Finding Similar Items

Liyao Xiang
xiangliyao.cn
Shanghai Jiao Tong University

Finding Similar Documents

- **Goal:** Given a large number (N in the millions or billions) of documents, 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'
- **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**

- How do we represent these documents to compare them efficiently?

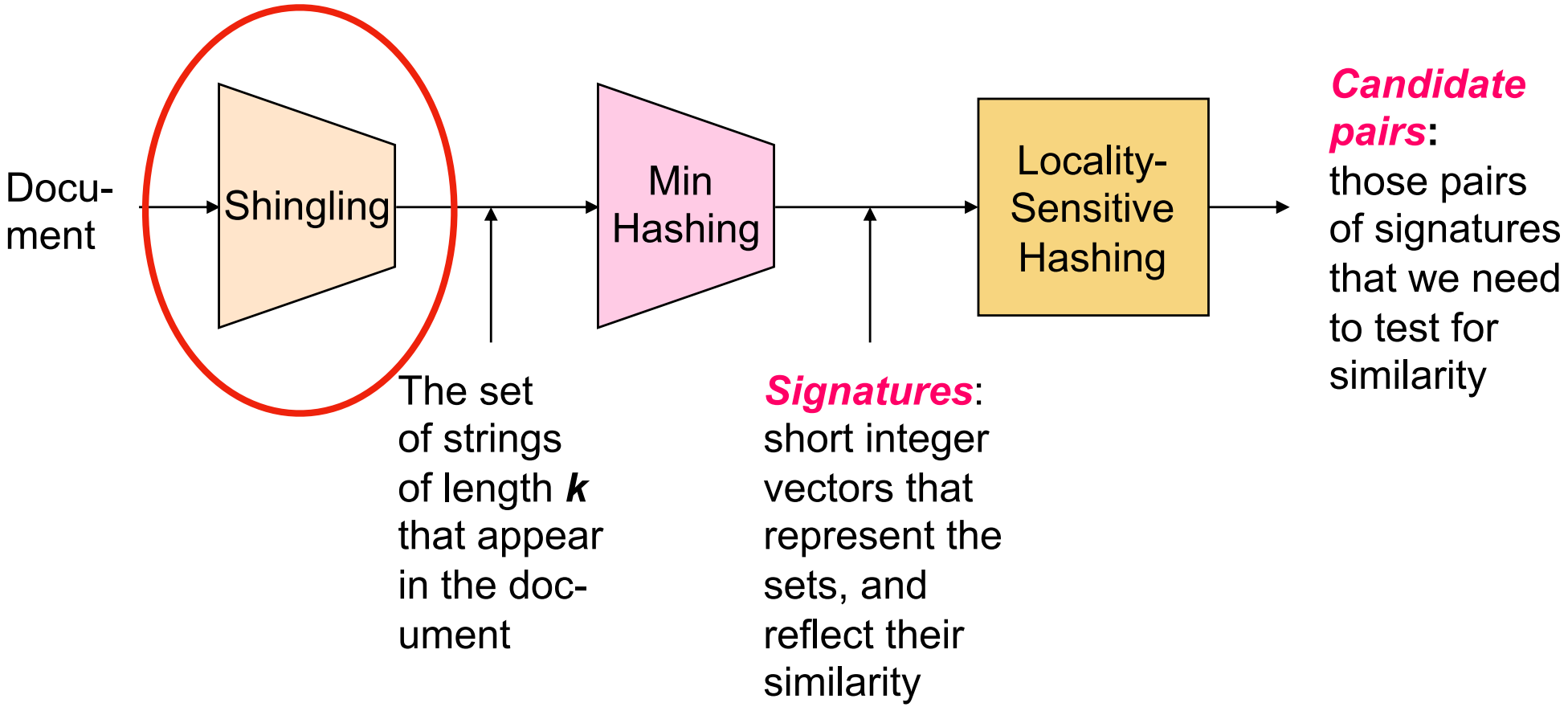
Idea of Hashing

- To efficiently compare long documents, we need to
  - quickly eliminate unlikely pairs
  - hash pairs that are likely to be similar into the same bucket
  - represent the documents by their short signatures
  - only compare signatures in the same bucket

Multiple Hashing

- Focus on pairs that are likely to be similar
- Hash items using many different hash functions
  - similar pairs wind up in the same bucket
  - examine only the candidate pairs
- **Example:** finding similar documents
  - **Shingling:** convert documents into sets
  - **Min-Hashing:** Convert large sets to short signatures, while preserving similarity
  - **Locality-Sensitive Hashing**

Locality-Sensitive Hashing



- What is shingling?

Shingling

- Convert documents to sets
- Simple approaches:
  - Document = set of words appearing in document
  - Document = set of `important' words
  - Don't work well. Why?
- Need to account for **ordering** of words!
- 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 else. E.g., k=2; document D1 = abcab;

    Set of 2-shingles: S(D1) = {ab, bc, ca}

Compressing Shingles

- To compress long shingles, we can hash them to a few bytes
- Represent a document by the set of **hash values** of its k-shingles
  - E.g., k=2; document D1 = abcab
    - set of 2-shingles: S(D1) = {ab, bc, ca}
    - hash the singles: h(D1) = {1, 5, 7}
    - document D1 is a set of its k-shingles C1 = S(D1)

The Choice of Shingles

- Construct the set of 9-shingles for a document and map each of those 9-shingles to a bucket number (4 bytes = 32 bits) in [0, 2^{32}-1]
  - Why is each shingle represented by 4 bytes instead of 9
  - **save space**!
- Why not use 4-shingles?
  - assume only 20 characters are frequently used, the number of different 4-shingles that are likely to occur is only $(20)^4$ = 160000
  - if use 9-shingles, there are more than 2^{32} likely shingles
- Choice of k:
  - k = 5 is OK for short documents
  - k = 10 is better for long documents

