IERG4300
Web-Scale Information Analytics

Finding Similar Items and Locality Sensitive Hash (LSH)

Prof. Wing C. Lau
Department of Information Engineering
wclau@ie.cuhk.edu.hk

Acknowledgements

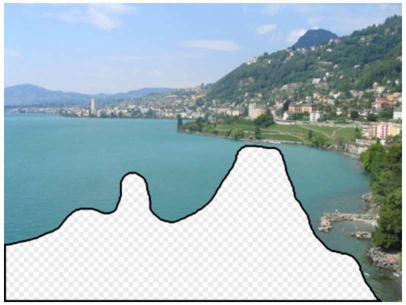
- Many slides used in this chapter are adapted from:
  - CS246 Mining Massive Data-sets, by Jure Leskovec, Stanford University.
  - COMS 6998-12 Dealing with Massive Data, by Sergei Vassilvitskii, (Yahoo! Research), Columbia University

[Hays and Efros, SIGGRAPH 2007]

Scene Completion Problem

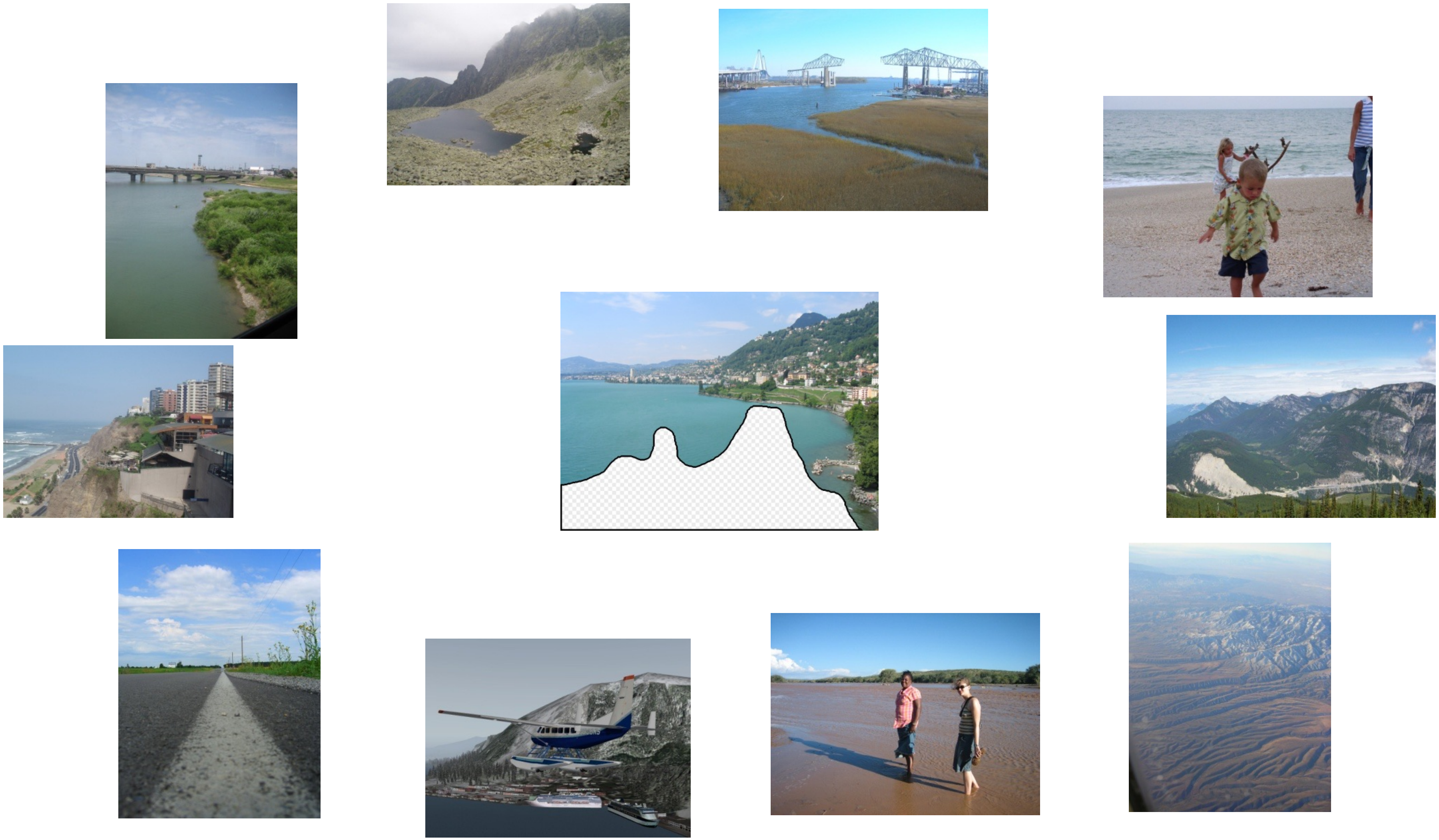
[Hays and Efros, SIGGRAPH 2007]

Scene Completion Problem



[Hays and Efros, SIGGRAPH 2007]

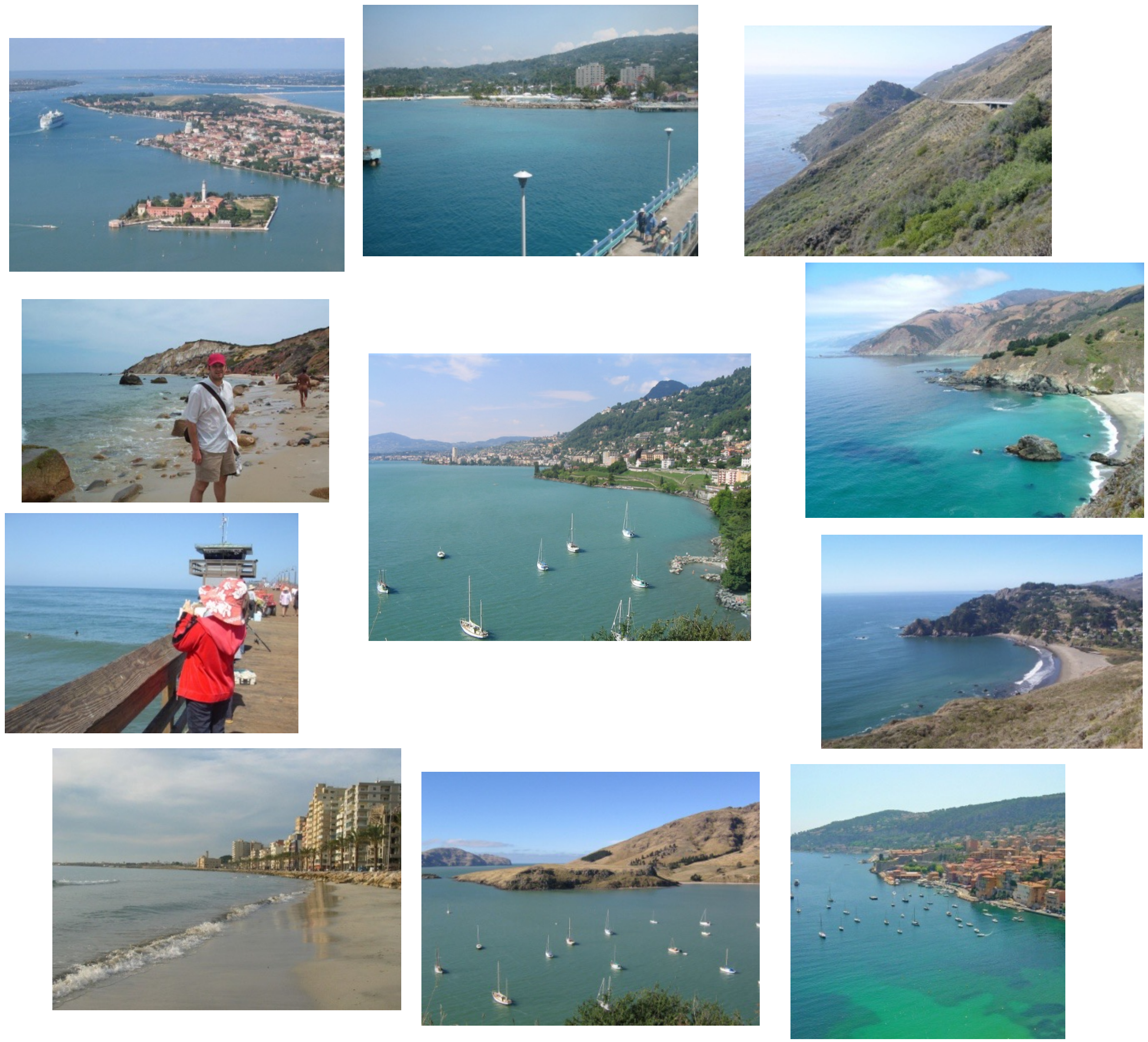
Scene Completion Problem



10 nearest neighbors from a collection of 20,000 images

[Hays and Efros, SIGGRAPH 2007]

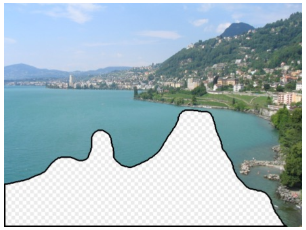
Scene Completion Problem



10 nearest neighbors from a collection of 2 million images

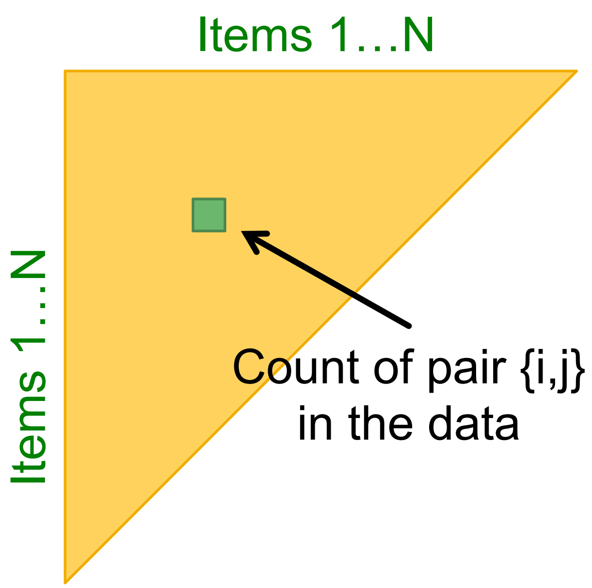
A Common Metaphor

- **Many problems can be expressed as finding "similar" sets:**
  - **Find near-neighbors in high-dimensional space**
- **Examples:**
  - **Web Pages with similar words**
    - For duplicate detection, classification by topic
  - **Customers who purchased similar products**
    - Products with similar customer sets
  - **Images with similar features**
  - Users who visited the similar websites



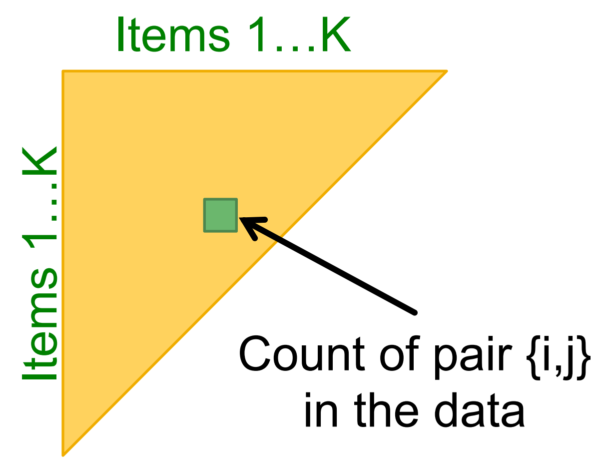
Relation to Previous Topic

- **Last time:** Finding frequent pairs



Naïve solution:
Single pass but requires space quadratic in the number of items

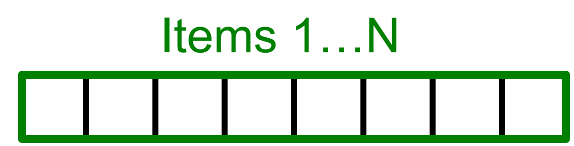
N … number of distinct items
K … number of items with support ≥ s



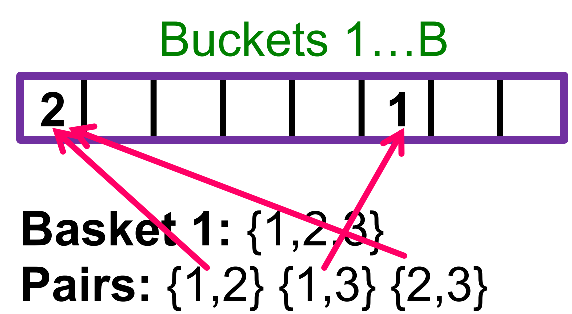
A-priori:
<u>First pass:</u> Find frequent singletons
For a pair to be **a candidate for a frequent pair**, its singletons have to be frequent!
<u>Second pass:</u>
Count only candidate pairs!

Relation to Previous Topic

- **Last time:** Finding frequent pairs
- **Further improvement: PCY**
  - **Pass 1:**
    - Count exact frequency of each item:

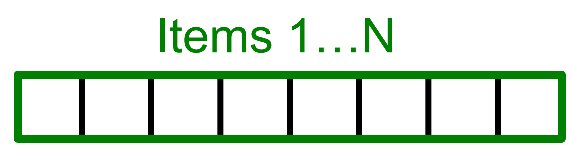


    - Take pairs of items {i,j}, hash them into *B* buckets and count of the number of pairs that hashed to each bucket:

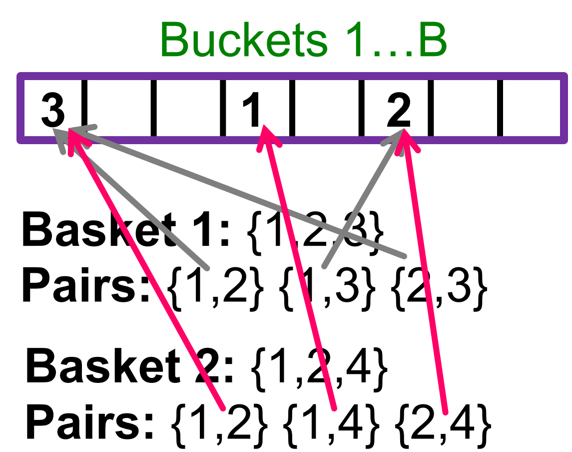


Relation to Previous Topic

- **Last time:** Finding frequent pairs
- **Further improvement: PCY**
  - **Pass 1:**
    - Count exact frequency of each item:



    - Take pairs of items {i,j}, hash them into *B* buckets and count of the number of pairs that hashed to each bucket:
  - **Pass 2:**
    - For a pair {i,j} to be a **candidate for a frequent pair,** its singletons have to be frequent and it has to hash to a frequent bucket!



