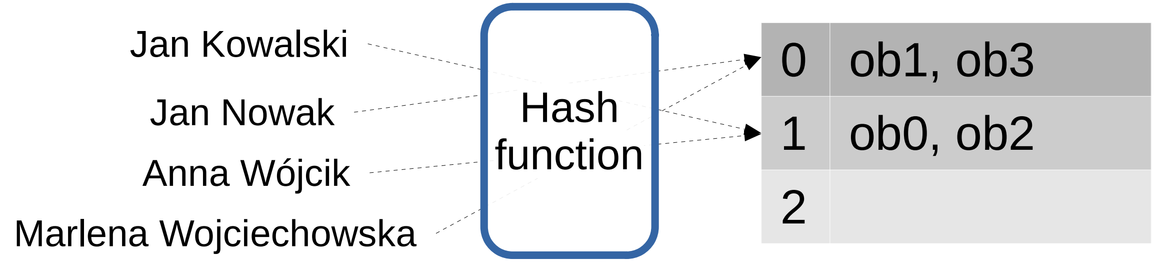
PDD

Lecture 5: Locality Sensitive Hashing

Prepared by Jacek Sroka

(based on Mining massive datasets)

Hashing



- Goal:
  - convert variable-length or composite keys into fixed length
  - What we want: deterministic, efficient to compute, uniformly distributes keys
- `hashCode` in Java
  - consistent with equals
  - not required that unequal objects produce different hash values, but we want that to improve performance of hash tables
- Examples
  - `hashCode := 7`
  - `hashCode := memory address`
  - `hashCode(int x) := 13x+7 (mod 10)`
- Cryptographic hashing
  - infeasible to reverse
  - infeasible to find different messages with the same hash value
  - a small change to a message should change the hash value (new hash value should look uncorrelated with the old one)

Hashing vs Locality Sensitive Hashing

- General hashing
  - Huge number of buckets (possibly more than items)
  - Small difference between items results in assigning to different bucket
  - Some degree of conflicts
- Locality Sensitive Hashing
  - Much less buckets than items
  - Similar items are assigned to the same bucket
  - Some degree of false positives and false negatives (can be controlled)

Applications

- Clustering
- Finding similar item sets
  - Plagiarism in the net
  - Grouping similar news articles
    - National news agency sells stories to publishers
  - Finding page mirrors (don’t want to show both in search results)
    - Different adds
    - Different ordering of content
  - Collaborative filtering
    - Users give “likes” to content
      - Similar users like similar sets of items
      - Similar items are liked by similar sets of users
    - Problem: find similar users/items
  - Entity resolution
    - Differences in phone numbers or addresses

Similarity of text documents

- There exist other approaches like TF/IDF, but we don’t want to check all pairs
- **Jaccard similarity** for sets

The plan

- **Shingling**: documents → sets
- **Minhashing**: large sets → short signatures (preserving similarity)
- **Locality sensitive hashing**: find small subset of all pairs that can be similar

Shingling

- K-shingles (k-grams) – sequence of `k` characters from the document
- `k=2`, `doc=trelemorele`
  - `2-shingle={tr,re,el,le,em,mo,or}`

- We represent documents as sets of their k-shingles
  - Similar documents have many shingles in common
  - Changing a word affects k-shingles `k` away from the word
  - Reordering paragraphs only affects shingles that cross paragraph boundaries

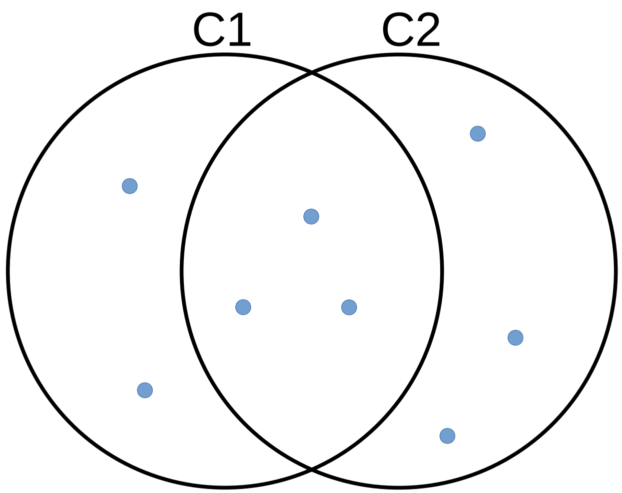
- `k=3`, `"ala ma kota"` vs `"ma kota ala"`
  - `"la ", "a m", " ma"` → `"ta ", "a a", " al”`