Shingles

- Each document is a 0/1 vector in the space of k-shingles
  - Each unique shingle is a dimension
  - Vectors are very sparse
- Documents that have lots of shingles in common have similar text, even if the text appears in different order

Shingles Built from Words

- Find web **pages** that had the same **articles**, regardless of the surrounding elements (e.g., advertisement)
- Need to bias the set of shingles in favor of the article
  - e.g., pages with the same article but different surrounding material have a higher similarity
- News articles, prose have many stop words: “and,” “you,” “to,”…
  - Not contribute to the topic
  - But useful to distinguish from the surrounding elements
- Define a shingle to be a stop word followed by the next two words

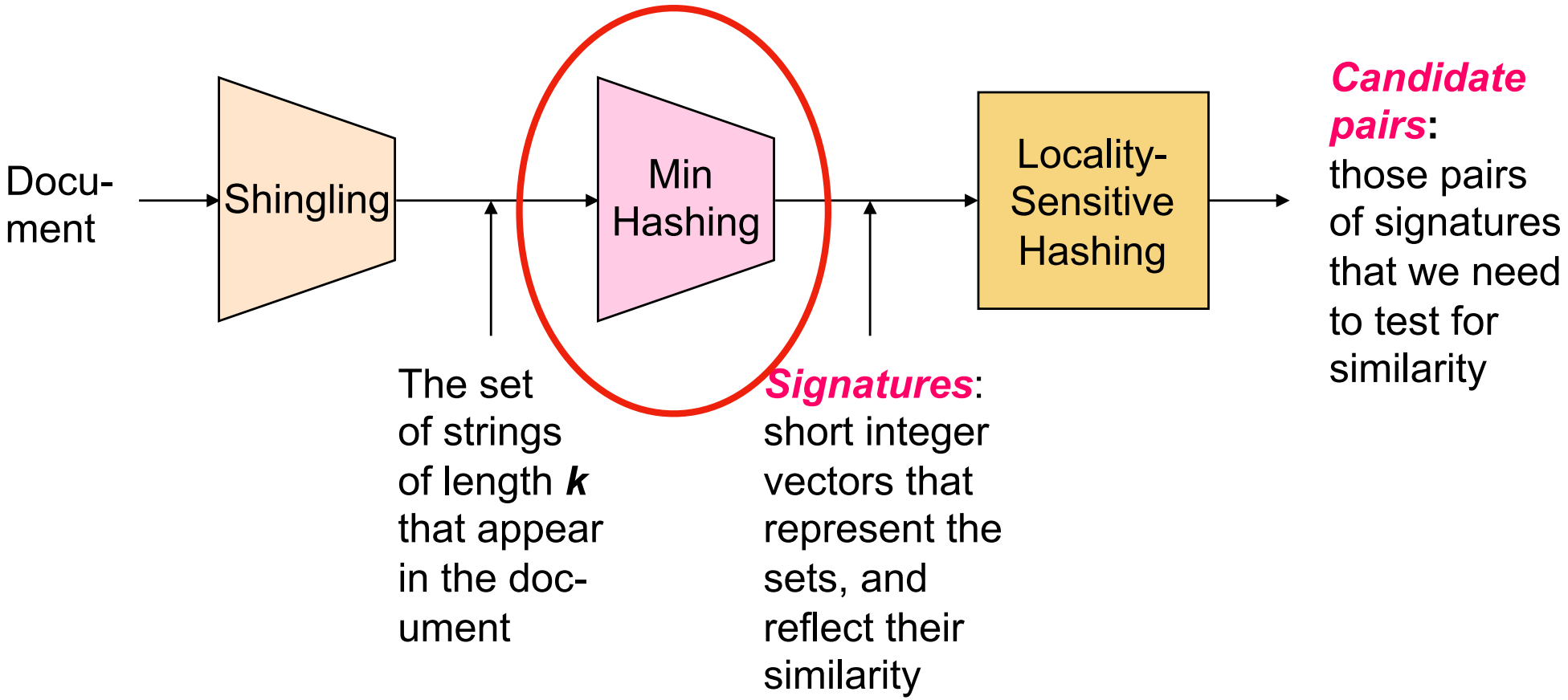
Example

- An ad might have the simple text “Get Vaccinated.” However, a news article with the same idea might read something like “A spokesperson for WHO says today that studies have shown it is important for people to get vaccinated.” We have underscored the stop words. Please list the shingles in the ad and the news article.

- Ad: None
- News article: A spokesperson for, for WHO says, that studies have, have shown it, it is important, ….

- Suppose we need to find near-duplicate documents among N = 1 million documents
  - We have turned documents into sets of shingles
  - Need to compute pairwise similarities for every pair of docs
  - how to compare a pair of set?

Locality-Sensitive Hashing



- How do we turn sets into signatures while preserving similarity?

Motivation for Minhash

- Suppose we need to find near-duplicate documents among N = 1 million documents
  - compute pairwise 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. **Too long**!
- Formulate the problem as **finding subsets that have significant intersection**
  - encode sets using 0/1 (bit, boolean) vectors
  - interpret set intersection as **bitwise AND**, and set union as **bitwise OR**
  - e.g., C1=10111, C2=10011; d(C1, C2)=?

Boolean Matrices

- Rows = elements (singles)
- Columns = sets (documents)
  - 1 in row **i** and column **j** if and only if **i** is a member of **j**
  - column similarity is the **Jaccard similarity** of the corresponding sets (rows with value 1)
  - sparse matrix!
- E.g., sim(C1, C2) = ?

Documents

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

C1 **C2**

Finding Similar Columns

- So far:
  - documents -> sets of shingles
  - represent sets as boolean vectors in a matrix
- **Goal:** finding similar columns while computing small signatures
  - Naive approach:
    - **signatures of columns:** small summaries of columns
    - **relate similarities of signatures and columns:** examine pairs of signatures to find similar columns
  - Comparing all pairs may take too much time!