Relation to Previous Lecture

- Last
- Fu

<u>Previous lecture: A-priori</u>

Main idea: Candidates

Instead of keeping a count of each pair, only keep a count for candidate pairs!

Today's lecture: Find pairs of similar docs

Main idea: Candidates

-- Pass 1: Take documents and hash them to buckets such that <u>documents that are similar hash to the same bucket</u>

-- Pass 2: Only compare documents that are **candidates** (i.e., they hashed to a same bucket)

Benefits: Instead of N^2 comparisons, we need O(N) comparisons to find similar documents

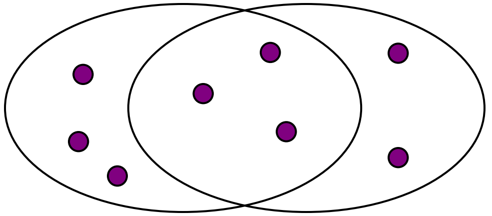
3}

{2,4}

Finding Similar Items

Distance Measures

- **Goal: Find near-neighbors in high-dim. space**
  - We formally define "near neighbors" as points that are a "small distance" apart
- For each application, we first need to define what "**distance**" means
- **Today: Jaccard distance (/similarity)**
  - The *Jaccard Similarity/Distance* of two sets is the size of their intersection / the size of their union:
  - $sim(C_1, C_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$
  - $d(C_1, C_2) = 1 - |C_1 \cap C_2| / |C_1 \cup C_2|$



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

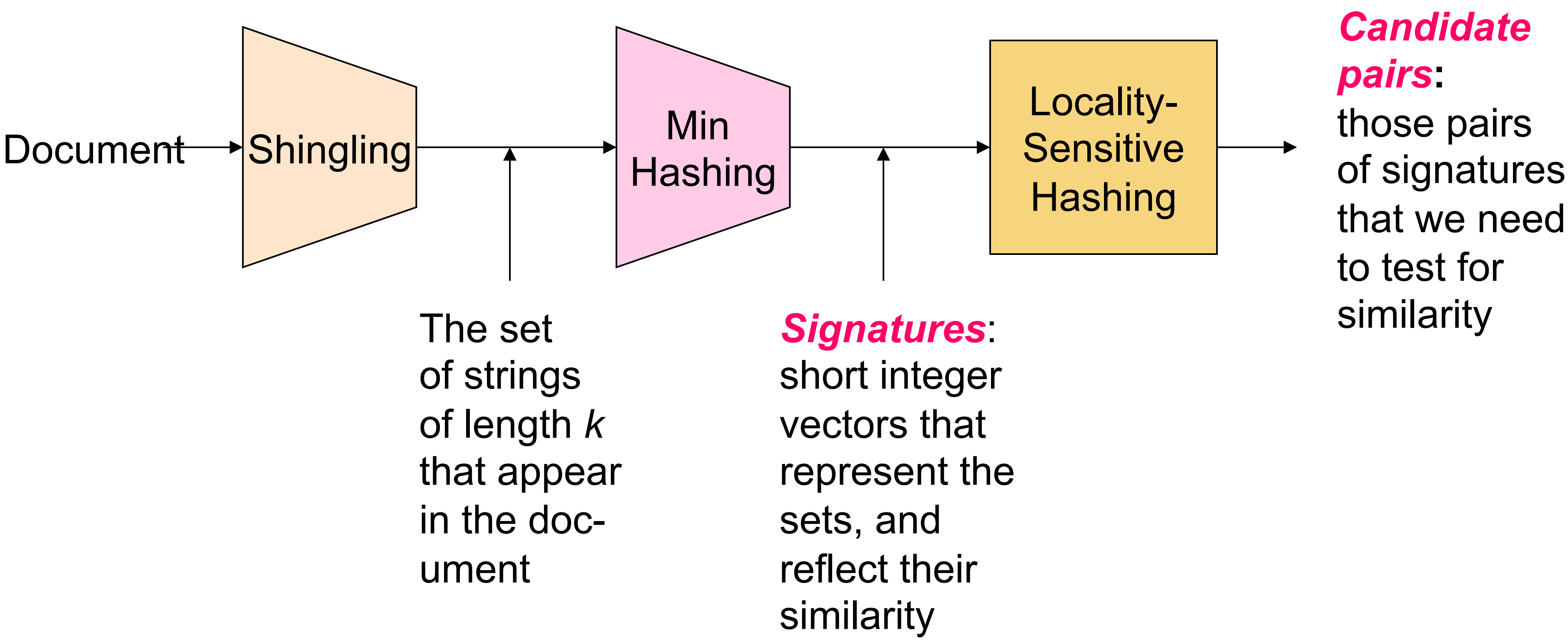
Finding Similar Documents

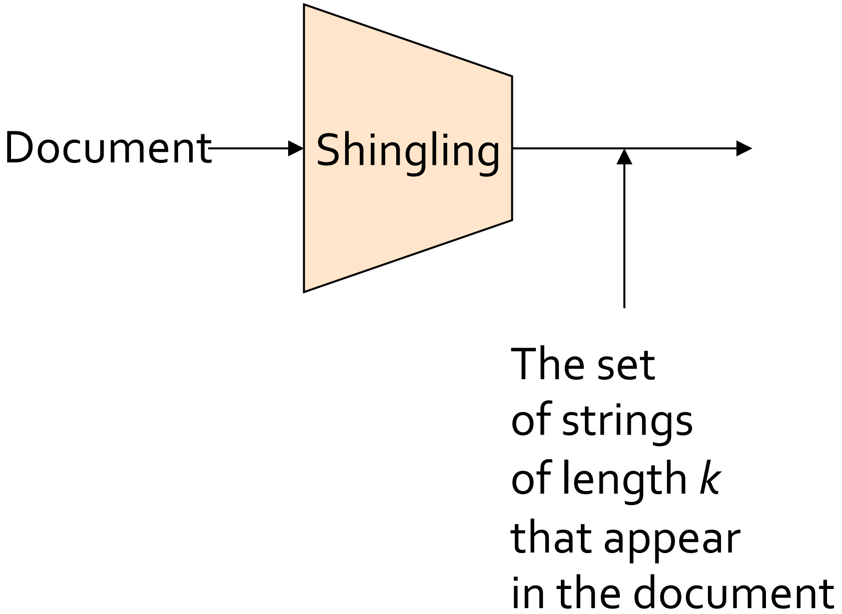
- **Goal:** Given a large number (*N* in the millions or billions) of text documents, find pairs that are "near duplicates"
- **Applications:**
  - Mirror websites, or approximate mirrors
    - Don't want to show both in a search
  - Similar news articles at many news sites
    - Cluster articles by "same story"
- **Problems:**
  - Many small pieces of one document can appear out of order in another
  - Too many documents to compare all pairs
  - Documents are so large or so many that they cannot fit in main memory

3 Essential Steps for Similar Docs

1. ***Shingling:*** Convert documents to sets
2. ***Minhashing:*** Convert large sets to short signatures, while preserving similarity
3. ***Locality-sensitive hashing:*** Focus on pairs of signatures likely to be from similar documents
   - **Candidate pairs!**

The Big Picture





Shingling

Shingling:

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 way: **Shingles (aka n-grams)!**

Define: Shingles

- A *k*-shingle (or *k*-gram) for a document is a sequence of *k* tokens that appears in the doc
  - Tokens can be characters, words or something else, depending on the application
  - Assume tokens = characters for examples

- **Example:** k=2; document D_1= abcab
  Set of 2-shingles: $S(D_1)$={ab, bc, ca}
  - **Option**: Shingles as a bag (multiset), count ab twice: $S'(D_1)$={ab, bc, ca, ab}

Compressing Shingles

- To **compress long shingles**, we can **hash** them to (say) 4 bytes
- **Represent a doc by the set of hash values of its *k*-shingles**
  - **Idea:** Two documents could (rarely) appear to have shingles in common, when in fact only the hash-values were shared
- **Example:** k=2; document D_1= abcab
  Set of 2-shingles: $S(D_1)$={ab, bc, ca}
  Hash the shingles: $h(D_1)$={1, 5, 7}

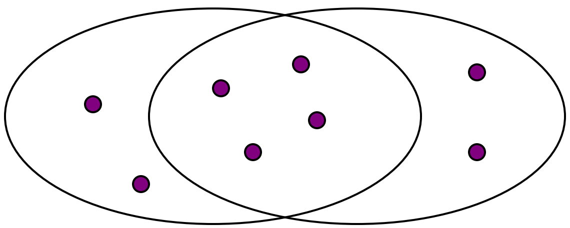
Working Assumption

- **Documents that have lots of shingles in common have similar text, even if the text appears in different order**
- **Caveat:** You must pick ***k*** large enough, or most documents will have most shingles
  - ***k*** = 5 is OK for short documents
  - ***k*** = 10 is better for long documents

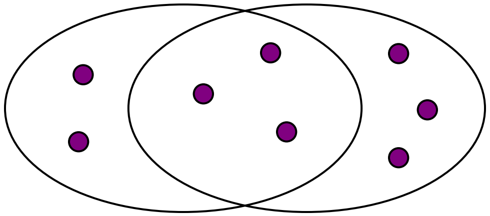
Similarity Metric for Shingles

- **Document D_1 = set of k-shingles C_1=S(D_1)**
- Equivalently, each document is a 0/1 vector in the space of *k*-shingles
  - Each unique shingle is a dimension
  - Vectors are very sparse
- A natural similarity measure is the **Jaccard similarity:**

$$Sim(D_1, D_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$$



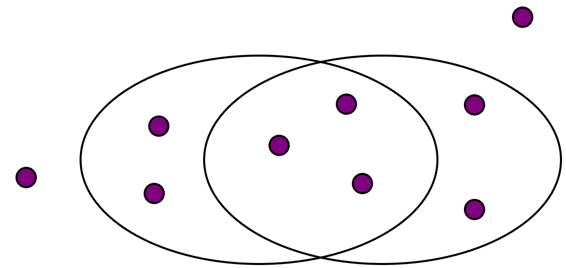
Encoding Sets as Bit Vectors



- Many similarity problems can be formalized as **finding subsets that have significant intersection**
- **Encode sets using 0/1 (bit, boolean) vectors**
  - One dimension per element in the universal set
- Interpret set intersection as bitwise **AND**, and 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
  - $d(C_1,C_2)$ = 1 – (Jaccard similarity) = 1/4

From Sets to Boolean Matrices

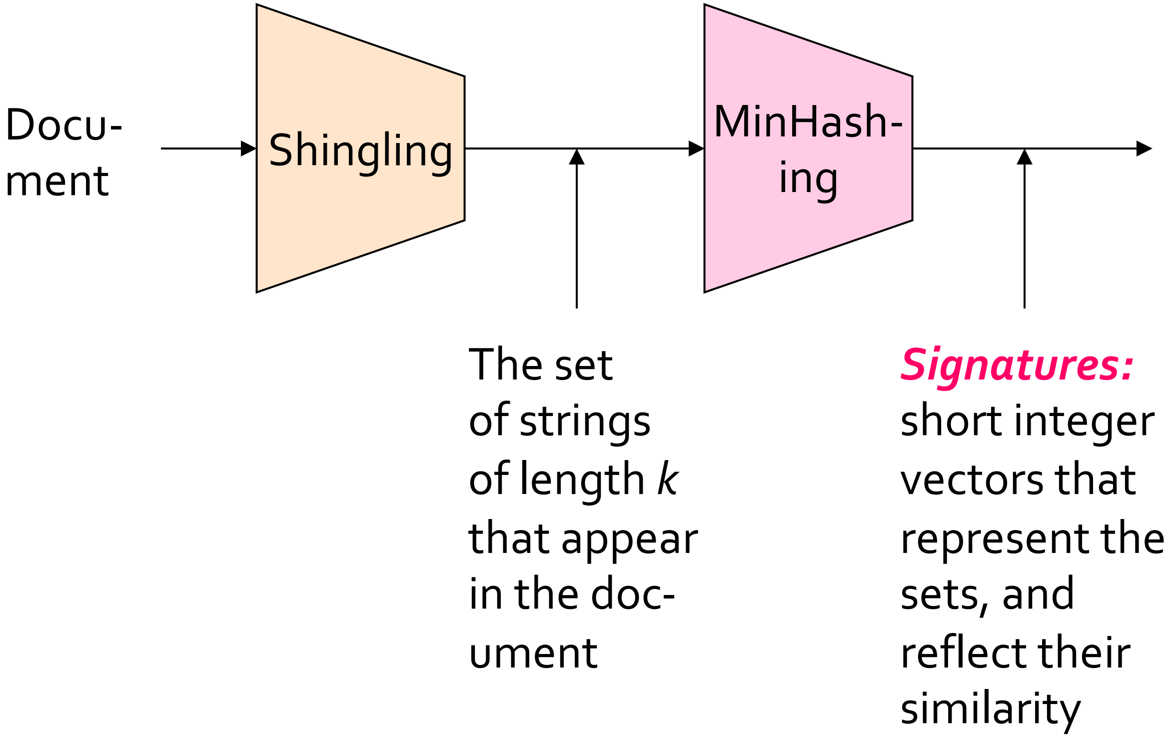


- Rows = elements (shingles)
- Columns = sets (documents)
  - 1 in row ***e*** and column ***s*** if and only if ***e*** is a member of ***s***
  - Column similarity is the Jaccard similarity of the corresponding sets (rows with value *1)*
  - **Typical matrix is sparse!**
- **Each document is a column:**
  - **Example:** sim(C_1 ,C_2) = ?
    - Size of intersection = 3; size of union = 6, Jaccard similarity (not distance) = 3/6
    - d(C_1,C_2) = 1 – (Jaccard similarity) = 3/6

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

Motivation for Minhash/LSH

- **Suppose we need to find near-duplicate documents among N=1 million documents**
- Naïvely, we'd have to compute **pairwise Jaccard similarites** for every pair of docs
  - i.e, $N(N-1)/2 \approx 5*10^{11}$ comparisons
  - At 10^5 secs/day and 10^6 comparisons/sec, it would take 5 days
- For N = 10 million, it takes more than a year…



MinHashing

Minhashing:

Outline: Finding Similar Columns

- **So far:**
  - Documents → Sets of shingles
  - Represent sets as boolean vectors in a matrix
- **Next Goal:** **Find similar columns, Small signatures**
- **Approach:**
  - **1) Signatures of columns:** small summaries of columns
  - **2) Examine pairs of signatures** to find similar columns
    - **Essential:** Similarities of signatures & columns are related
  - **3) Optional:** Check that columns with similar signatures are really similar
- **Warnings:**
  - Comparing all pairs may take too much time: **Job for LSH**
    - These methods can produce false negatives, and even false positives (if the optional check is not made)