Shingle size

- Usually 4-10
- Longer shingles differentiate more
- Long shingles can be hashed to 4 byte tokens
  - Rare collisions

Jaccard similarity

- Sim(C1,C2)=|C1∩C2|/|C1**U**C2|



Sim(C1,C2)=3/8

Indicator metrices

- Rows – elements of the universal set
  - All k-shingles
- Columns = sets
- Sparse
- 1 in row **e** and column **S** <=> shingle **e** ϵ document **S**

Jaccard similarity for indicator matrix

- Jaccard similarity = similarity of rows with at least one 1 (proof by coloring :))

C1	C2	
1	1	==
0	1	!=
1	0	!=
0	0	
1	1	==
1	0	!=

`Sim(C1,C2) = 2/5`

- Types of rows:

C1	C2	
1	1	a
1	0	b
0	1	c
0	0	d

`Sim(C1, C2) = a/(a+b+c)`

Minhashing

- Permute rows randomly
- Minhash function `h(C)` = number of the first (in the permuted order) row with 1
- Use many (e.g. 100) independent permutations (hash functions) to create signature
- Indicator matrix → signature matrix

1	4	3	1	0	1	0
3	2	4	1	0	0	1
7	1	7	0	1	0	1
6	3	6	0	1	0	1
2	6	1	0	1	0	1
5	7	2	1	0	1	0
4	5	5	1	0	1	0

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

Surprising property

- Probability over all permutation that `h(C1)=h(C2)` is `Sim(C1,C2)`
- Both are `a/(a+b+c)`
  - Find first row with at least one `1`
  - If type-**a** then `h(C1)=h(C2)`
  - If type-**b** or type-**c** then `h(C1)!=h(C2)`
  - Random permutation picks a row as first with probability `a/(a+b+c)`
- Similarity of signatures = fraction of the minhash functions in which they agree
  - Fraction of rows in which columns of integers agree
- Expected similarity of signatures = Jaccard similarity of columns
  - The longer signatures, the smaller the error

Example

1	4	3	1	0	1	0
3	2	4	1	0	0	1
7	1	7	0	1	0	1
6	3	6	0	1	0	1
2	6	1	0	1	0	1
5	7	2	1	0	1	0
4	5	5	1	0	1	0

3/4

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

2/3

Example

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

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

3/4

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

1

Example

1	4	3	1	0	1	0
3	2	4	1	0	0	1
7	1	7	0	1	0	1
6	3	6	0	1	0	1
2	6	1	0	1	0	1
5	7	2	1	0	1	0
4	5	5	1	0	1	0

0

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

0

Implementation

- Hard to pick permutation for large number of rows (e.g. 1 bilion)
  - Takes lots of memory
  - Accessing rows in permuted order leads to thrashing
- Good approximation is to use hash functions
  - Let $h_i(r)$ give the order of rows for i-th permutation

```
for each row r do:
   for each hash function hi do:
      compute hi(r)
   for each column c
      if c has 1 in row w
         for each hash function hi do
            if hi(r) < M(i,c) then
               M(i,c) := hi(r)
```

Example

```
for each row r do:
   for each hash function hi do:
      compute hi(r)
   for each column c
      if c has 1 in row w
         for each hash function hi do
            if hi(r) < M(i,c) then
               M(i,c) := hi(r)
```

f(x)= x mod 5
g(x)=(2x+1) mod 5

Row	C1	C2
1	1	0
2	0	1
3	1	1
4	1	0
5	0	1

	Sig1	Sig2
f(1)=1	1	∞
g(1)=3	3	∞
f(2)=2	1	2
g(2)=0	3	0
f(3)=3	1	2
g(3)=2	2	0
f(4)=4	1	2
g(4)=4	2	0
f(5)=0	1	0
g(5)=1	2	0