Hashing Columns (Signatures)

- Idea: `hash' each column C to a small **signature** h(C) s.t.:
  - h(C) is small enough that the signature fits in memory
  - sim(C1, C2) is the same as the `similarity' of signatures h(C1) and h(C2)
- Goal: **find a hash function h(·)** s.t.:
  - if sim(C1, C2) is high, then with high prob. h(C1) = h(C2)
  - if sim(C1, C2) is low, then with high prob. h(C1) ≠ h(C2)
- Hash docs into **buckets**. Expect that `most' pairs of near duplicate docs hash into the same bucket!

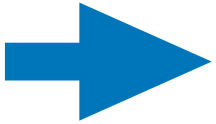
Min-Hashing

- Clearly, the hash function depends on the similarity metric: **not all similarity metrics have a suitable hash function**
- For **Jaccard similarity**, we have a suitable hash function:
- **Min-hashing**
  - rows of the boolean matrix permuted under random permutation p
  - Define a **"hash" function $h_p(C)$** = the index of the **first** (in the permuted order p) row in which column ***C*** has value **1**:
  - $h_p(C) = \min_p p(C)$
  - Use several (e.g., 100) independent hash functions (that is, permutations) to create a signature of a column

Min-Hashing

- To minhash a set represented by a matrix:
  - pick a permutation of rows p
  - compute the minhash value of any column as **the number of the first row**, in order p, in which the column **has a 1**

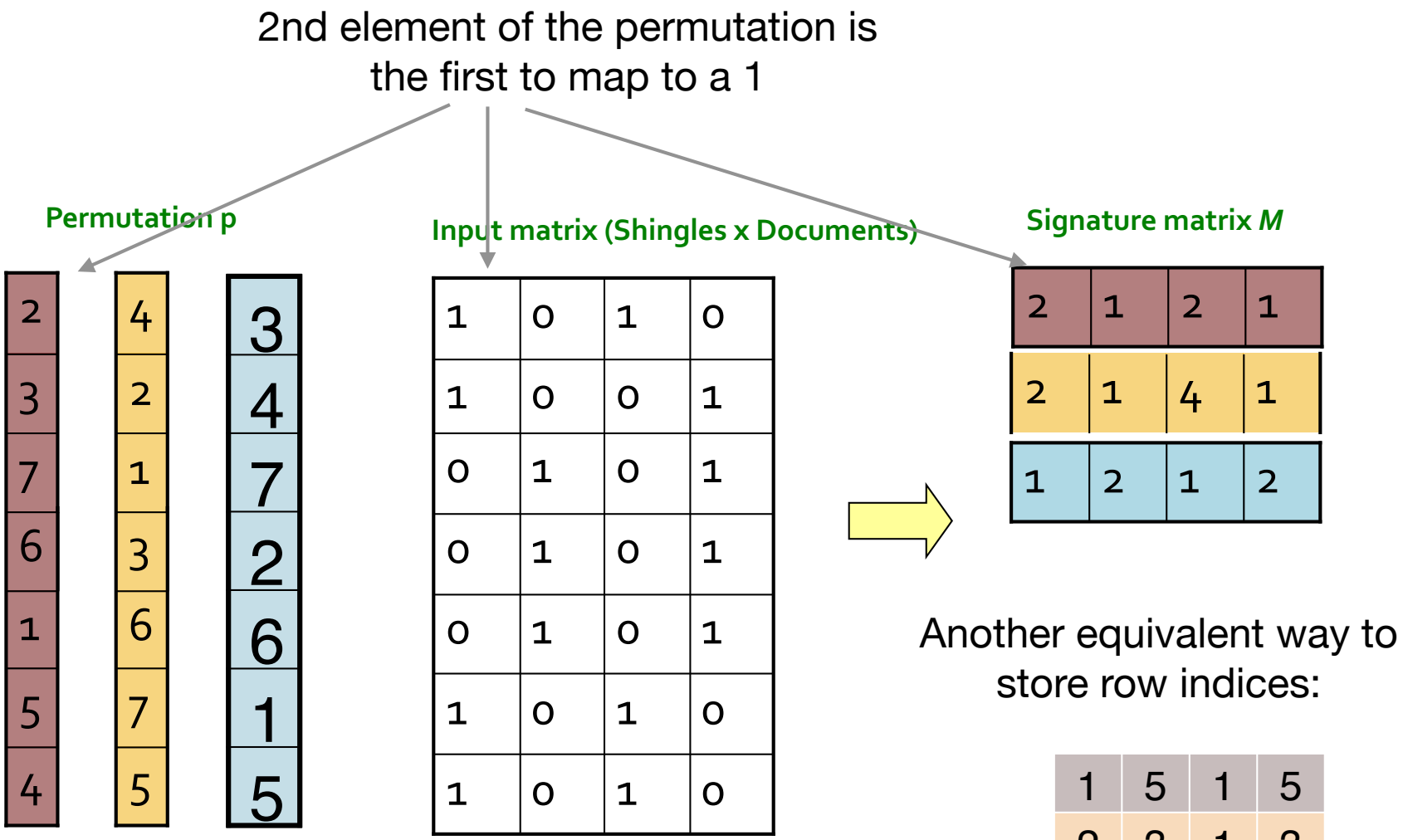
Element	S_1	S_2	S_3	S_4
a	1	0	0	1
b	0	0	1	0
c	0	1	0	1
d	1	0	1	1
e	0	0	1	0



Element	S_1	S_2	S_3	S_4
b	0	0	1	0
e	0	0	1	0
a	1	0	0	1
d	1	0	1	1
c	0	1	0	1

p：beadc　　**h(S1) = a，　h(S2) = c**

Min-Hashing Example



Another equivalent way to store row indices:

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

look up the matrix in the order of "1, 2, 3…" by p

The Minhash Property

- Choose a random permutation p
- Claim: Pr[$h_p(C_1) = h_p(C_2)$] = $sim(C_1, C_2)$. Why?
  - Let **X** be a doc (set of shingles), ***y*** ∈ ***X*** is a shingle
  - Then: Pr[p(y) = min(p(X))] = 1/|X|
    - It is equally likely that any ***y*** ∈ ***X*** is mapped to the ***min*** element
  - Let ***y*** be s.t. p(y) = min(p($C_1 \cup C_2$)). Then either:
    - p(y) = min(p(C_1)) if y ∈ C_1 , or
    - p(y) = min(p(C_2)) if y ∈ C_2
  - So the prob. that **both** are true is the prob. ***y*** ∈ $C_1 \cap C_2$
  - Pr[min(p(C_1))=min(p(C_2))]=$|C_1 \cap C_2|/|C_1 \cup C_2|$= $sim(C_1, C_2)$

Pr: probability
p: permutation

0	0
0	0
1	**1**
0	0
0	1
1	0

one of the two columns had to have 1 at position y

Four Types of Rows

- Given cols C_1 and C_2, rows may be classified as:

	C1	C2
A	1	1
B	1	0
C	0	1
D	0	0

- If it's a type-A row, then $h(C_1) = h(C_2)$
- If a type-B or type-C row, then not
- a = # rows of type A, etc.
  - $sim(C_1, C_2) = a/(a + b + c)$
  - $Pr[h(C_1) = h(C_2)] = Sim(C_1, C_2)$
- Look down the cols C_1 and C_2 until we see a 1

Similarity for Signatures

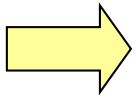
- We have Pr[$h_p(C_1) = h_p(C_2)$] = $sim(C_1, C_2)$. Now generalize to **multiple** hash functions
- The **similarity of two signatures** is the **fraction of the hash functions** in which they agree
- Because of Min-Hash property, the similarity of columns is the same as the expected similarity of their signatures. What are Jaccard similarities and signature similarities for 1 vs 3, 2 vs 4, 1 vs 2, 3 vs 4?

Permutation p

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

Minhash Signatures

- We pick at random some number n of permutations of the rows of M (perhaps 100 permutations): h_1, h_2, …, h_n
- Construct the **minhash signature** for S: [$h_1(S)$, $h_2(S)$, …, $h_n(S)$]
- Form a **signature matrix**
  - has the same number of columns as M but only n rows
  - much smaller than M!
  - the expected number of rows in which two columns agree equals the Jaccard similarity of the corresponding sets
  - more rows in the signature matrix, the smaller the error of estimating the Jaccard similarity

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
  - **sig(C)[i] = min (p_i(C))**
  - the signature of document C is small ~100 bytes!
- We **compressed** long bit vectors into short signatures!
- Pick a random permutation of millions or billions of rows is **time-consuming**!
- Simulate the effect of a random permutation by **a random hash function** that maps row numbers to buckets

Implementation

- **Row hashing** instead of permutation!
  - pick k = 100 hash functions k_i
  - ordering under k_i gives a random row permutation
  - for each column **C** and hash-func. k_i, keep a "slot" for the min-hash value
  - initialize all **sig(C)[i] =** ∞
  - scan rows looking for 1s
    - suppose row ***j*** has 1 in column ***C***
    - 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 are random integers
p is a prime number (p > N)

Example

- We consider two hash functions for the following matrix. Please estimate the Jaccard similarities of S_1 vs S_2, S_1 vs S_3, and S_1 vs S_4 according to the signature matrix.

Row	S_1	S_2	S_3	S_4	$x+1 \mod 5$	$3x+1 \mod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

Example

Row	S_1	S_2	S_3	S_4	$x+1 \mod 5$	$3x+1 \mod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