Hashing Columns (Signatures)

- **Key idea:** "hash" each column C to a small ***signature*** $H(C)$, such that:
  - **(1)** $H(C)$ is small enough that the signature fits in RAM
  - **(2)** $sim(C_1, C_2)$ is the same as the "similarity" of signatures $H(C_1)$ and $H(C_2)$
- **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. $H(C_1) \neq H(C_2)$
- **Hash documents into buckets, and 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
- **There is a suitable hash function for Jaccard similarity: Min-hashing**

Estimating $sim(V, W) = |V \cap W| / |V \cup W|$

- **Key Idea:**

If we pick a "winner", say x, among all elements of $V \cup W$ in *a uniformly random manner,* then:

Prob[Element x is the winner] = $1 / |V \cup W|$

and

Prob[$x \in V \cap W$] = $|V \cap W| / |V \cup W| = sim(V, W)$...Eq.(1)

$\Rightarrow$ If we can repeat the experiment many times and be able to detect and count the cases of "winner $\in V \cap W$", we can estimate Prob[$x \in V \cap W$], and thus $sim(V, W)$ (per Eq.(1):

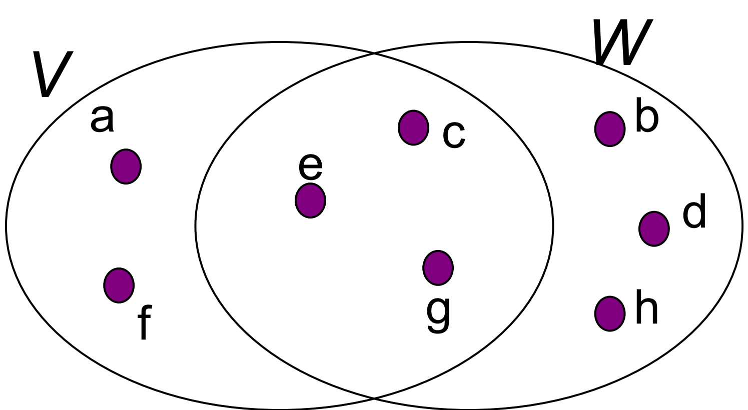
Algorithm 1 Similarity(V,W)

$counter \leftarrow 0$
for $i = 1$ **to** 100 **do**
  Pick a random element $x \in V \cup W$
  if $x \in V \wedge x \in W$ **then**
    $counter \leftarrow counter + 1$
return $counter/100$

Estimating *sim*(*V*, *W*)=|*V*∩*W*|/|*V*∪*W*|

Now, let's use the following way to pick a winner within *V*∪*W* in a uniformly random manner :

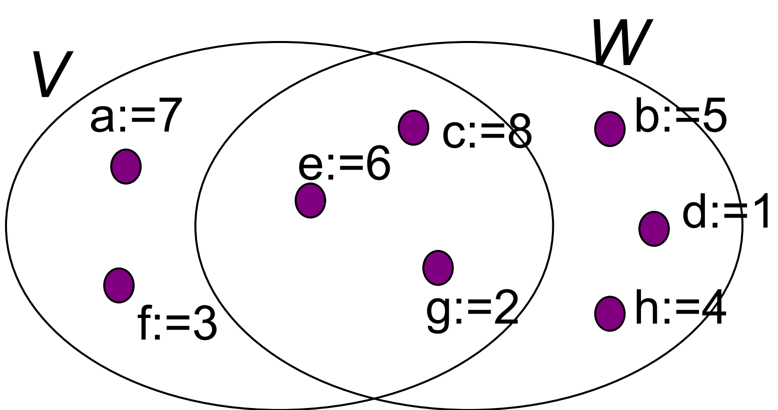
- After randomly permute the ordering of all elements in *V*∪*W*, assign a *unique* value to each element according to its order in the resultant permutation, e.g. "1" to the 1st element, "2" to the 2nd element, and so on ………………………………(*)



Estimating *sim*(*V*, *W*)=|*V*∩*W*|/|*V*∪*W*|

Now, let's use the following way to pick a winner within *V*∪*W* in a uniformly random manner :

- After randomly permute the ordering of all elements in *V*∪*W*, assign a *unique* value to each element according to its order in the resultant permutation, e.g. "1" to the 1st element, "2" to the 2nd element, and so on ……………………………………(*)

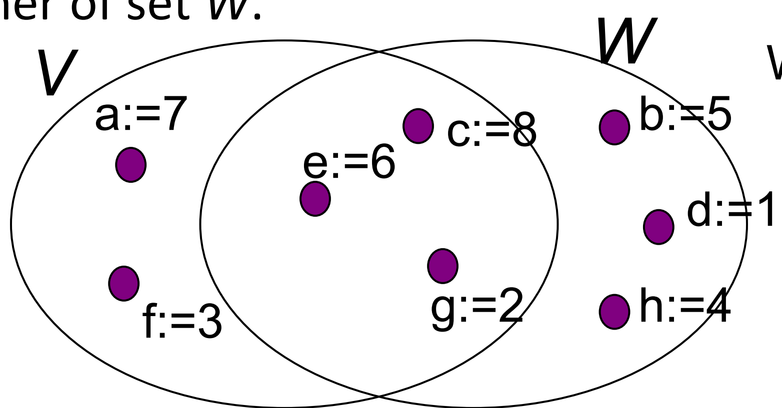


Estimating *sim*(*V*, *W*)=|*V*∩*W*|/|*V*∪*W*|

Now, let's use the following way to pick a winner within *V*∪*W* in a uniformly random manner :

- After randomly permute the ordering of all elements in *V*∪*W*, assign a *unique* value to each element according to its order in the resultant permutation, e.g. "1" to the 1st element, "2" to the 2nd element, and so on ……………………………….(*)
- Among all elements in *V*∪*W*, we declare the element, say x, with the smallest assigned value (according to (*)), the winner of *V*∪*W*.
- Similarly, within set *V*, we declare the element with the smallest assigned value (according to (*)), the winner of set *V*.
- Similarly, within set *W*, we declare the element with the smallest assigned value (according to (*)), the winner of set *W*.

Winner of *V* is g

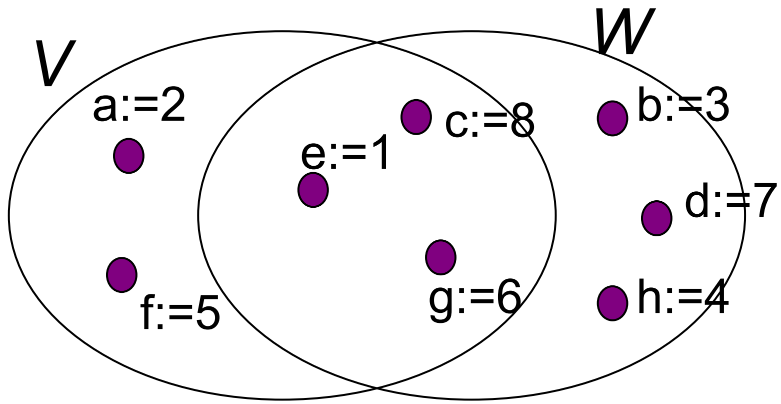


Winner of *V*∪*W* is d

Winner of *W* is also d

Estimating *sim*(*V*, *W*)=|*V*∩*W*|/|*V*∪*W*|

- Now, try another randomly permutation, followed by value assignment ;
- This time, say, e becomes the element with the smallest value assigned and thus the winner!



Notice that e = Winner of $V \cup W$ = Winner of V = Winner of W

(The winning element x $\in V \cap W$) iff (The winner of V is also the winner of W)

Estimating *sim*(V, W)=|V∩W|/|V∪W| (cont'd)

Observe that:

(The winning element x $\in V \cap W$) **iff** (The winner of set V is also the winner of set W) .....(**)

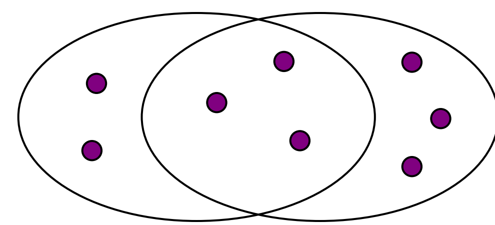
- Since the event of the R.H.S. of (**) is readily observable, we can use this condition to determine whether x, the winning element of the current permutation belongs to $V \cap W$.

- By repeating the experiment in (*) using different random permutations and count the number of times the event specified in the R.H.S. of (**) is observed,

  we can estimate Prob[x $\in V \cap W$] (which is = *sim*(V, W)) according to Eq.(1) as follows:

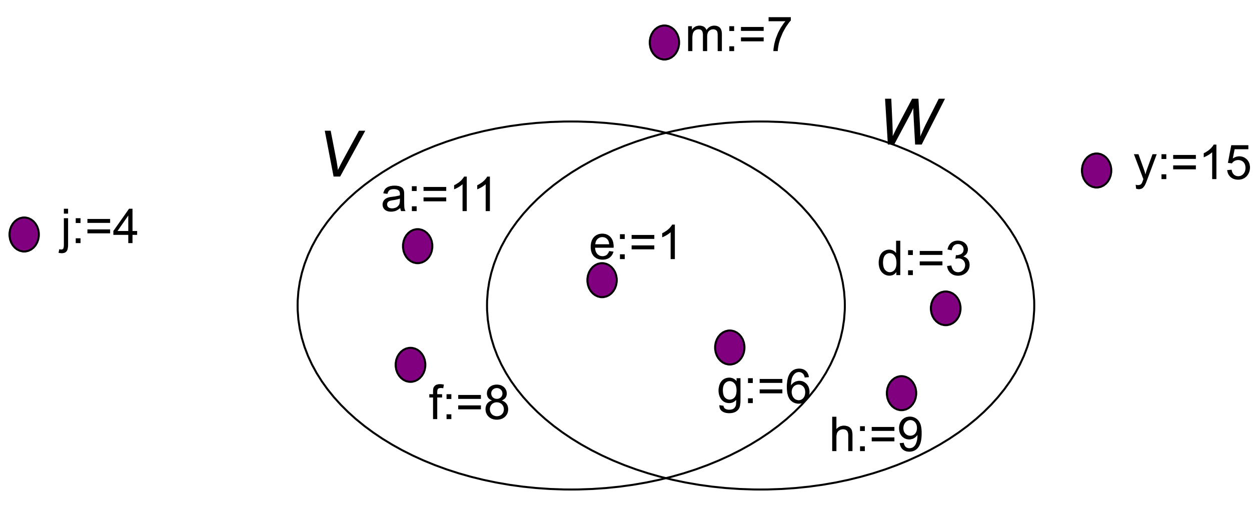
Algorithm 2 Similarity(V, W)

```
counter ← 0
for i = 0 to N do
    Randomly permute the ordering of elements in V ∪ W
    Assign a value to each element according to the resultant order
    if (smallest value within V == smallest value within W)
        counter ← counter + 1
end /* of for */
return (counter / N) /* as an est. of sim(V,W) */
```



Estimating *sim*(*V*, *W*)=|*V*∩*W*|/|*V*∪*W*|

- Now, try another randomly permutation, followed by value assignment ;
- This time, say, e becomes the element with the smallest value assigned and thus the winner!



Notice that e = Winner of *V*∪*W* = Winner of *V* = Winner of *W*

(The winning element x ∈ V ∩ W) iff (The winner of V is also the winner of W)

Min-Hashing Example

Note: An alternative way (equivalent) is to store row #'s of the winning element BEFORE the permutation

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

Element a, i.e. 2nd row after the permutation, is the winner in Col. 1 because it is the first to map to 1 ; Element e can't be the winner for Col. 1 because e does NOT appear in doc. represented by Col. 1.

Permutation π / **Input matrix (Shingles x Documents)**

Elements (Shingles)	π	π	π	Doc 1	Doc 2	Doc 3	Doc 4
a	2	4	3	1	0	1	0
b	3	2	4	1	0	0	1
c	7	1	7	0	1	0	1
d	6	3	2	0	1	0	1
e	1	6	6	0	1	0	1
f	5	7	1	1	0	1	0
g	4	5	5	1	0	1	0

Signature matrix *M*

(stores the row #'s of winning element AFTER permutation.)

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

4th row after the permutation, i.e. element a, is the first to map to a 1

Min-Hashing

- Imagine the rows of the boolean matrix permuted under **random permutation** π
- Define a **"hash" function** $h_\pi(C)$ = the row number of the **first row** (according to permuted order π) in which column C has a value of 1:
  - We skip rows with a zero because it means the corresponding element is NOT a member of Col. C anyway !

Define $h_\pi(C)$ = *row # (after permutation* π*) of winner of Col. C*

Alternatively, we can also use:

$h'_\pi(C)$ = *row # (before permutation* π*) of winner of Col. C*

- Use several (e.g., 100) independent hash (permutation) functions to create a signature of a column.

Min-Hashing Example

Note: Another (equivalent) way is to store row indexes before permutation

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

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 *M*

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

Similarities:

	1-3	2-4	1-2	3-4
Col/Col	0.75	0.75	0	0
Sig/Sig	0.67	1.00	0	0

- Prob[$h_\pi(C_1) = h_\pi(C_2)$] is the same as $sim(D_1, D_2)$

Alternative Derivation for

Pr[*Same winner for* C_1 & C_2] = Pr[$h(C_1) = h(C_2)$] = sim(C_1, C_2)

- **Given cols C_1 and C_2, there are 4 types of rows:**

Type	C_1	C_2
A	1	1
B	1	0
C	0	1
D	0	0

  - **a** = # rows of type A, etc.