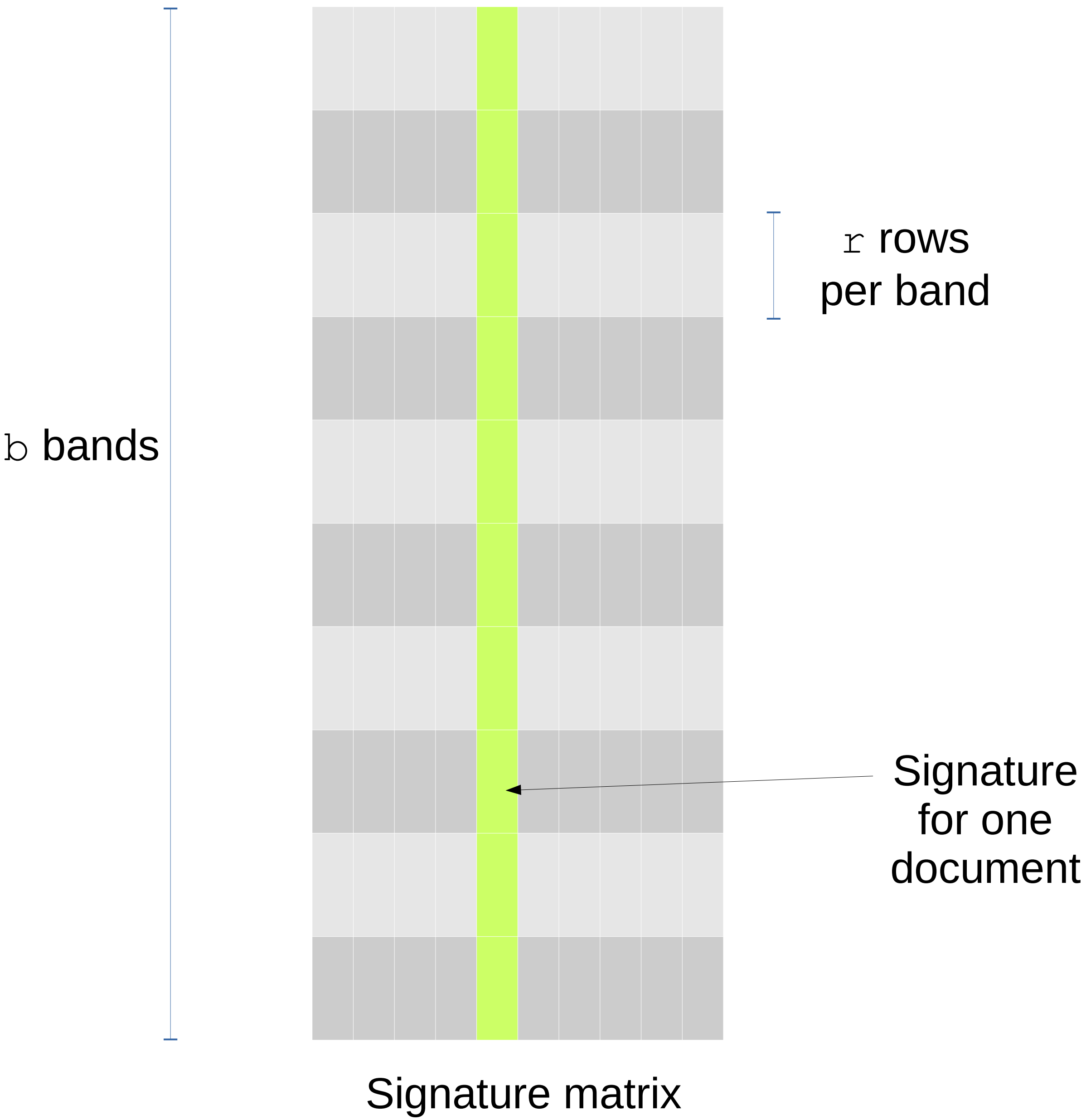
Implementation cont.

- If data is given by column not by row
  - Columns=documents, rows=shingles
  - Sort matrix once so it is by row