	S_1	S_2	S_3	S_4
h_1	∞	∞	∞	∞
h_2	∞	∞	∞	∞

	S_1	S_2	S_3	S_4
h_1	1	∞	∞	1
h_2	1	∞	∞	1

	S_1	S_2	S_3	S_4
h_1	1	∞	2	1
h_2	1	∞	4	1

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	1	2	4	1

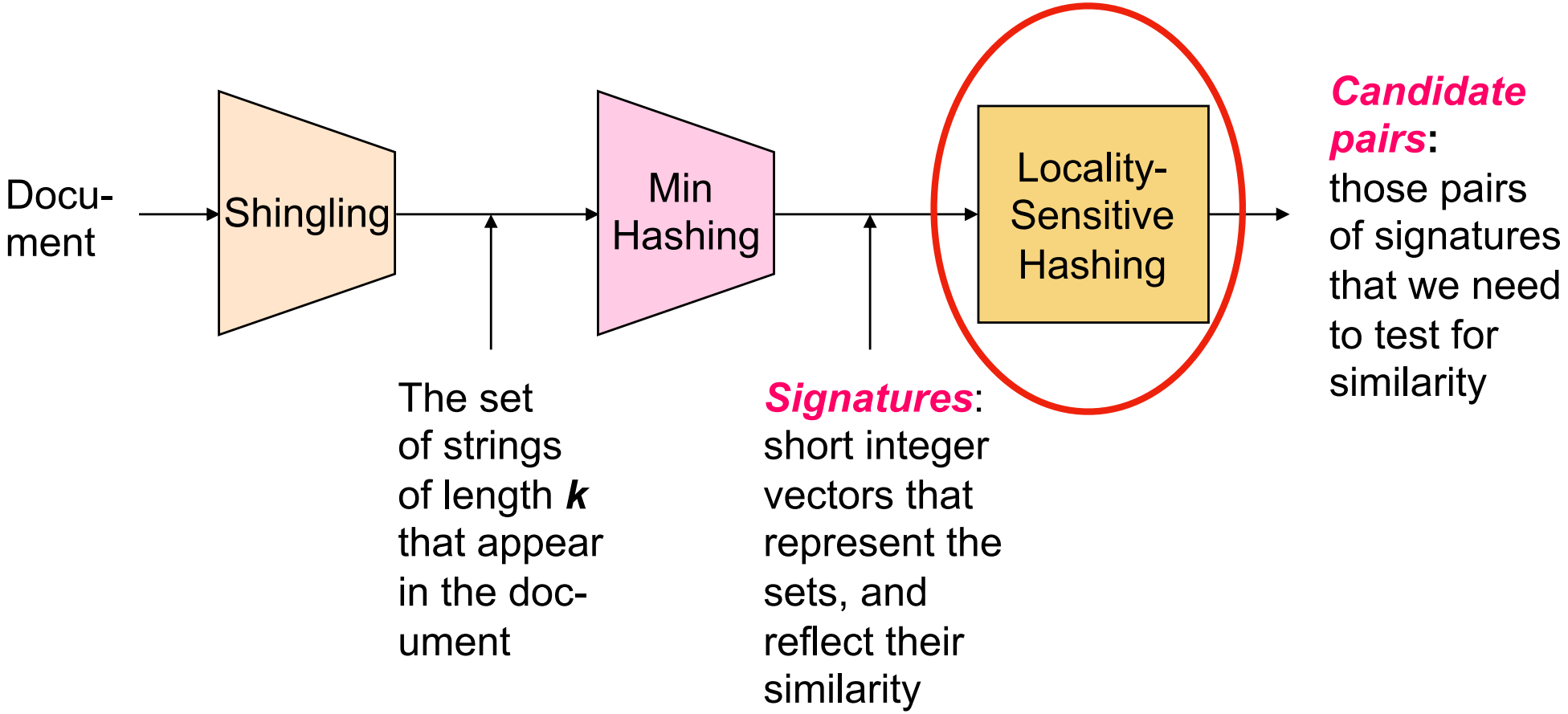
	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	0	2	0	0

	S_1	S_2	S_3	S_4
h_1	1	3	0	1
h_2	0	2	0	0

sim(S1,S2) = 0
sim(S1,S3) = 1/2
sim(S1,S4) = 1

In fact,
sim(S1,S2) = 0
sim(S1,S3) = 1/4
sim(S1,S4) = 2/3

Locality-Sensitive Hashing



- How do we retain pairs likely to be similar?

Locality-Sensitive Hashing

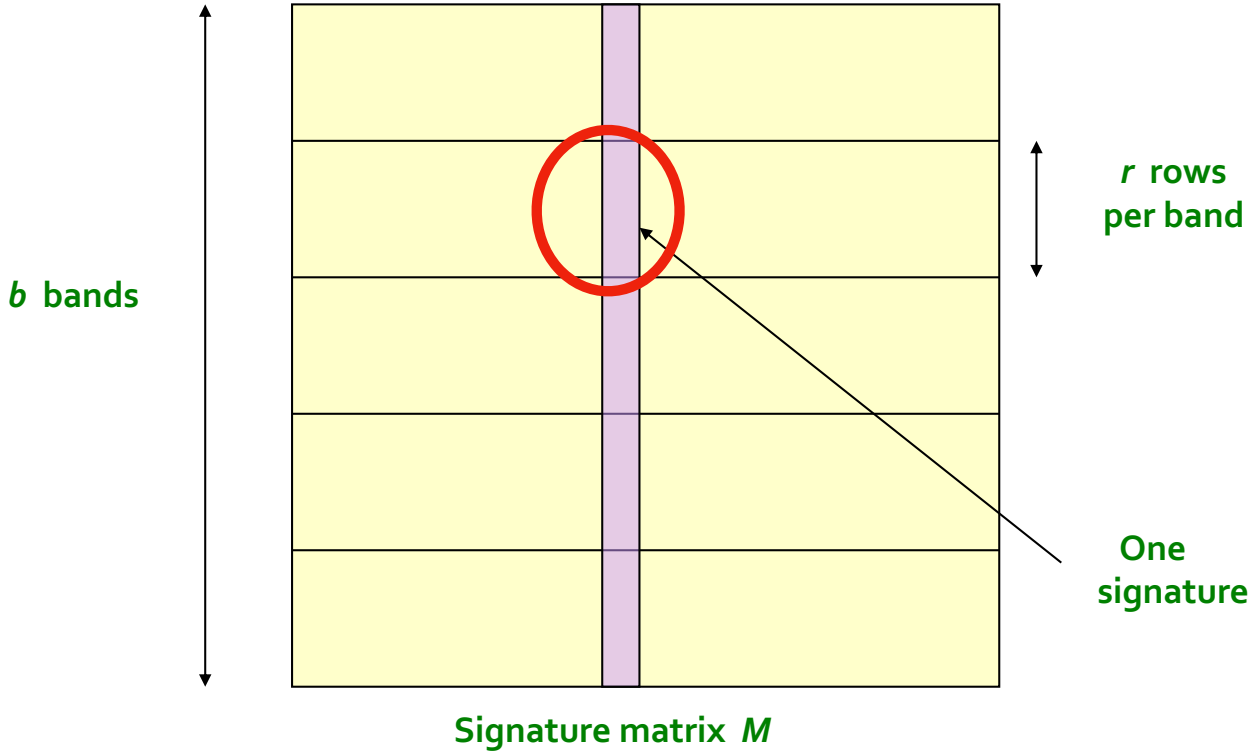
- Suppose we have 1,000,000 documents, and we use signatures of length 250. How many bytes do we need for the signature of one document?
  - 1000 bytes
- If we use 1000 bytes per document for the signatures, how many bytes in total do we need to store all signature matrices?
  - 10^9 bytes (1 GB) are needed! How many pairs of documents we need to compare? $C^2_{1,000,000}$ **Too many! Can we do better?**
- We only need to focus on pairs that are likely to be similar, without investigating every pair
- Consider any pair that hashed to the same bucket to be a **candidate pair**