By definition of Jaccard Similarity: **sim(C_1, C_2) = a/(a +b +c)**......Eq.(2)

Now, after random shuffling of rows, look down the cols of C_1 and C_2 row-by-row until we see at least one 1: (i.e. a winner is detected)

- If it's a type-*A* row => same winner for C_1 and C_2 , i.e. $h(C_1) = h(C_2)$,
- If a type-*B* or type-*C* row, then different winners for C_1 and C_2

BUT: **Pr**[*Same winner for* C_1 and C_2]

= **Pr** [$h(C_1) = h(C_2)$] = **Pr**[$h'(C_1) = h'(C_2)$]

= **Pr**[*Reaching a type-A row before a type-B or type-C row*]

= **a/(a +b +c)** /* due to the # of type-A,B,C, rows in C_1 and C_2 */

= **sim(C_1, C_2)** /* by Eq.(2) */

Similarity for Signatures

As a result, we have:

Pr[*winner of* C_1 = *winner of* C_2] =
Pr[$h_\pi(C_1) = h_\pi(C_2)$] = *sim*(C_1, C_2).......................... Eq.(3)

- We will use multiple hash functions to realize different random permutations among the elements within the Columns
- **Define the *similarity of two signatures* to be the fraction of the hash functions in which they agree**
- Per Eq.(3), the similarity of columns (2 sets) is the same as the expected ***similarity of their signatures***

MinHash Signatures

- **Pick K=100 random permutations of the rows**
- Think of ***sig*(C)** as a column vector

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

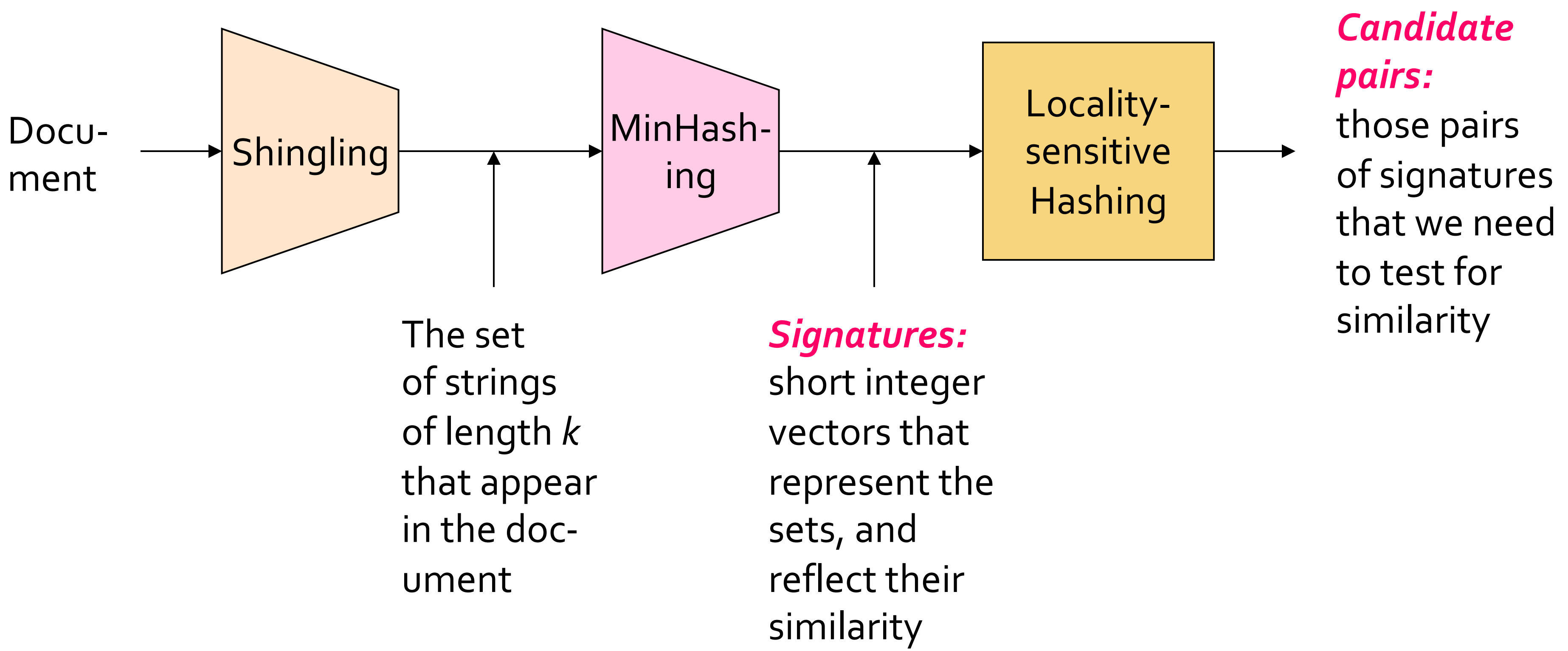
Note: The sketch (signature) of document *C* is small -- ~100 bytes!

- **We achieved our goal! We "compressed" long bit vectors into short signatures**

Implementation Trick

- **Permuting rows even once is prohibitive**
- **Row hashing!**
  - Pick **K = 100** hash functions k_i
  - Ordering under k_i gives a random row permutation!
- **One-pass implementation**
  - For each column C and hash-func. k_i keep a "slot" for the min-hash value
  - Initialize all $sig(C)[i] = \infty$
  - **Scan rows looking for 1s**
    - Suppose row j has 1 in column C
    - Then for each k_i :
      - If $k_i(j) < sig(C)[i]$, then $sig(C)[i] \leftarrow k_i(j)$

How to pick a random hash function h(x)?
Universal hashing:
$h_{a,b}(x)=((a \cdot x+b) \bmod p) \bmod N$
where:
a,b … random integers
p … prime number (p > N)



Locality Sensitive Hashing

Locality-sensitive hashing:

LSH: First Cut

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

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

- **LSH – General idea:** Use a function ***f(x,y)*** that tells whether ***x*** and ***y*** is a ***candidate pair:*** a pair of elements whose similarity must be evaluated

- **For minhash matrices:**
  - Hash columns of signature matrix ***M*** to many buckets
  - Each pair of documents that hashes into the same bucket is a **candidate pair**

Candidates from Minhash

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

- Pick a similarity threshold ***s*** (0 < s < 1)
- Columns ***x*** and ***y*** of ***M*** are a **candidate pair** if their signatures agree on at least fraction ***s*** of their rows:
  M (i, x) = M (i, y) for at least frac. ***s*** values of ***i***
  - We expect documents ***x*** and ***y*** to have the same (Jaccard) similarity as is the similarity of their signatures

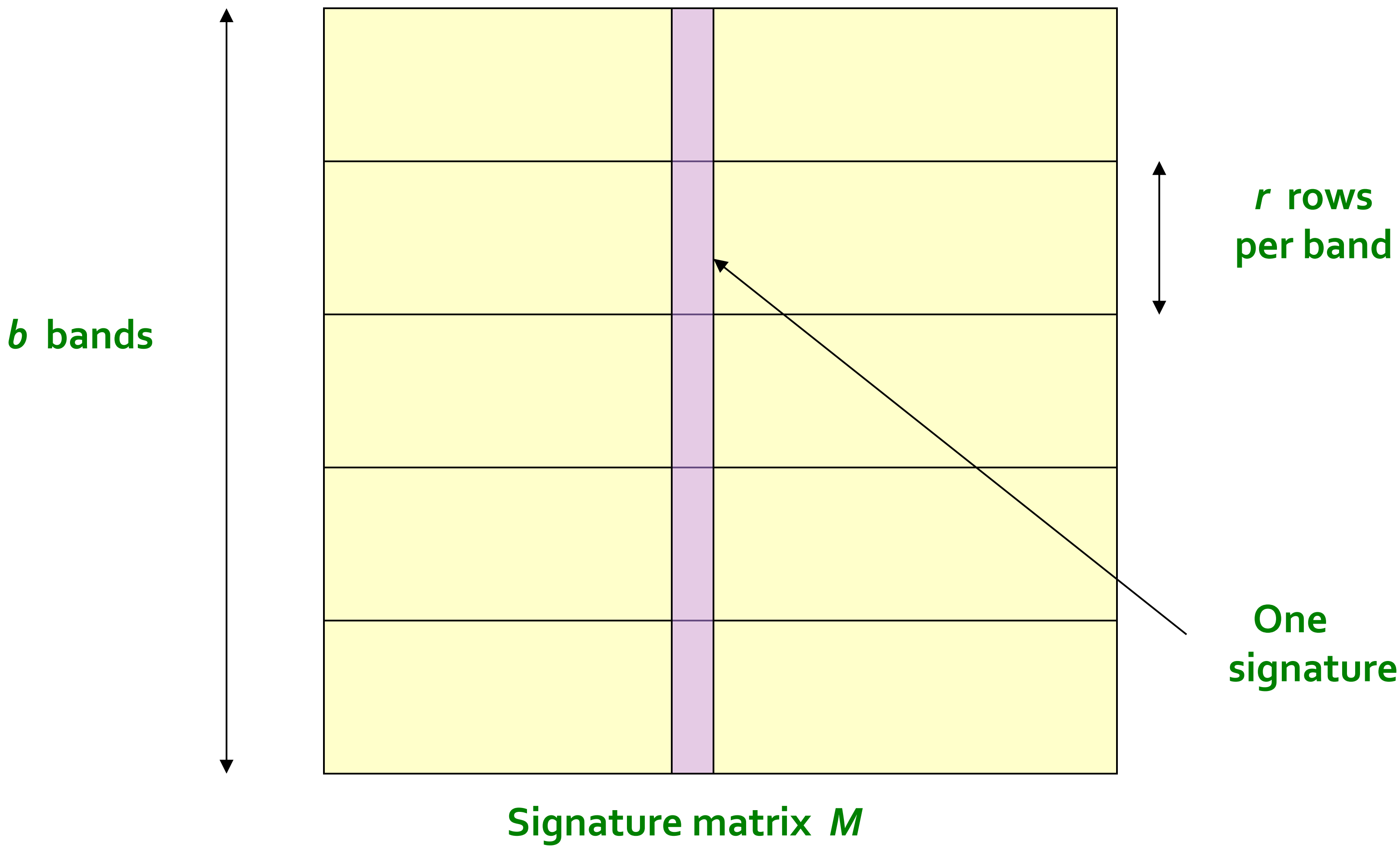
LSH for Minhash

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

- **Big idea: Hash columns of signature matrix *M* several times**
- Arrange that (only) **similar columns** are likely to **hash to the same bucket**, with high probability
- **Candidate pairs are those that hash to the same bucket**

Partition *M* into *b* Bands

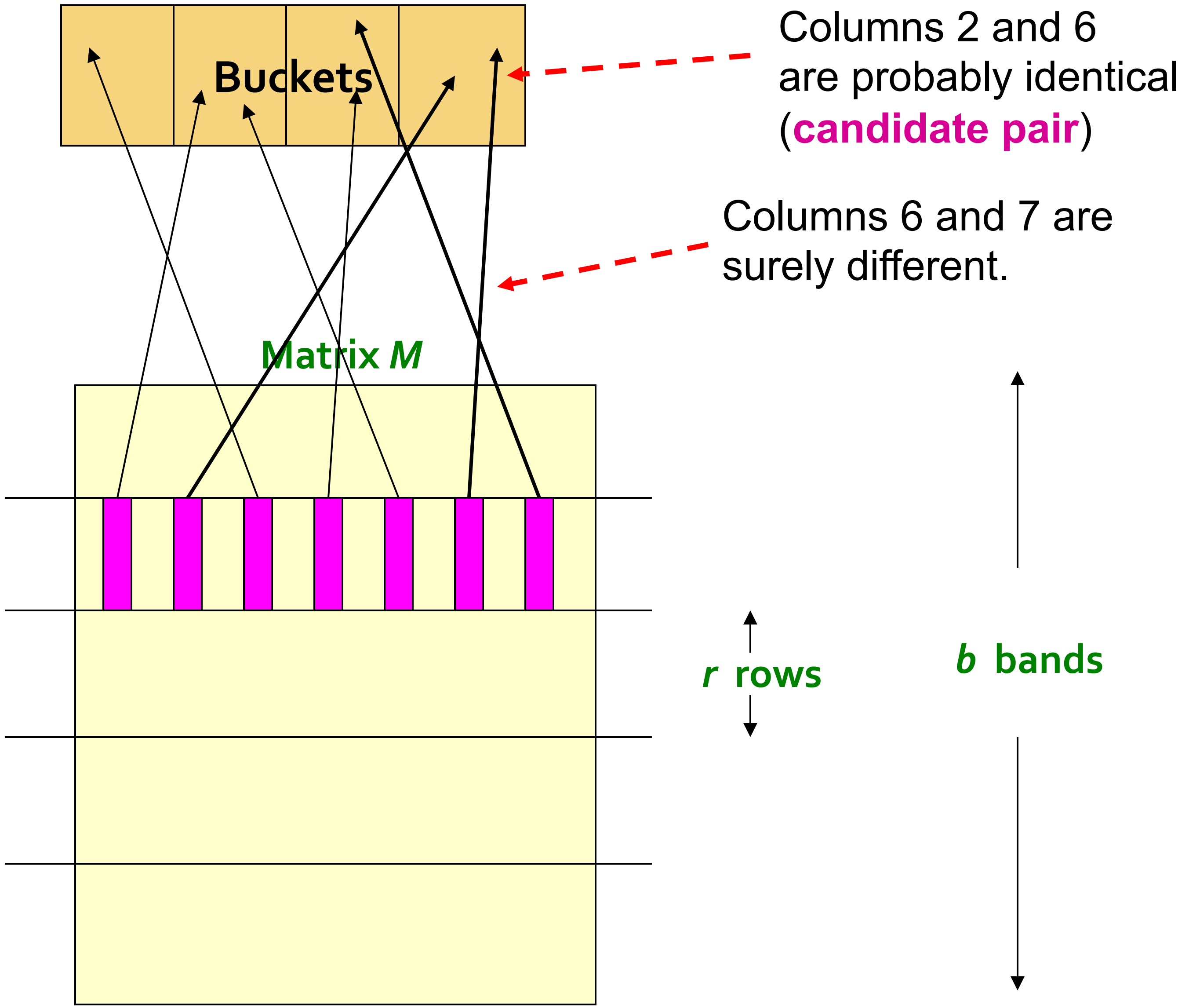
2	1	4	1
1	2	1	2
2	1	2	1



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 ≥ **1** band
- Tune ***b*** and ***r*** to catch most similar pairs, but few non-similar pairs

Hashing Bands



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

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

Assume the following case:

- Suppose 100,000 columns of ***M*** (100k docs)
- Signatures of 100 integers (rows)
- Therefore, signatures take 40Mb
- 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

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

- **Find pairs of** $\geq$ ***s***=0.8 similarity, set **b**=20, **r**=5
- **Assume:** 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 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 of the 20 bands: $(1-0.328)^{20} = 0.00035$
  - 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

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

- **Find pairs of ≥ *s*=0.8 similarity, set b=20, r=5**
- **Assume:** $sim(C_1, C_2) = 0.3$
  - Since $sim(C_1, C_2) < \mathbf{s}$ 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 then it will turn out their similarity is below threshold **s**