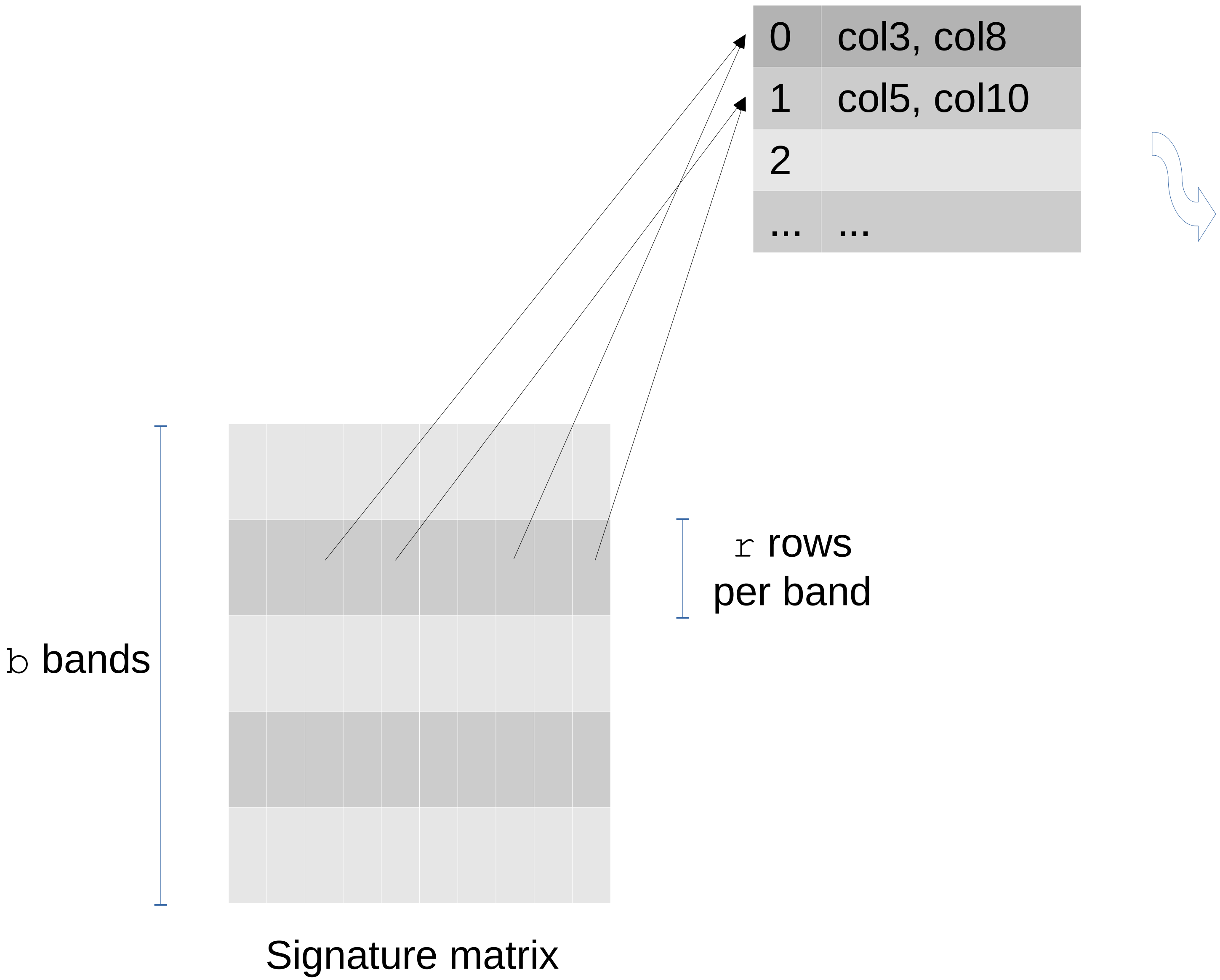
Locality sensitive hashing

- **Goal**: produce small list of **candidate pairs** – elements that must be evaluated
- Use LSH to hash into many buckets. Elements in the same bucket are candidate pairs.
- Pick a similarity threshold `t (< 1)`
  - Pair of columns is candidate if they agree in at least fraction of `t` rows, i.e., `M(i,c)=M(i,d)` for at least fraction `t` values of `i`
- Big idea: hash columns of `M` several times
  - Candidate pairs = hash to at least one same bucket

Partition into bands



Hash function for one bucket



`col3` and `col5` different for sure

`col5` and `col10` probably identical

Examples

- `100,000` columns (`5,000,000,000` pairs of signatures)
- signatures of `100` integers (`40Mb`)
- Goal: find all `80%` similar pairs of documents

