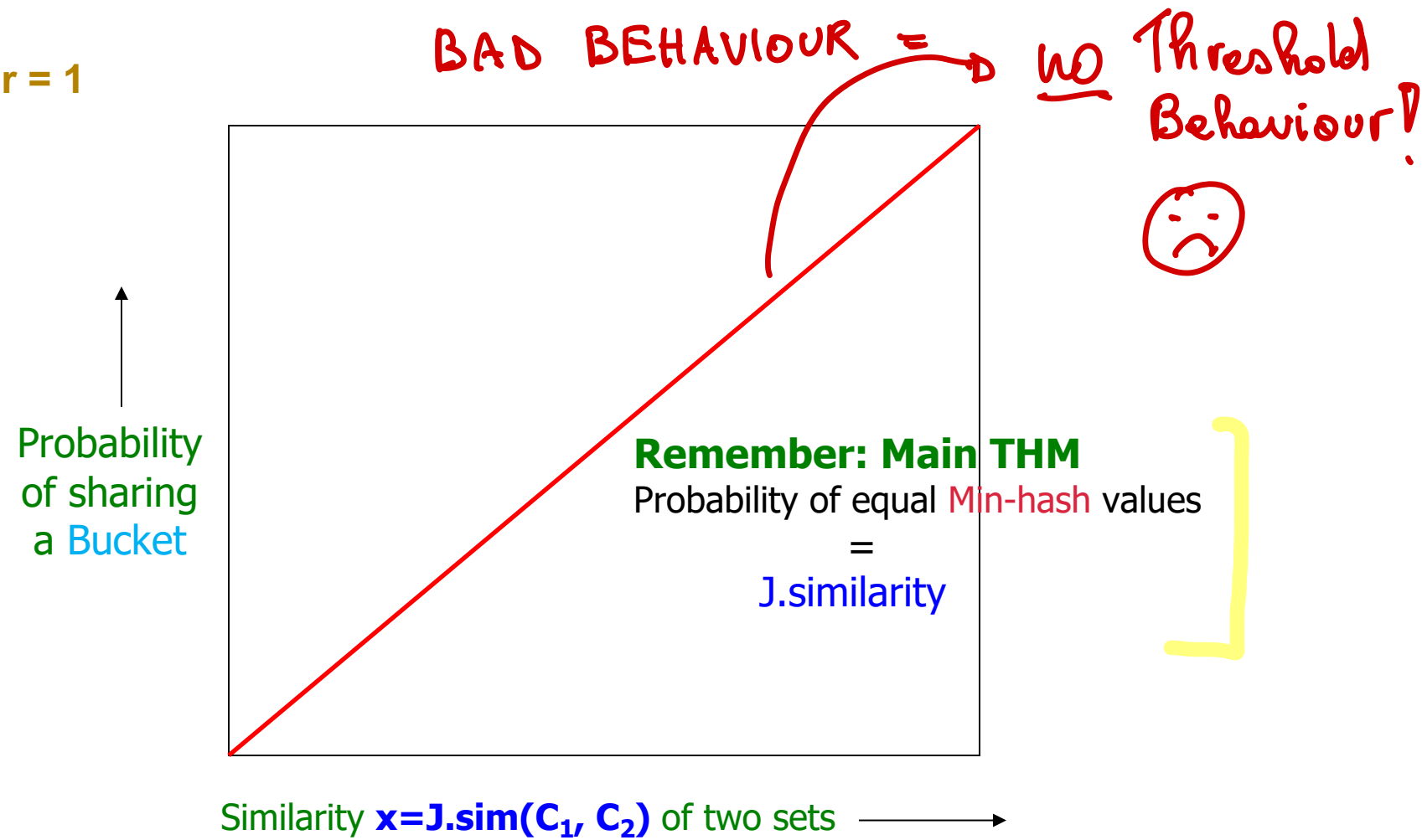
Analysis of LSH – Ideal Performance

x = J.sim(C_1, C_2); s = target similarity threshold



What 1 Band of 1 Row Gives You

Case **r = 1**



Using b Bands, *r* rows/band

suppose Docs C_1 and C_2 have J.similarity ***x.***

Pick any Band (with *r* rows):

$\Pr\{$ *All* rows in Band are *equal* $\} = x^r$

$\Pr\{\exists$ row in Band *unequal* $\} = 1 - x^r$

$\Pr\{\nexists$ Band identical $\} = (1 - x^r)^b$

$\Pr\{\exists$ Band identical $\} = 1 - (1 - x^r)^b$

Increases with **b**

Decreases with **r**

LSH Involves a Tradeoff

For a fixed threshold s, tune:

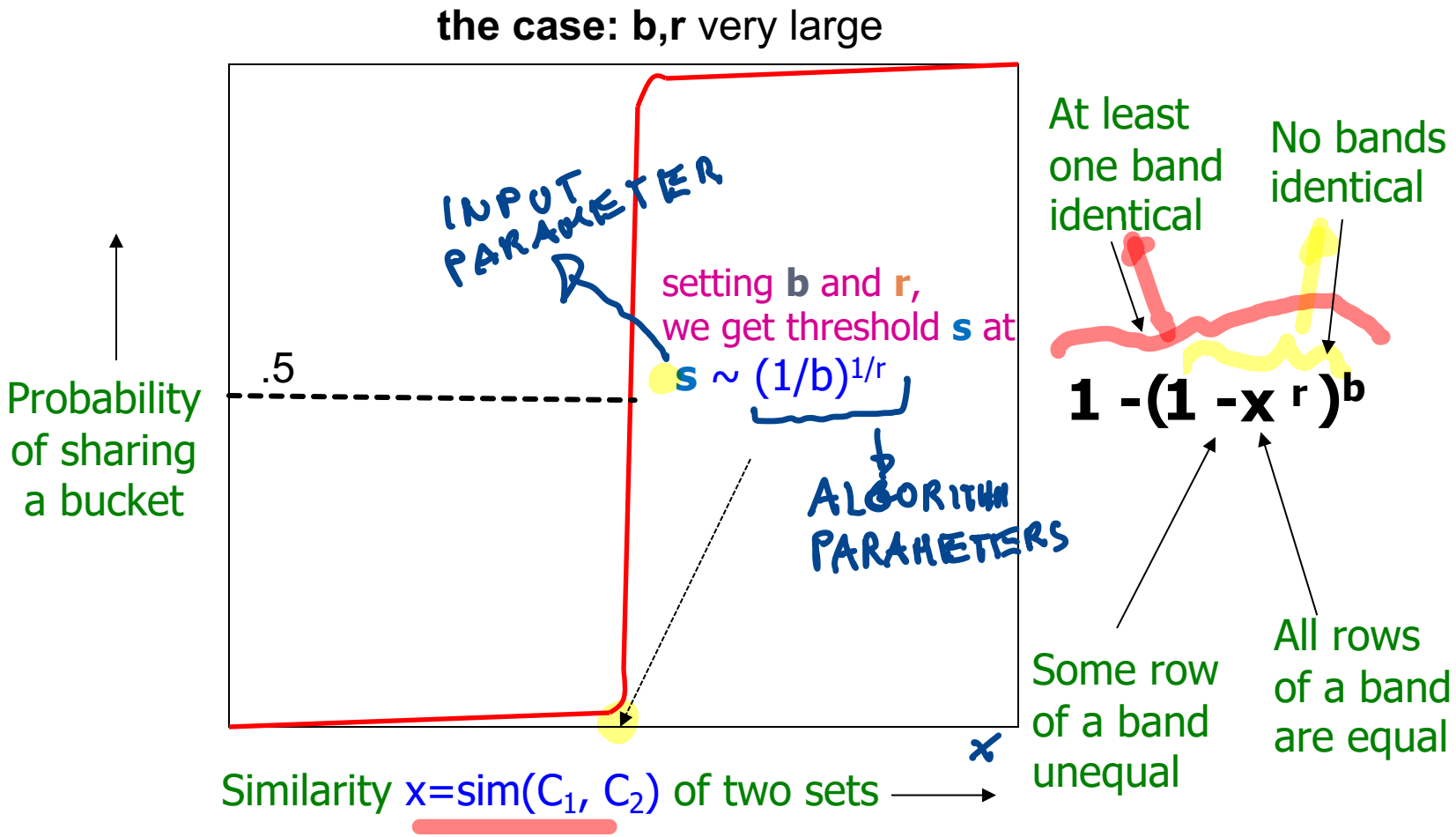
- The number **t** of Min-Hashes (rows of ***SIG*(*,*)**)
- The number of **Bands *b***, and, hence:
- The number of rows ***r*** per band

to balance false positives/negatives

(a) and (b)

Example: If we decreased to **15 Bands** of **5** rows, the number of false positives (Case II before) would *go down*, but the number of false negatives (Case I before) would *go up*

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



LSH: The Algorithmic Goal

Task: Given the **t** Signature columns for each Doc (i.e. Matrix ***SIG*(*,*)**) and the *sim threshold* **s** (with $0 < s \leq 1$) , **find** all the Doc pairs having: Signature Sim ≥ **s**

***LSH* Solution**: Apply the **LSH** **Band** technique to ***SIG*(*,*)**, by setting parameters **b** and **r** such that: $b \cdot r \leq t$ and $(1/b)^{1/r} \approx s$

Output: *All* (candidate) pairs of Docs having the same signature in *all* the **r** raws of *at least one* **Band**