LSH Involves a Tradeoff

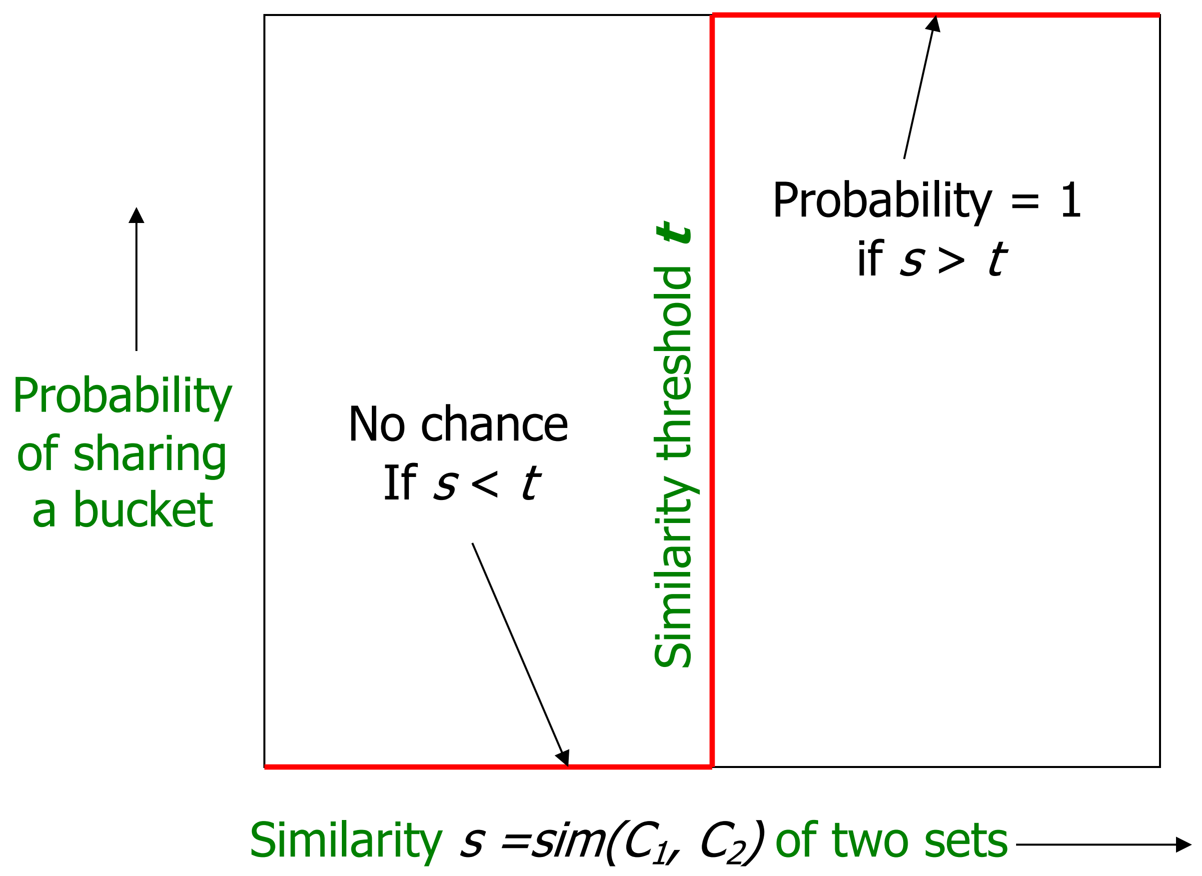
2	1	4	1
1	2	1	2
2	1	2	1

- **Pick:**
  - the number of minhashes (rows of ***M***)
  - the number of bands ***b***, and
  - the number of rows ***r*** per band

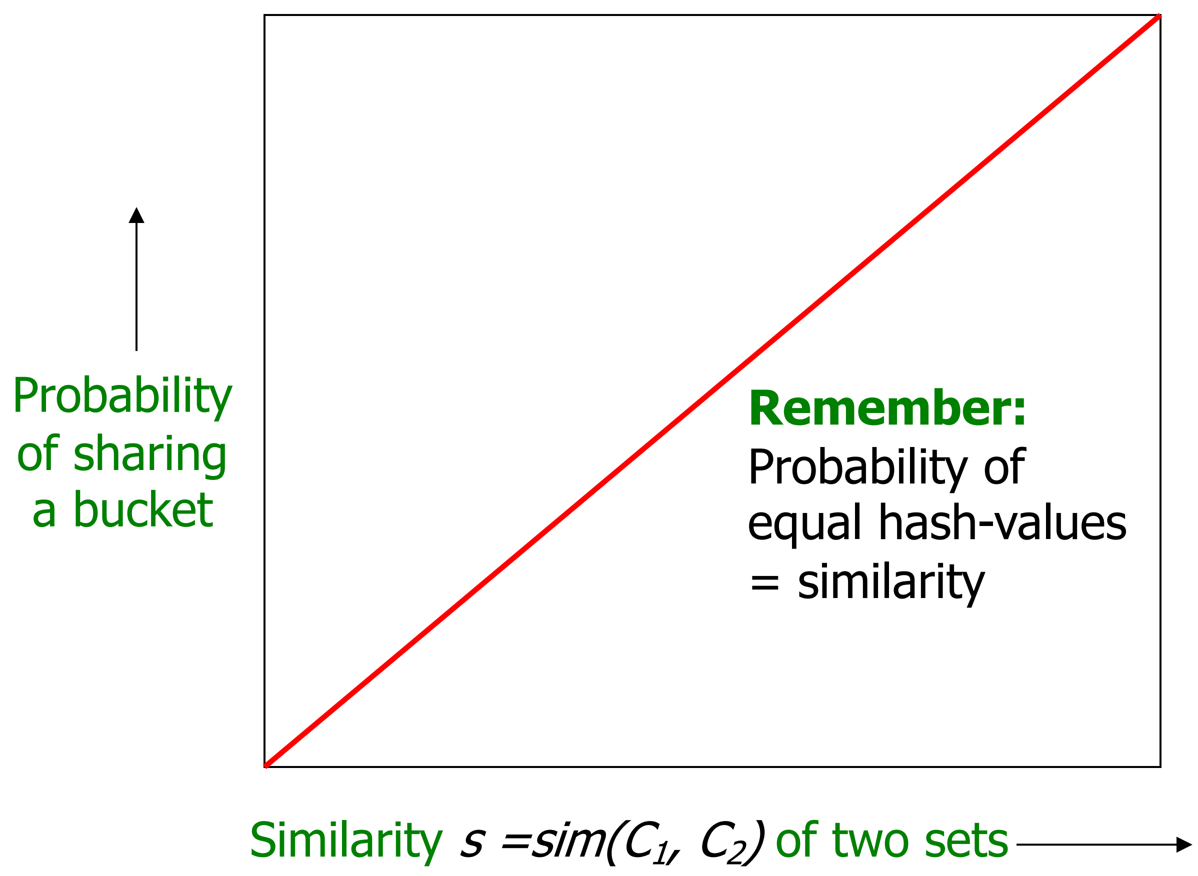
  to balance false positives/negatives

- **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

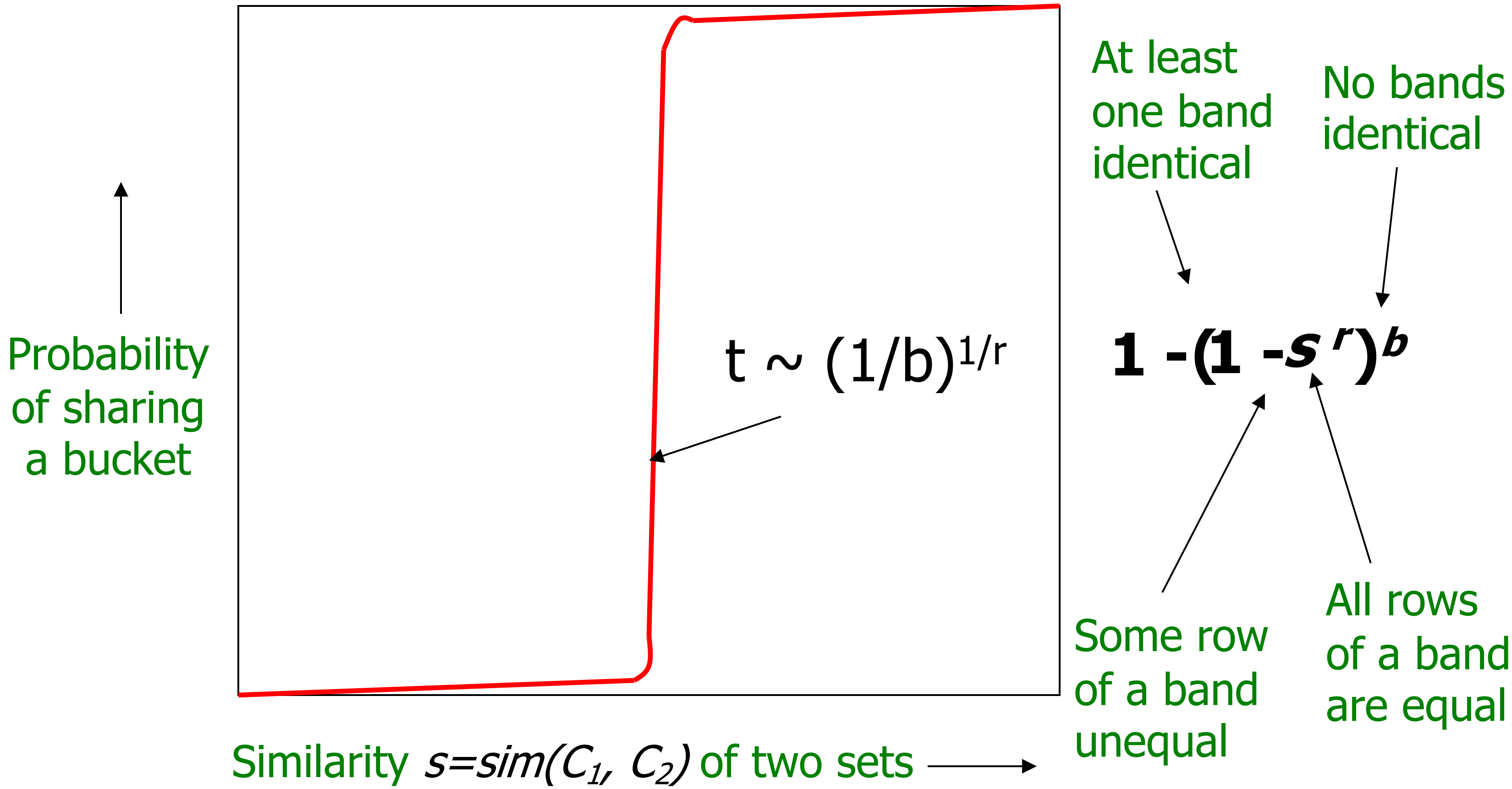


b bands, *r* rows/band

- Columns C_1 and C_2 have similarity s
- Pick any band (r rows)
  - Prob. that all rows in band equal = s^r
  - Prob. that some row in band unequal = $1 - s^r$
- Prob. that no band identical = $(1 - s^r)^b$
- Prob. that at least 1 band identical =

$$1 - (1 - s^r)^b$$

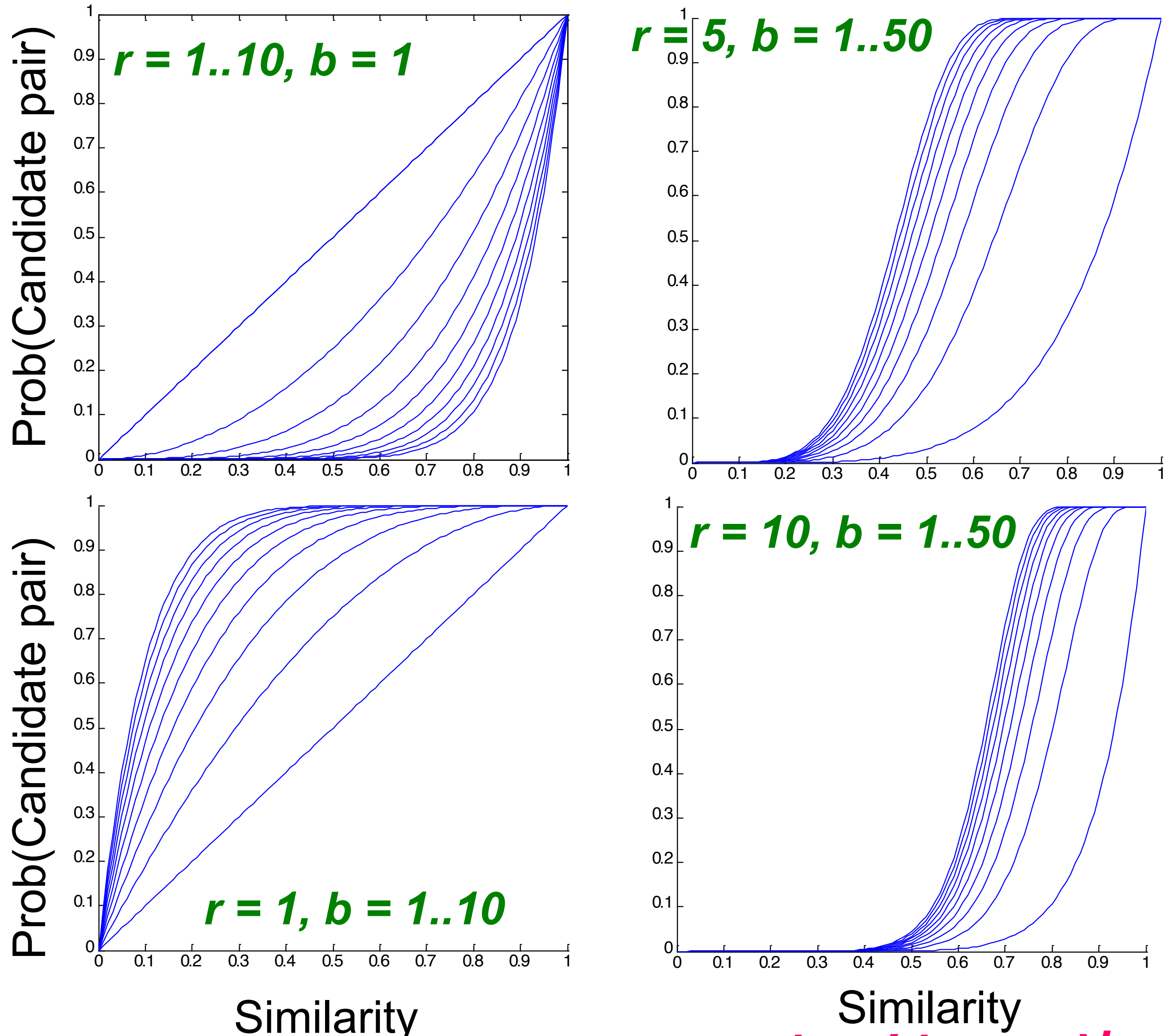
What *b* Bands of *r* Rows Gives You



S-curves as a Func. of *b* and *r*

Given a fixed threshold ***t***.