<u>20 bands of 5 integers/band</u>

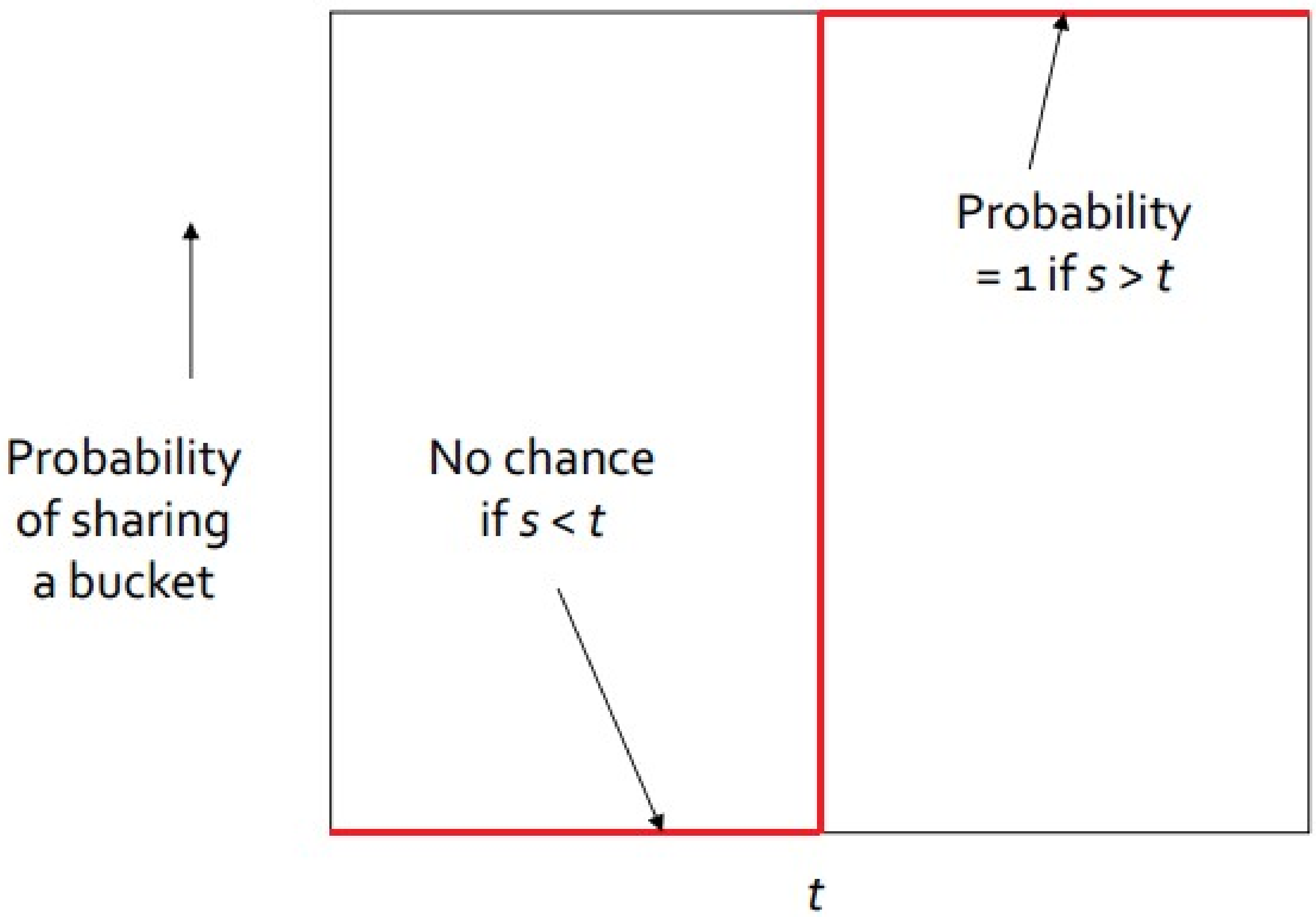
`C1,C2` are `80%` similar

- Probability `C1,C2` identical in one band: `(0.8)^5 =0.328`
- Probability `C1,C2` **not** similar in any of 20 bands:
  `(1-0.328)^20=.00035`
  - 1/3000th of 80%-similar pairs are false negatives