Possible Parameter Tuning:

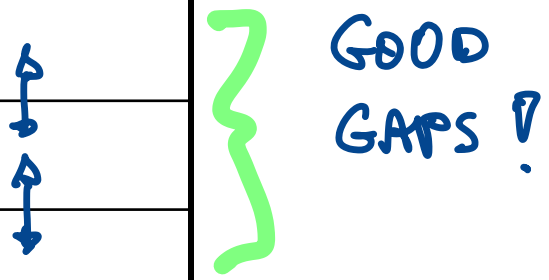
- increasing **b** → decreases the number of false negatives
- increasing **r** → decreases the number of false positives

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

J.Sim(C_1,C_2) = x ; b = 20; r = 5

Prob. that at least 1 Band is identical:

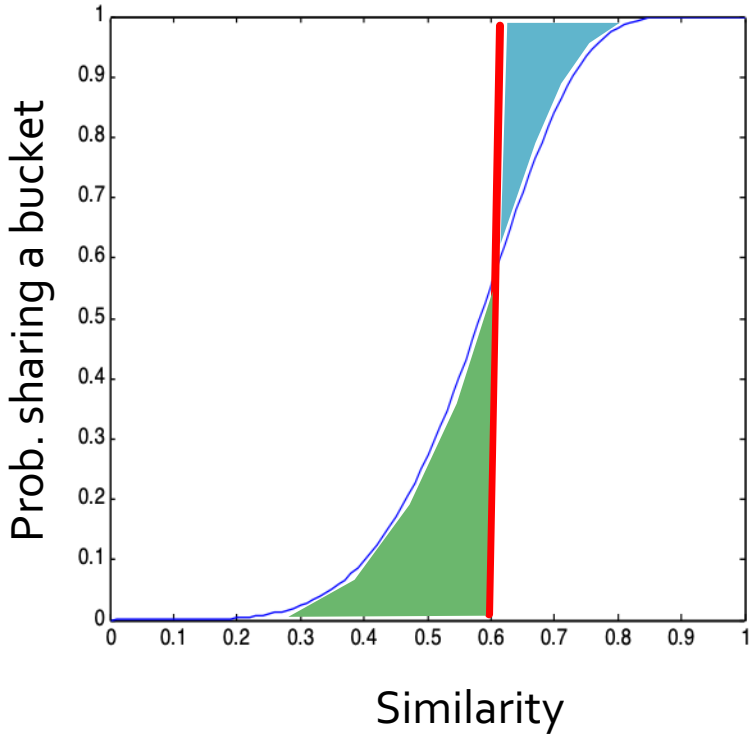
x	$1-(1-x^r)^b$
.2	.006
.3	.047
.4	.186
.5	.470
.6	.802
.7	.975
.8	.9996



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

Picking *r* and *b* to get the *best S-curve*

- **t= 50** hash-functions (**r**=5, **b**=10)



Blue area: False Negative rate
Green area: False Positive rate

LSH Summary

Tune ***b*** ***and*** ***r*** to get almost all pairs with similar signatures, but eliminate most pairs that do not have similar signatures: *most-balanced* choice is: **s** $\sim (1/b)^{1/r}$

Check in main memory that **candidate pairs** really do have **similar signatures** (i.e. same values in several raws)

Optional: In another pass through data, check that the selected candidate pairs really represent similar Docs

↳ avoid false Pos

Summary: 3 Steps

Shingling: Convert Docs to **Sets**

- We used **hashing** to assign each **k-shingle** an ***ID (one raw of the Big Matrix M)***

- J. SIM. for SUBSETS

Min-Hashing: Convert large sets to short signatures, while preserving *similarity*

- We used **similarity-preserving hashing** to generate signatures with property $\mathbf{Pr[}h_\pi(C_1) = h_\pi(C_2)\mathbf{]} = sim(C_1, C_2)$
- We used **hashing** to get around generating random ***permutations***

Locality-Sensitive Hashing: Focus on *pairs* of signatures likely to be from *similar* docs

- We used **hashing** to find **candidate pairs** of similarity $\geq$ **s**

[the BAND Technique]

Summary: 3 Steps

Shingling: Convert Docs to **Sets**

- We used **hashing** to assign each **k-shingle** an ***ID (one raw of the Big Matrix M)***

Min-Hashing: Convert large sets to short signatures, while preserving *similarity*

- We used **similarity-preserving hashing** to generate signatures with property $\mathbf{Pr[}h_\pi(C_1) = h_\pi(C_2)\mathbf{]} = sim(C_1, C_2)$
- We used **hashing** to get around generating random ***permutations***

Locality-Sensitive Hashing: Focus on *pairs* of signatures likely to be from *similar* docs

- We used **hashing** to find **candidate pairs** of similarity $\geq$ **s**