We want to choose ***r*** and ***b*** such that the ***P(Candidate pair)*** has a "step" right around ***t***.



$y = 1 - (1 - s^r)^b$

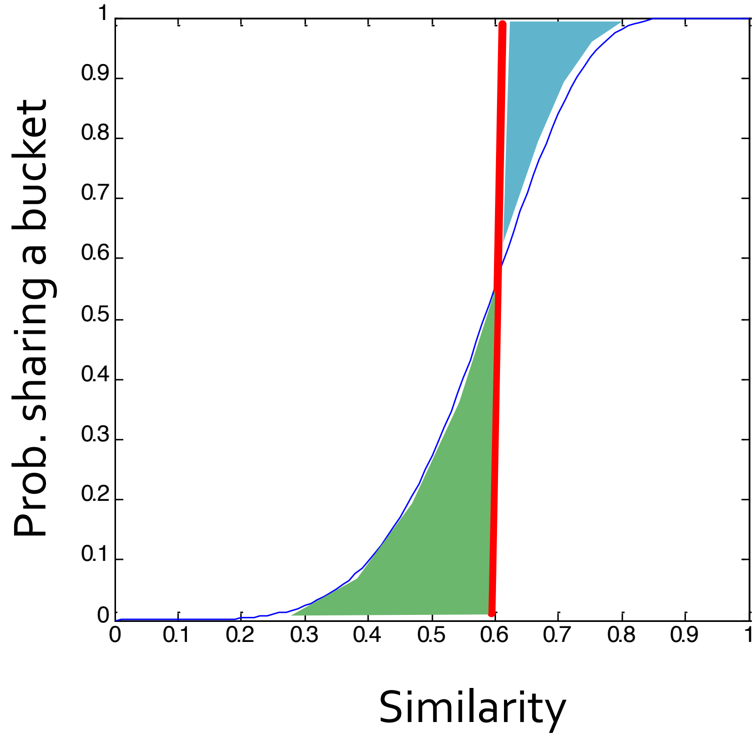
Example: $b = 20$; $r = 5$

- Similarity level 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)



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

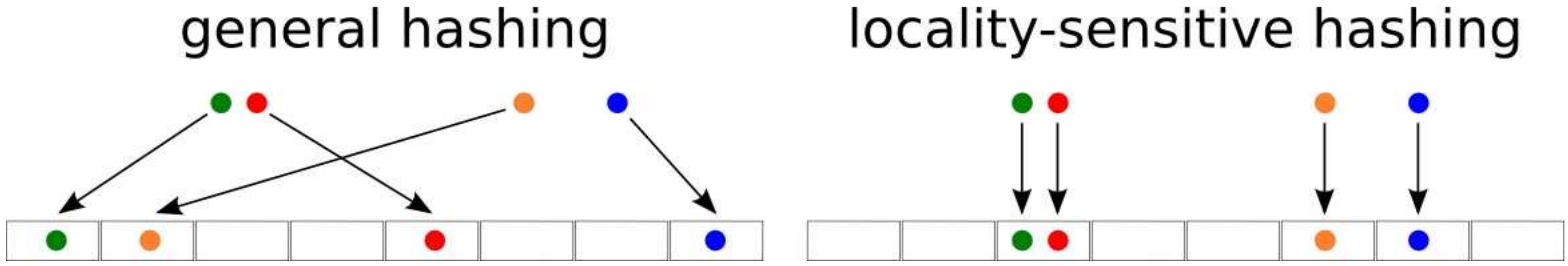
LSH Summary

- 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**
- **Optional:** In another pass through data, check that the remaining candidate pairs really represent similar documents

Summary: 3 Steps

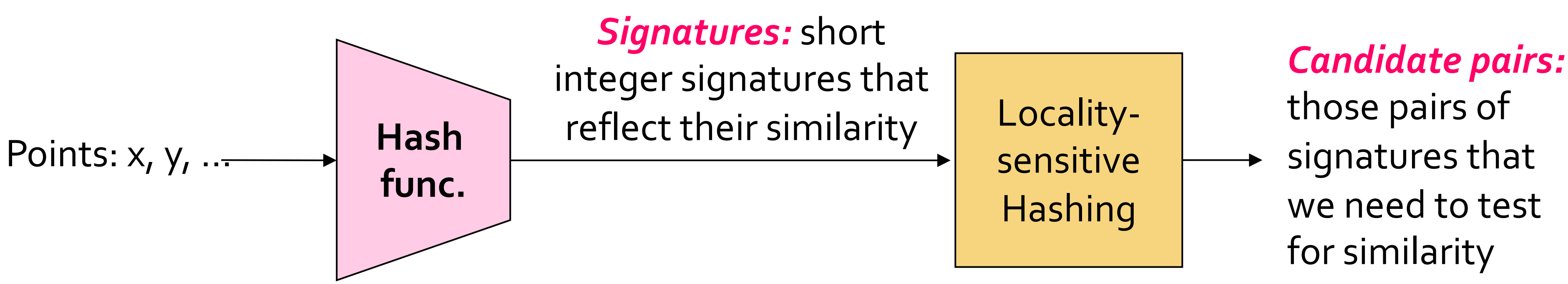
- **Shingling:** Convert documents to sets
  - We used hashing to assign each shingle an ID
- **Min-hashing:** Convert large sets to short signatures, while preserving similarity
  - We used **similarity preserving hashing** to generate signatures with property $\mathbf{Pr}[h_\pi(C_1) = h_\pi(C_2)] = sim(C_1, C_2)$
  - We used hashing to get around generating random permutations
- **Locality-sensitive hashing:** Focus on pairs of signatures likely to be from similar documents
  - We used hashing to find **candidate pairs** of similarity $\geq$ **s**
  - **Notice that MinHash is only good for constructing LSH under the Jaccard similarity ;**
    - **Other Hash functions exist for LSH under for other similarity metrics, e.g. cosine similarity or hamming distance etc.**

Theory of Locality Sensitive Hashing (LSH)



Generalization of LSH for other Distance Metrics

- **MinHash works for Jaccard Similiarity [d(x,y) = 1 – sim(x,y)]**
- **Different LSH methods for other distance metrics:**
  - **Cosine distance,**
  - **Euclidean distance etc**



Design a family of hash functions s.t.

If $d(x,y) \le d_1$, then prob. that $h(x) = h(y)$ is high, e.g. at least p_1

AND

If $d(x,y) \ge d_2$, then prob. that $h(x) = h(y)$ is low, e.g. at most p_2

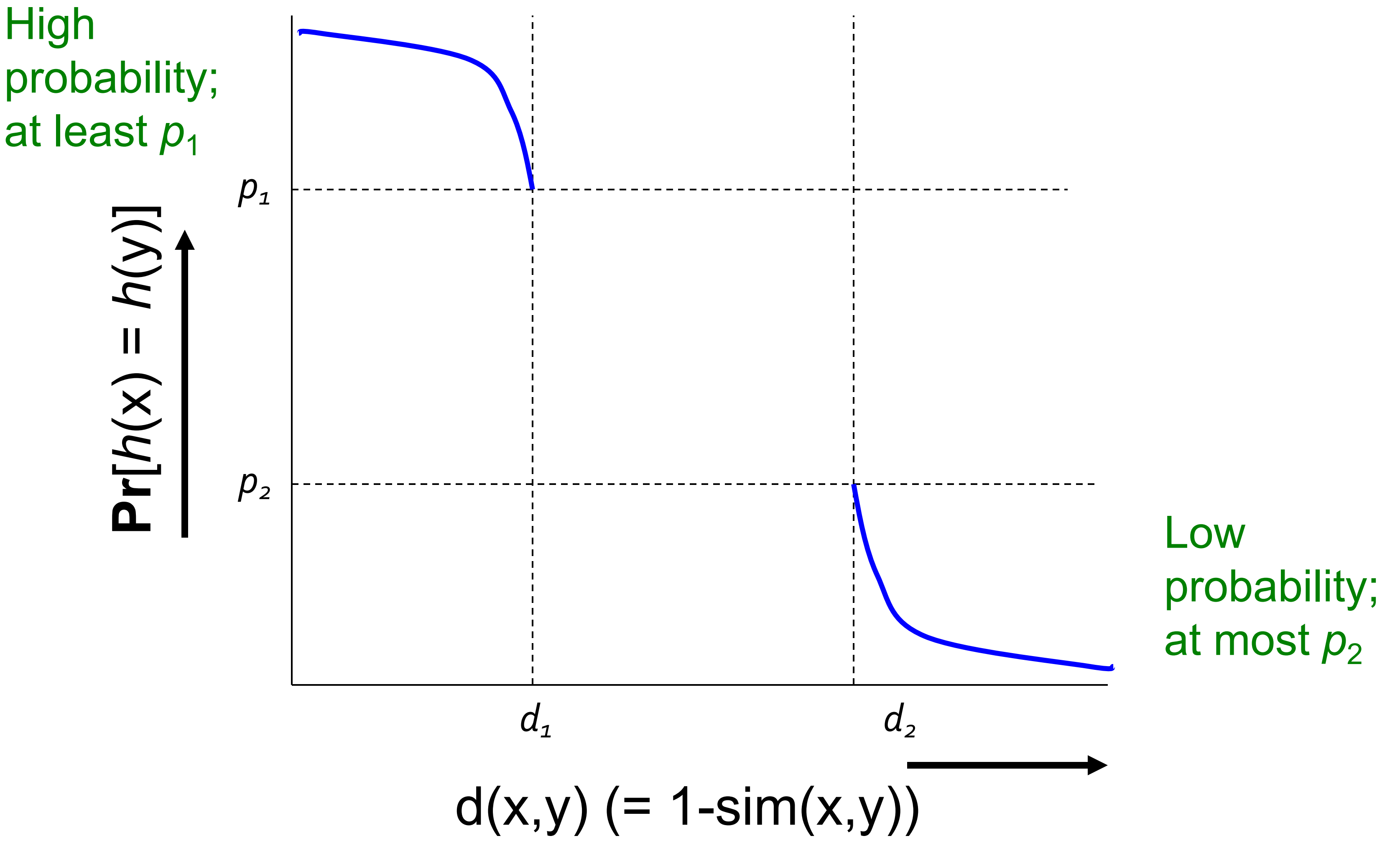
The design of the hash function h( ) will depend on the choice of distance metric, d(x,y) = 1 – sim(x,y)

Amplify the hash family to get the "S" curve using *AND* and *OR* constructions

Locality-Sensitive (LS) Families

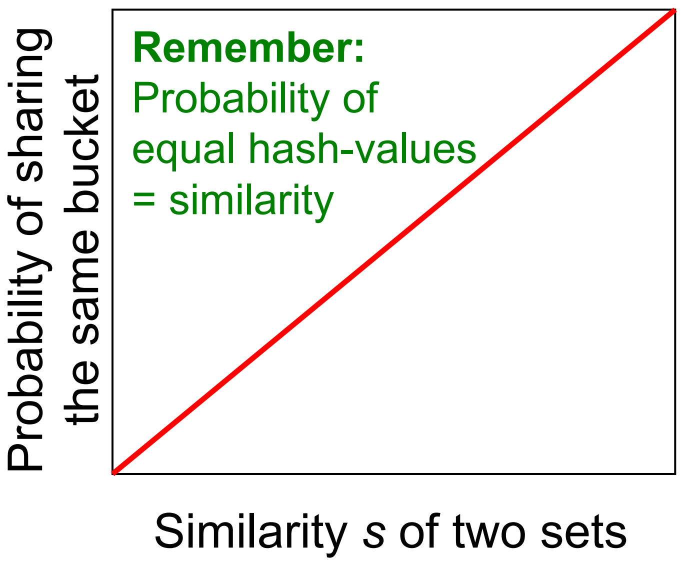
- **Suppose we have a space *S* of points with a distance measure *d***
- A family ***H*** of hash functions is said to be ***(d_1, d_2, p_1, p_2)-sensitive*** if for any ***x*** and ***y*** in ***S***:
  1. If $d(x, y) \leq d_1$, then the probability over all $h \in H$, that $h(x) = h(y)$ is at least p_1
  2. If $d(x, y) \geq d_2$, then the probability over all $h \in H$, that $h(x) = h(y)$ is at most p_2

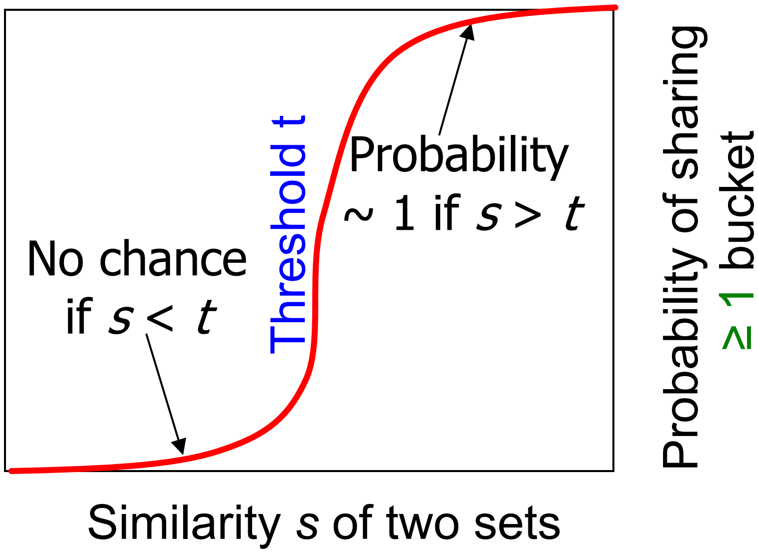
A (d_1,d_2,p_1,p_2)-sensitive function



Recap: The S-Curve

- **The S-curve is where the “magic” happens**





This is what 1 band & 1 row gives you

$\Pr[h_\pi(C_1) = h_\pi(C_2)] = sim(D_1, D_2)$

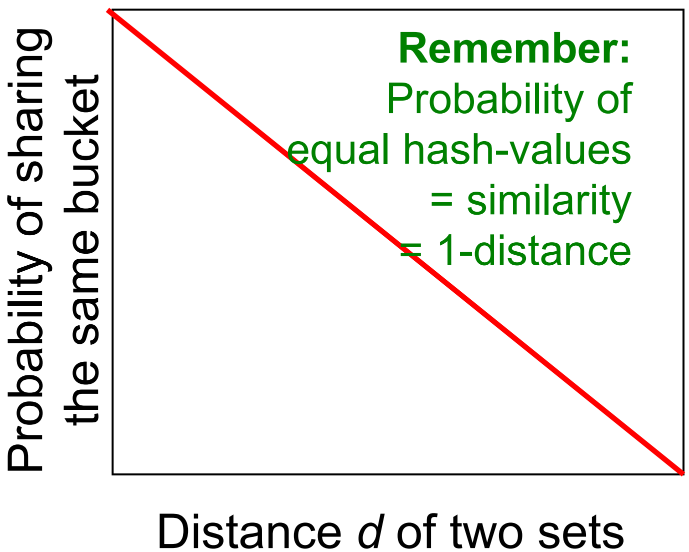
This is what we want!