`C1,C2` are `40%` similar

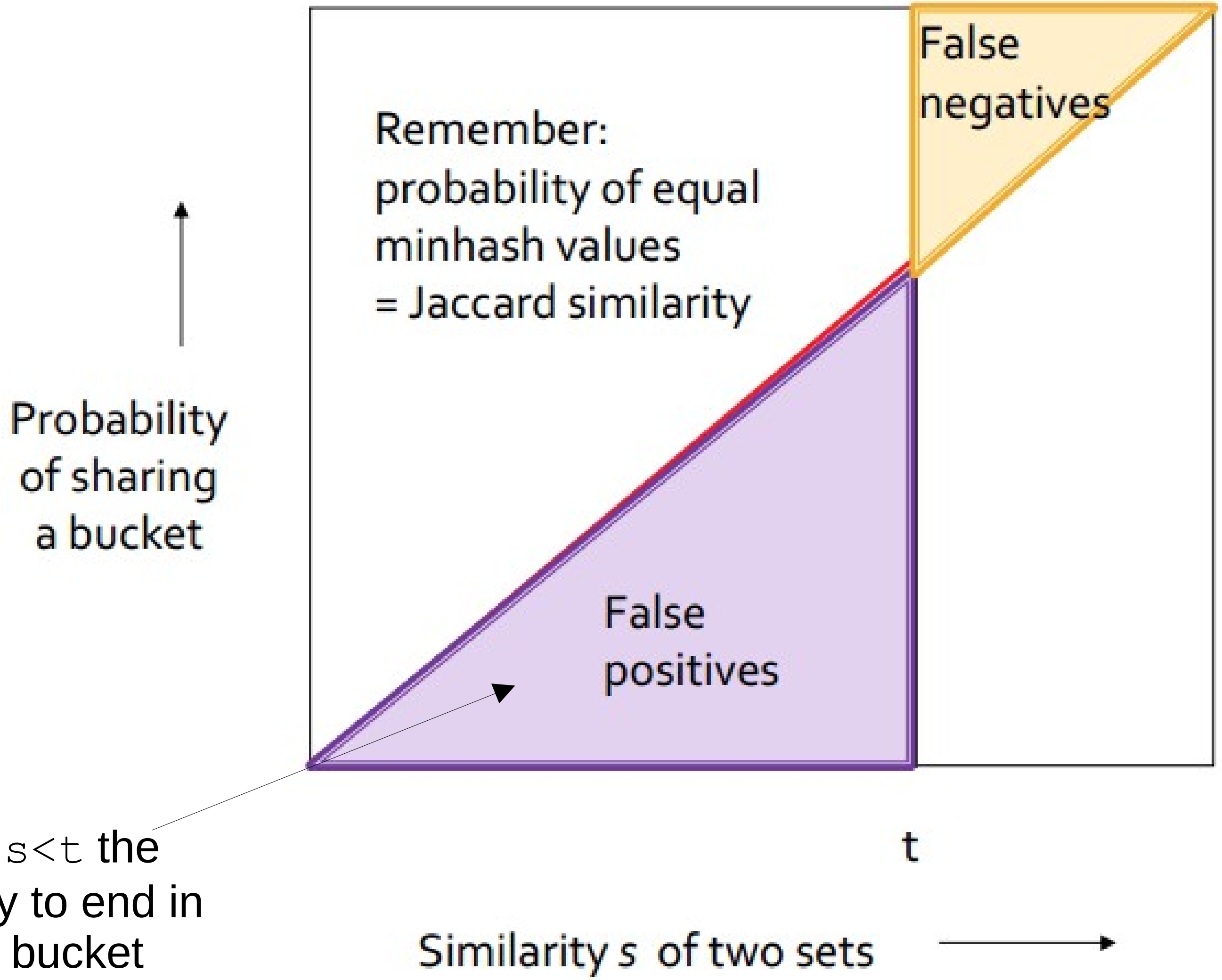
- Probability `C1,C2` identical in one band: `(0.4)^5 =0.01`
- Probability `C1,C2` identical in >= 1 of 20 bands: `<= 20*0.01=0.2`
  - Small chance for false positives
  - Decreases quickly for lower similarities