Locality-Sensitive Hashing

- **Goal**: Find documents with Jaccard similarity at least **s** (e.g. **s**=0.8)
- **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 Min-Hash matrices:
  - Hash columns of signature matrix **M** to many buckets. Each pair of documents that hashes into the same bucket is a **candidate pair**
  - 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 fraction **s** values of **i**

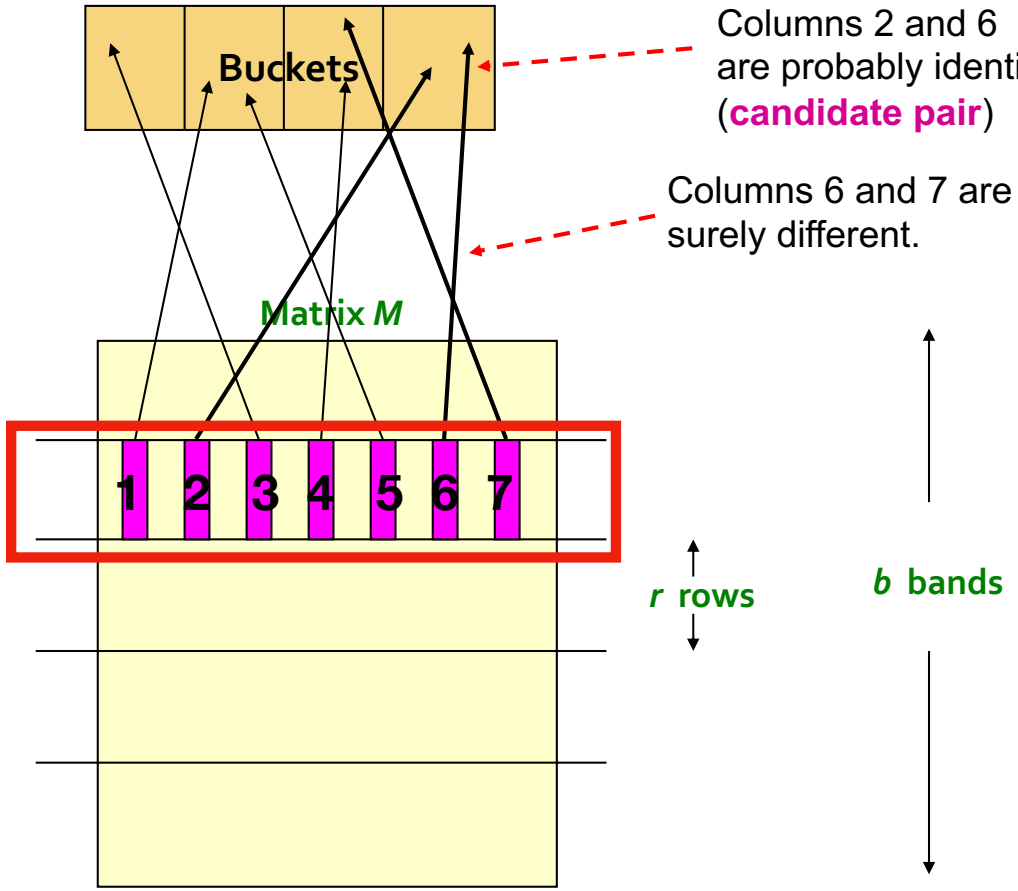
LSH for Min-Hash

- Idea: Hash columns of signature matrix **M** several times
- Arrange that **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 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
- **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



Assume: there are enough buckets that columns are unlikely to hash to the same bucket unless they are identical in a band; "same bucket" means "identical in that band"

Examples

- Suppose
  - 100,000 columns of **M** (100k docs)
  - Signatures of 100 integers (rows), 4 bytes x 100 x 100,000 = 4 x 10^7 bytes = 40Mb in total
  - Choose **b**=20 bands for **r**=100 / 20 = 5 integers/band
  - Goal: Find pairs of documents that are at least **s**=0.8 similar
- If sim(C1, C2)>=**s**, we want C1, C2 to be a candidate pair — want to hash them to **at least 1 common bucket** (at least 1 band identical)
- Prob. C1, C2 identical in one particular band? $(0.8)^5=0.328$
- Prob. C1, C2 are not similar in all of the 20 bands? $(1-0.328)^{20}=0.00035$
  - about 1/3000th of the 80%-similar column pairs are **false negatives**
  - we would find 99.965% pairs of truly similar documents