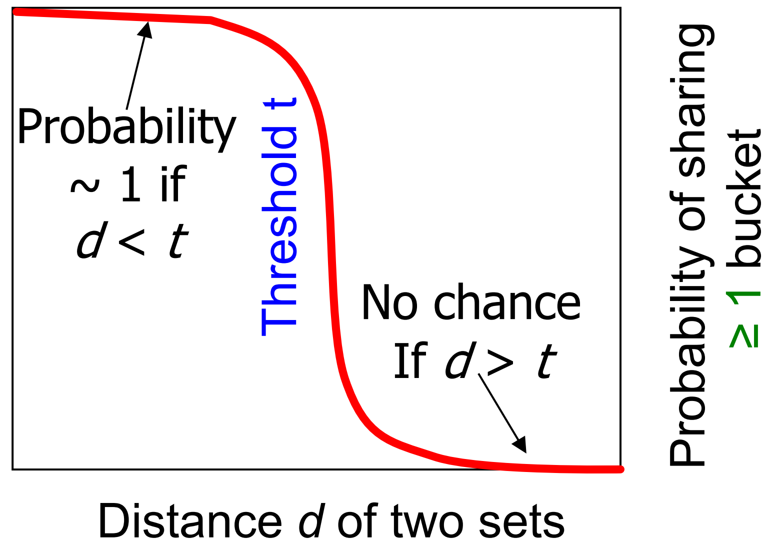
How to get a step-function?

By choosing *r rows* and *b bands*!

Recap: The S-Curve

- **The S-curve is where the “magic” happens**





This is what 1 band and 1 row gives you

$\Pr[h_\pi(C_1) = h_\pi(C_2)] = \text{sim}(D_1, D_2)$

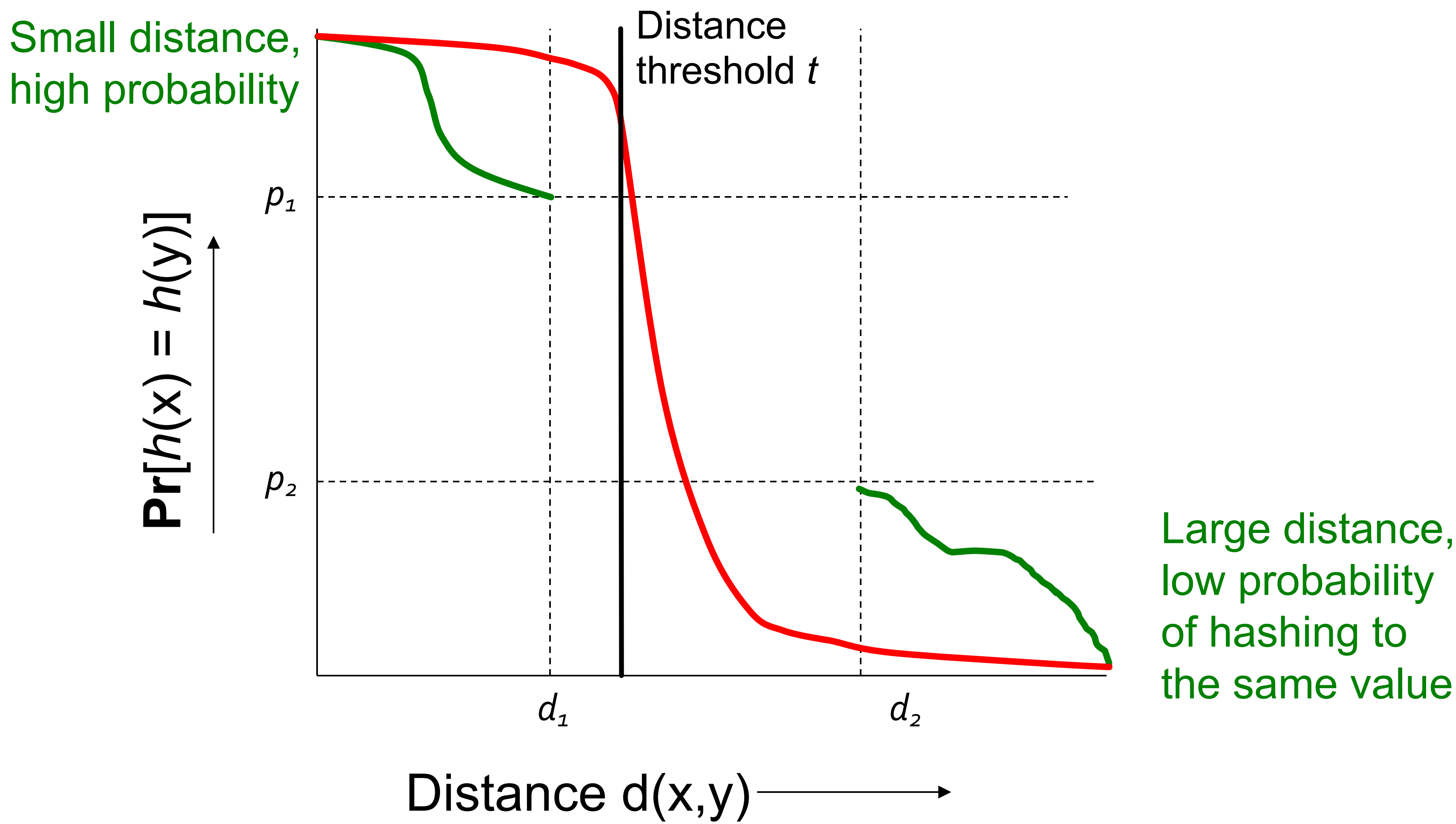
$= 1 - \text{distance}(D_1, D_2)$

This is what we want!

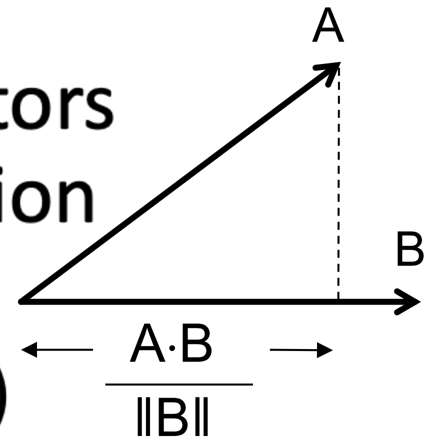
How to get a step-function?

By choosing *r* rows and *b bands* !

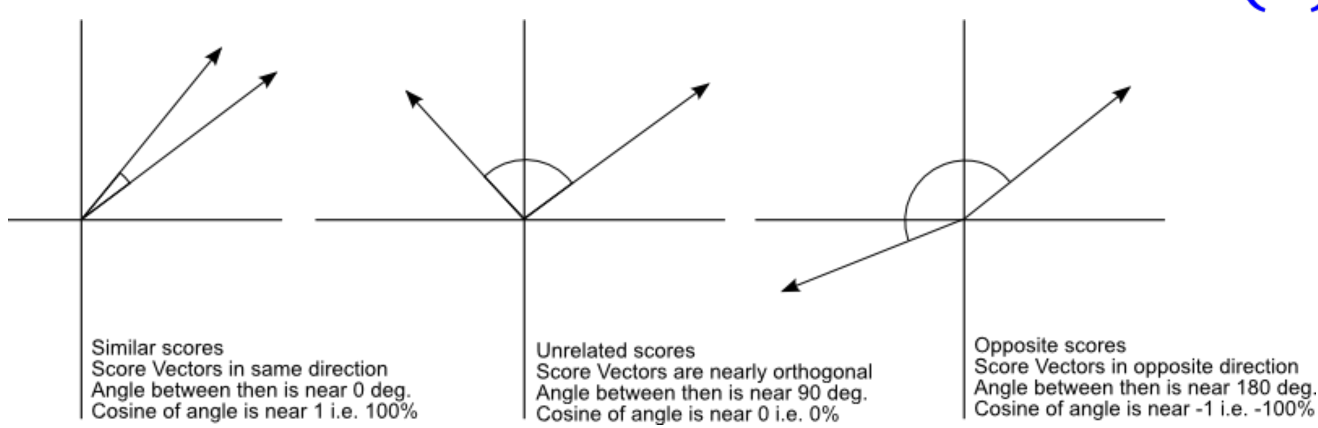
A (d_1,d_2,p_1,p_2)-sensitive function



Cosine Distance



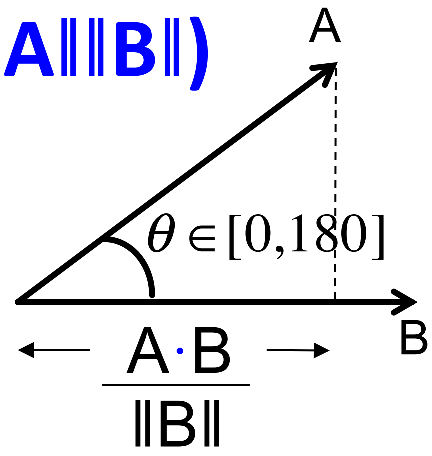
- ***Cosine distance*** = angle between vectors from the origin to the points in question
  d(A, B) = θ = arccos(A·B / ‖A‖·‖B‖)
  - Has range $\mathbf{0} \ldots \boldsymbol{\pi}$ (equivalently 0...180°)
  - Can divide θ by $\boldsymbol{\pi}$ to have distance in range 0...1
- **Cosine similarity = 1-d(A,B)**
  - But often defined as **cosine sim:** $\cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|}$



- Has range -1…1 for general vectors
- Range 0..1 for non-negative vectors (angles up to 90°)

LSH for Cosine Distance

- For **cosine similarity** = $\cos\theta = (A \cdot B / \|A\|\|B\|)$
- **cosine distance** $d(A, B) = \theta/180$
- There is a technique called **Random Hyperplanes**
  - Technique similar to Minhashing
- **Random Hyperplanes** is a *$(d_1, d_2, (1-d_1/180), (1-d_2/180))$-sensitive* family for any d_1 and d_2



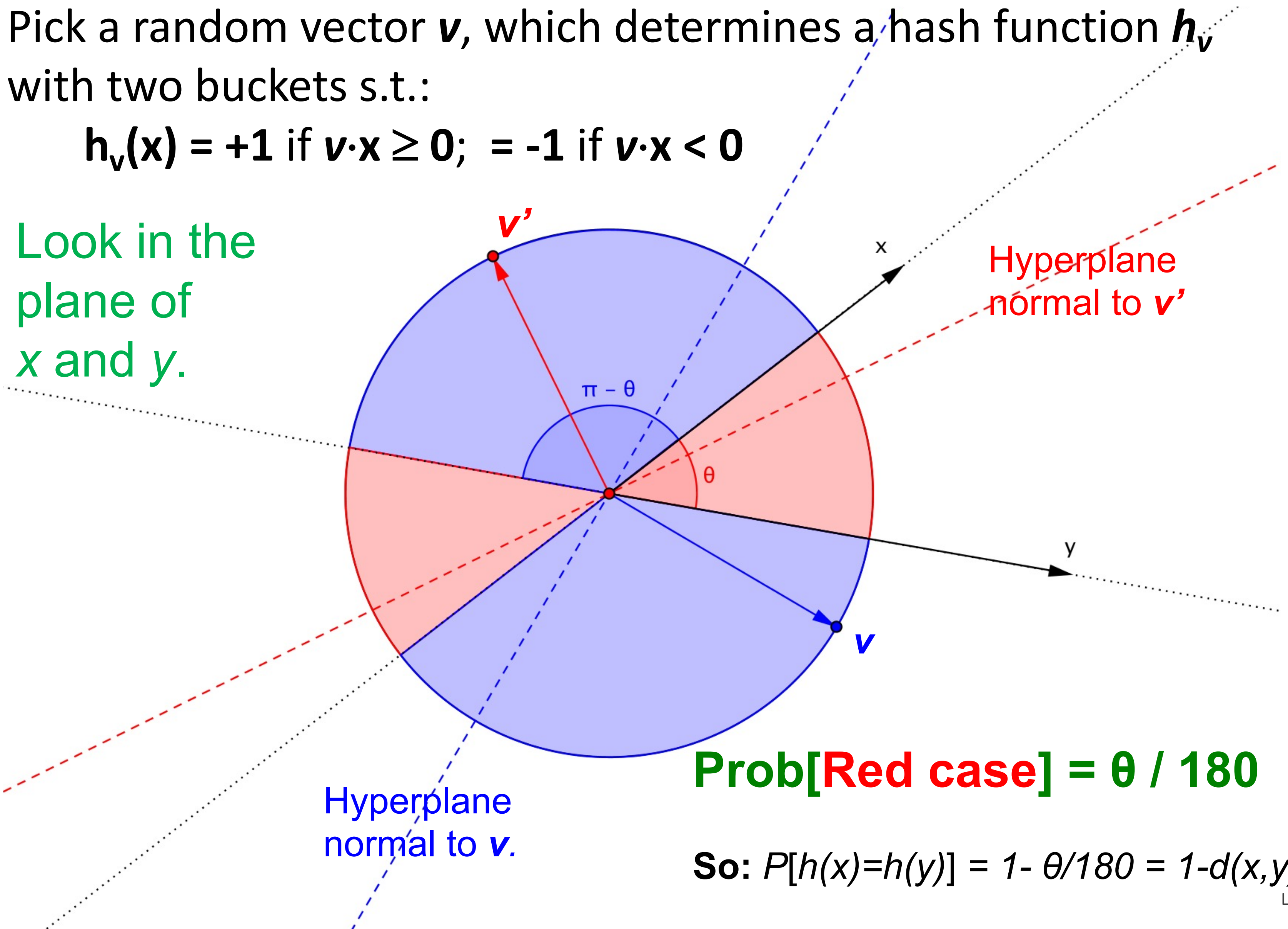
- **Reminder:** *(d_1, d_2, p_1, p_2)-sensitive*
  1. If $d(x,y) \leq d_1$, then prob. that $h(x) = h(y)$ is at least p_1
  2. If $d(x,y) \geq d_2$, then prob. that $h(x) = h(y)$ is at most p_2

Random Hyperplane

- Pick a random vector ***v***, which determines a hash function h_v with two buckets s.t.:

$$h_v(x) = +1 \text{ if } v \cdot x \geq 0; \;\; = -1 \text{ if } v \cdot x < 0$$

Look in the plane of *x* and *y*.