What we aim for



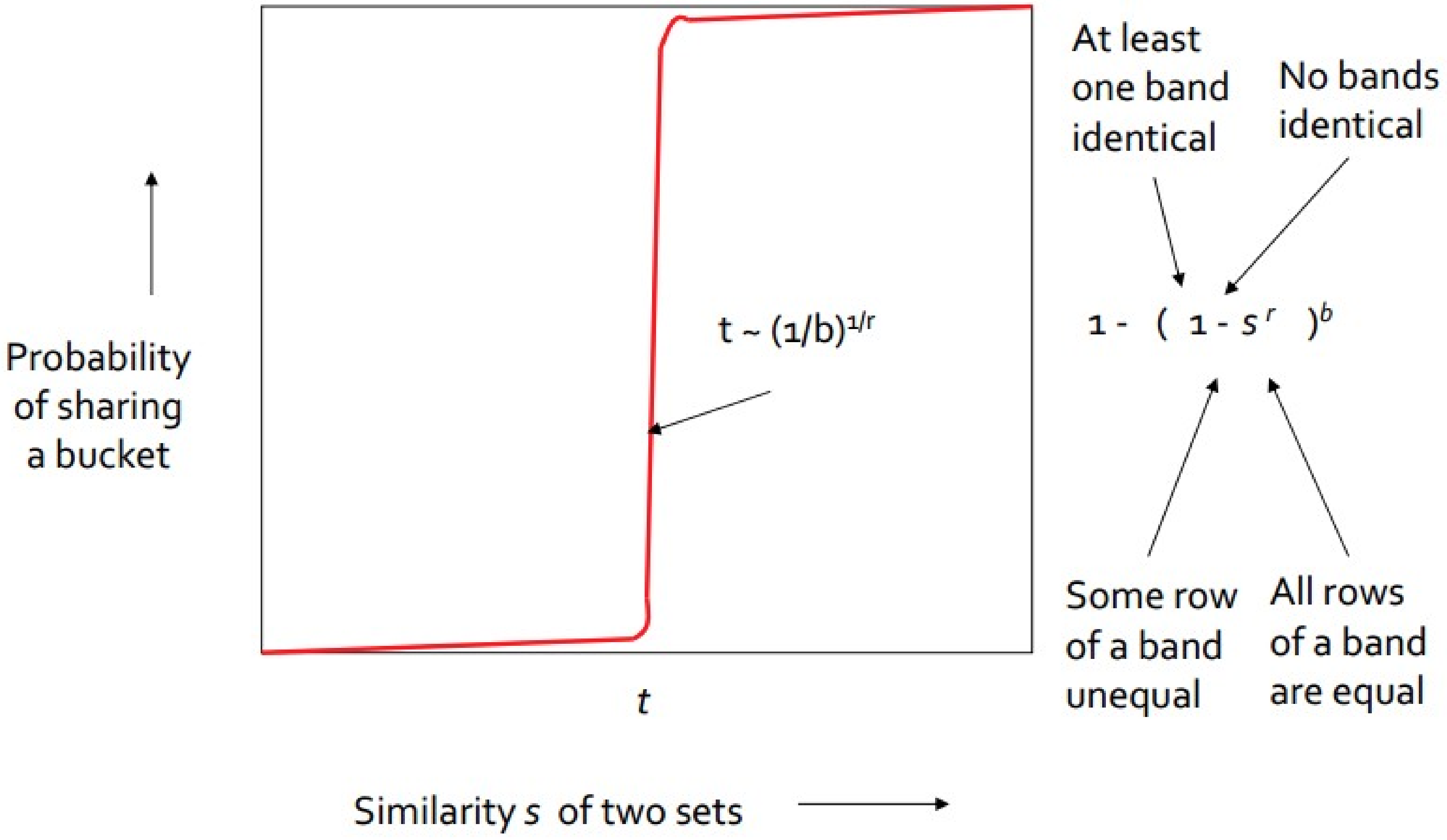
Similarity s of two sets ->

What we get with one band



For each `s<t` the probability to end in the same bucket resulting in false possitive is `s`

What we get with b bands



Example: `b=20; r=5`

s	$1-(1-s^r)^b$
0.2	0.006
0.3	0.047
0.4	0.186
0.5	0.470
0.6	0.802
0.7	0.975
0.8	0.9996

Summary

- Tune to get almost all pairs with similar signatures, but eliminate most pairs that do not have similar signatures
- Check that candidate pairs really do have similar signatures
- Optional: In another pass through data, check that the remaining candidate pairs really represent similar sets

LSH for Euclidean distance

- Hash function correspond to lines
- Partition line into buckets of size a
- Hash its point into a bucket containing its projection onto the line
- Nearby points always close; distant rarely
- See https://www.youtube.com/watch?v=arjbdAEf9c0