Prob[Red case] = θ / 180

So: $P[h(x)=h(y)] = 1- \theta/180 = 1-d(x,y)$

Signatures for Cosine Distance

- Pick some number of random vectors *v*, and hash your data for each vector

- The result is a signature (*sketch*) of **+1**'s and **–1**'s for each data point: x, y, …

- Can be used for LSH like we used the Minhash signatures for Jaccard distance

- Amplify using **AND/OR** constructions

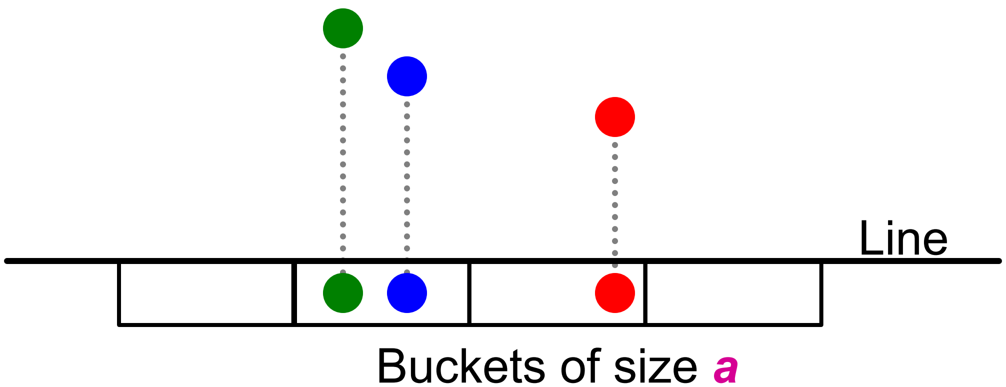
How to pick random vectors?

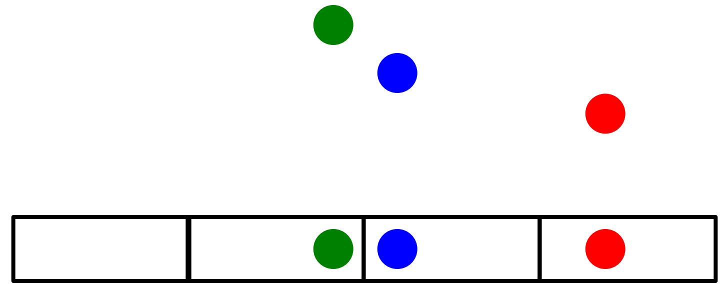
- Expensive to pick a random vector in ***M*** dimensions for large ***M***
  - Would have to generate ***M*** random numbers
- **A more efficient (but approximated) approach**
  - It is "close enough" to consider only vectors ***v*** consisting of +1 and –1 components
    - **Why?** Assuming data is random, then vectors of +/-1 cover the entire space evenly (and does not bias in any way)
      - This only gives an APPROXIMATED result, but not an exact one !!

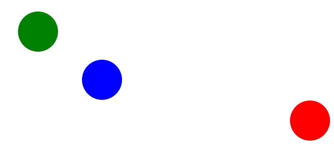
LSH for Euclidean Distance

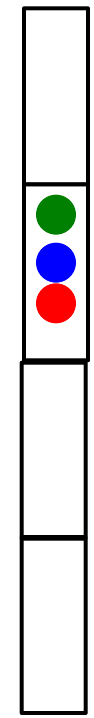
- **Simple idea: Hash functions correspond to lines**
- Partition the line into buckets of size ***a***
- **Hash each point to the bucket containing its projection onto the line**
- **Nearby points are always close**;
  distant points are rarely in same bucket

Projection of Points



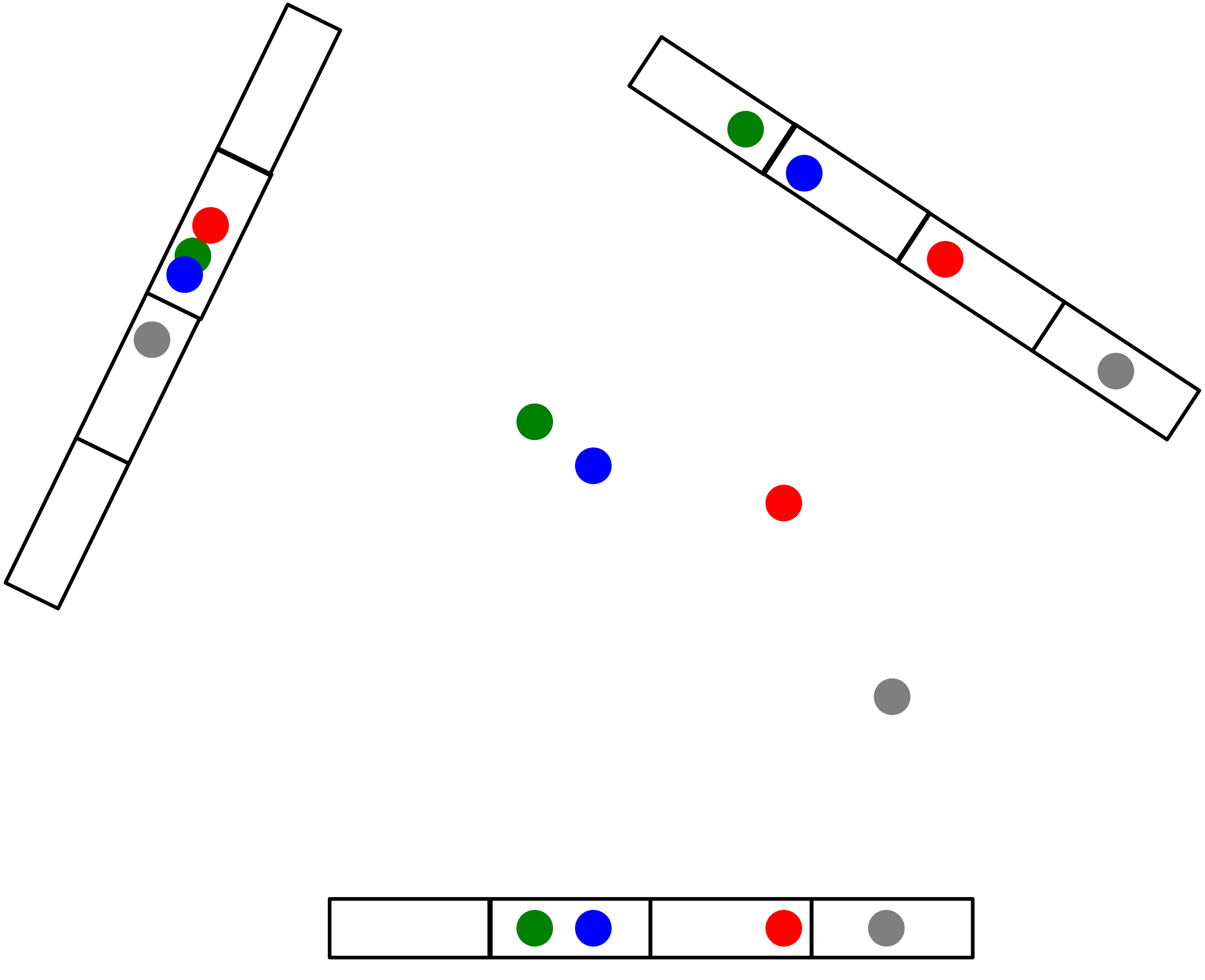




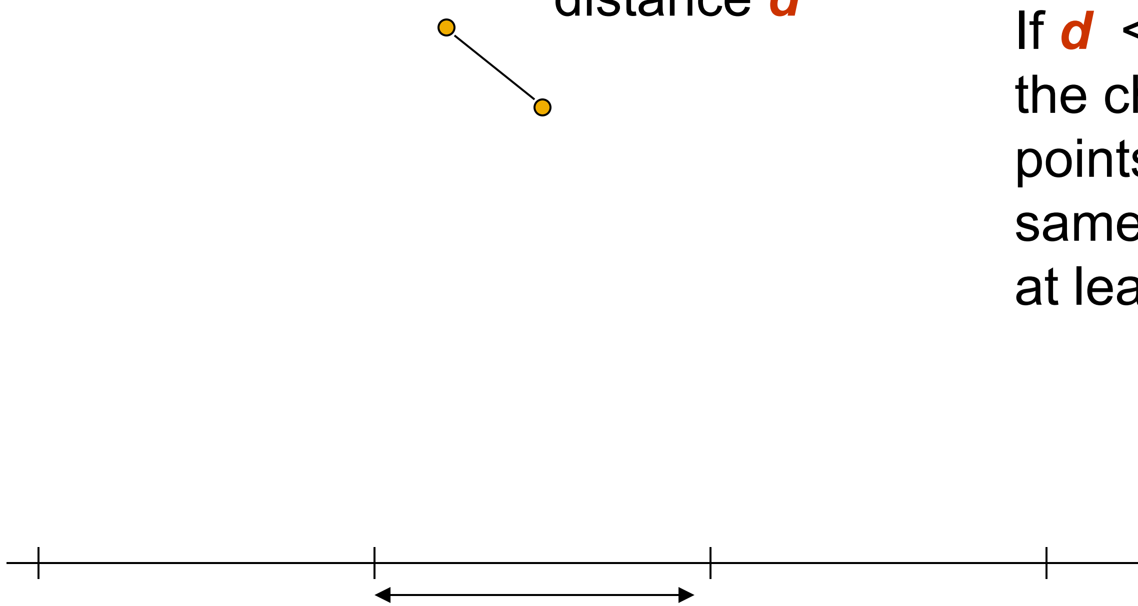


- **"Lucky" case:**
  - Points that are close hash in the same bucket
  - Distant points end up in different buckets
- **Two "unlucky cases:**
  - **Top:** unlucky quantization
  - **Bottom:** unlucky projection

Multiple Projections



Projection of Points

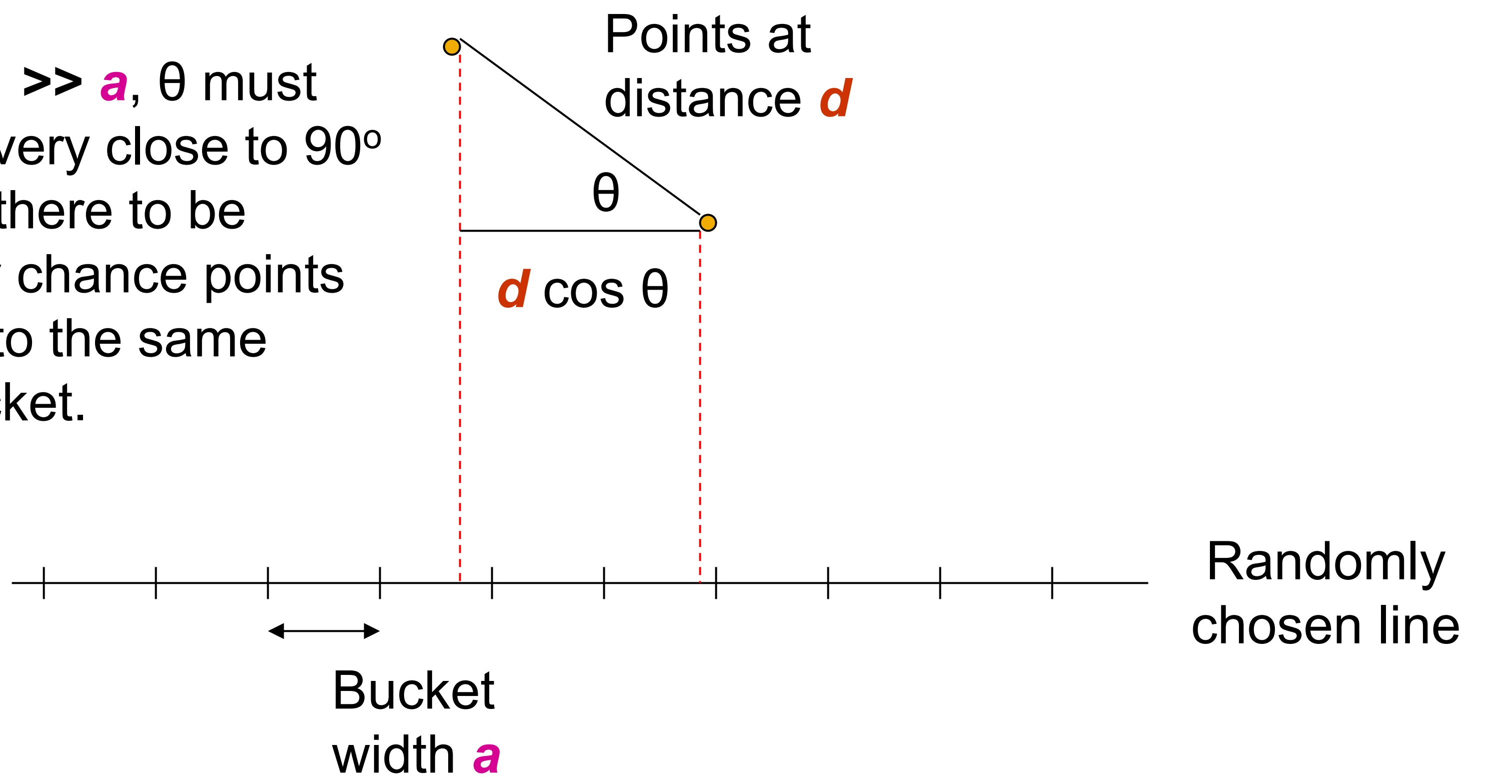


If *d* << *a*, then the chance the points are in the same bucket is at least **1** – *d*/*a*.

- For example, If points are distance ***d* ≤ *a*/2**, the probability they are in same bucket **≥ 1- *d*/*a* = ½**

Projection of Points

If ***d*** **>>** ***a***, θ must be very close to 90^o for there to be any chance points go to the same bucket.



- For example, if points are distance $\boldsymbol{d \geq 2a}$ apart,
  then they can be in the same bucket only if $\boldsymbol{d \cos \theta \leq a}$
  => $\cos \theta \leq 1/2$
  So, for $60 \leq \theta \leq 90$, i.e., at most 1/3 probability

An LSH-Family for Euclidean Distance

- If points are distance $d \leq a/2$, prob. they are in same bucket $\geq 1 - d/a = ½$
- If points are distance $d \geq 2a$ apart, then they can be in the same bucket only if $d \cos \theta \leq a$
  - $\cos \theta \leq ½$
  - $60 \leq \theta \leq 90$, i.e., at most 1/3 probability
- Yields a ***(a/2, 2a, 1/2, 1/3)**-sensitive* family of hash functions for any ***a***
- **Amplify using AND-OR cascades**