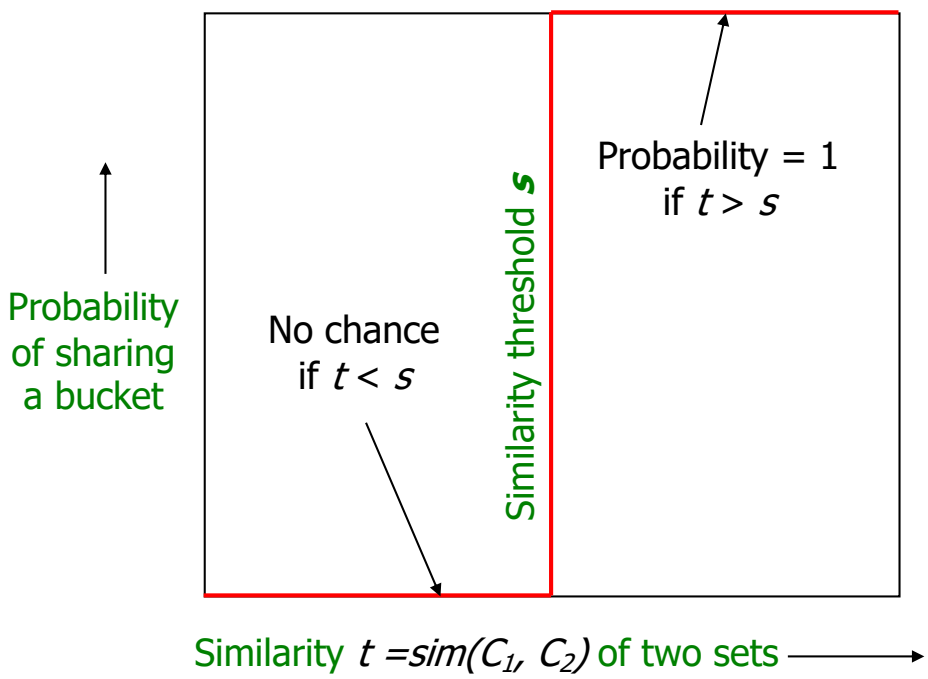
Examples

- Suppose
  - 100,000 columns of **M** (100k docs)
  - Signatures of 100 integers (rows), 40Mb in total
  - Choose **b**=20 bands for **r**=5 integers/band
  - Goal: Find pairs of documents that are at least **s**=0.3 similar
- If sim(C1, C2)<**s**, we want C1, C2 to hash to **NO common buckets** (all bands should be different)
- Prob. C1, C2 identical in one particular band? $(0.3)^5=0.00243$
- Prob. C1, C2 are identical in at least 1 of 20 bands? $1-(1-0.00243)^{20}=0.0474$
  - about 4.74% pairs of docs with similarity 30% end up becoming candidate pairs (**false positives**). We will have to examine them but their similarity <**s**

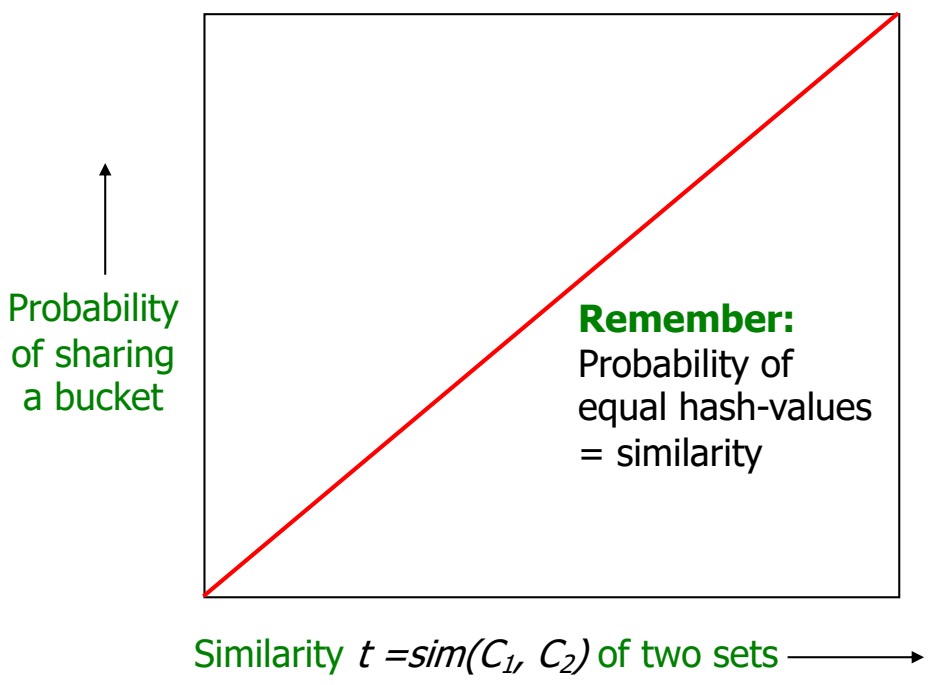
Tradeoff

- To balance false positives/negatives, we should pick:
  - The number of Min-Hashes (rows of **M**)
  - The number of bands **b**
  - The number of rows **r** per band

Ideal:

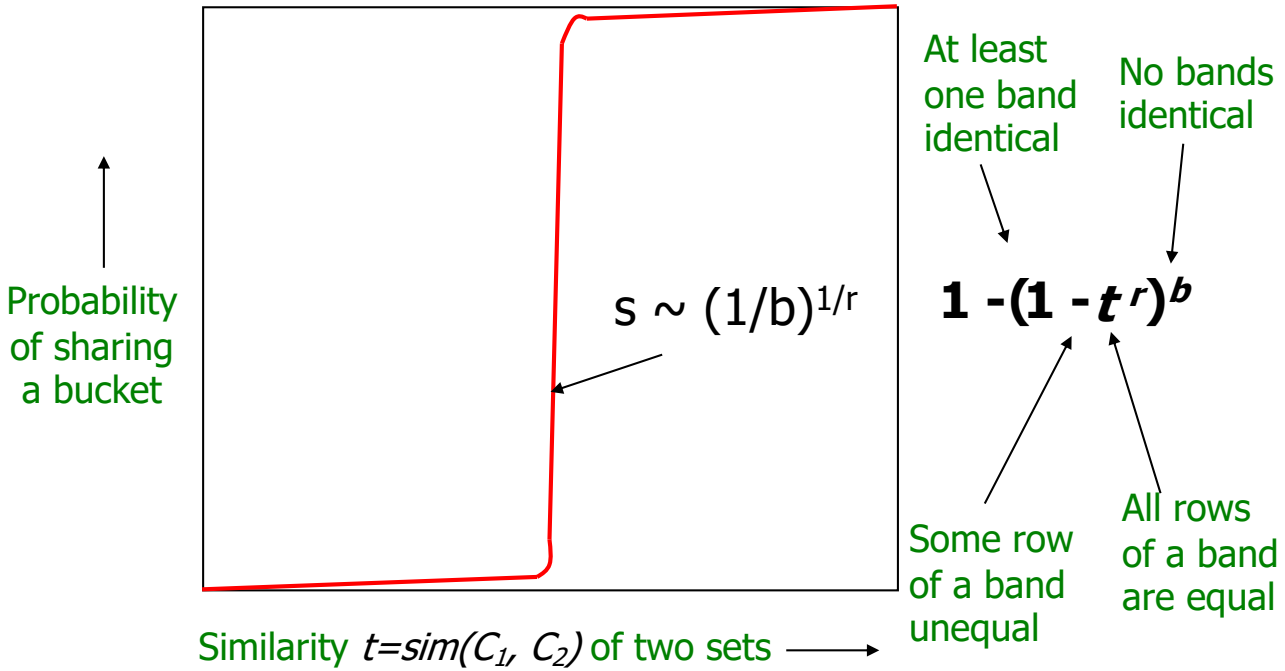


1 Band per Row:



Choices of b & r

- Columns C1 and C2 have similarity **t**
- Pick any band (**r** rows)
  - Prob. that all rows in band equal = $\mathbf{t^r}$
  - Prob. that some row in band unequal = $\mathbf{1 - t^r}$
- Prob. that no band identical = $\mathbf{(1 - t^r)^b}$
- Prob. that at least 1 band identical = $\mathbf{1 - (1 - t^r)^b}$

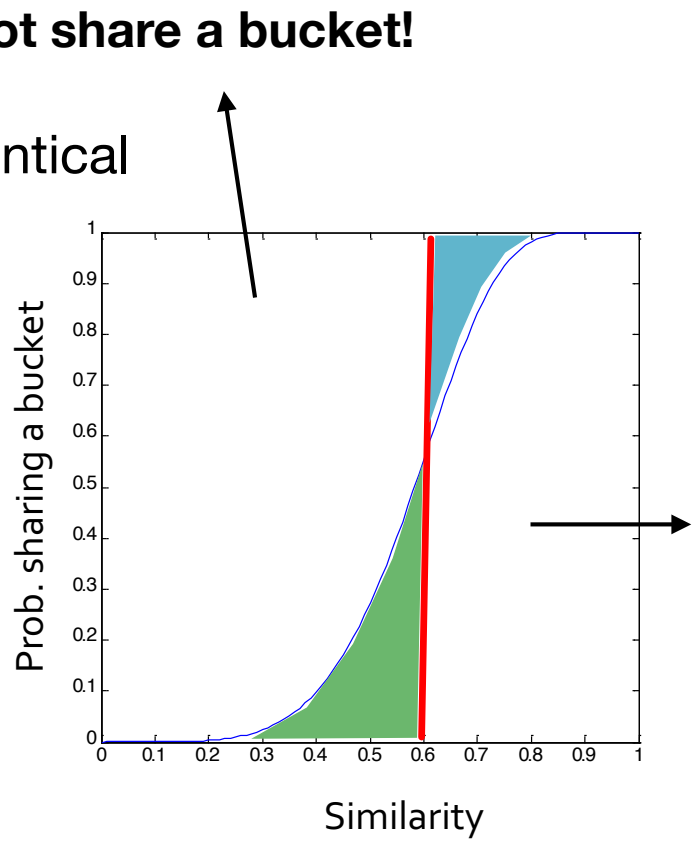


Choices of b & r

- 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**

Prob. that at least 1 band is identical

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

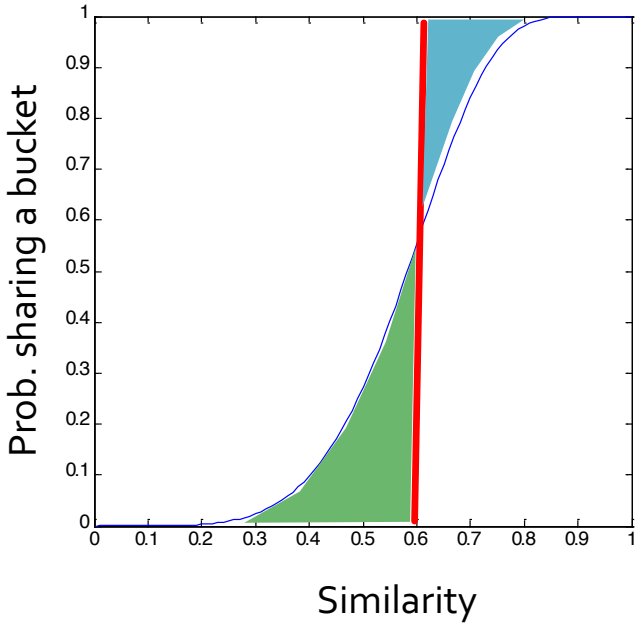


Combining the Techniques

- Find the set of candidate pairs for similar documents and then discover the truly similar documents among them

    1. Pick a value of k and construct from each document the set of **k-shingles**
    2. Sort the document-shingle pairs to order them by shingle
    3. Pick a length n for the minhash signatures. Compute the **minhash signatures** for all the documents (sorted lists)
    4. Choose a threshold **t** that defines how similar documents have to be in order for them to be regarded as “similar pair.” Pick a number of bands **b** and a number of rows **r** such that **br = n**, and the threshold **t** is approximately $\mathbf{(1/b)^{1/r}}$.
    5. Construct candidate pairs by applying the **LSH** technique
    6. Examine each candidate pair’s signatures and determine whether the fraction of components in which they agree is at least **t**

- Can we do better with regard to FN and FP?



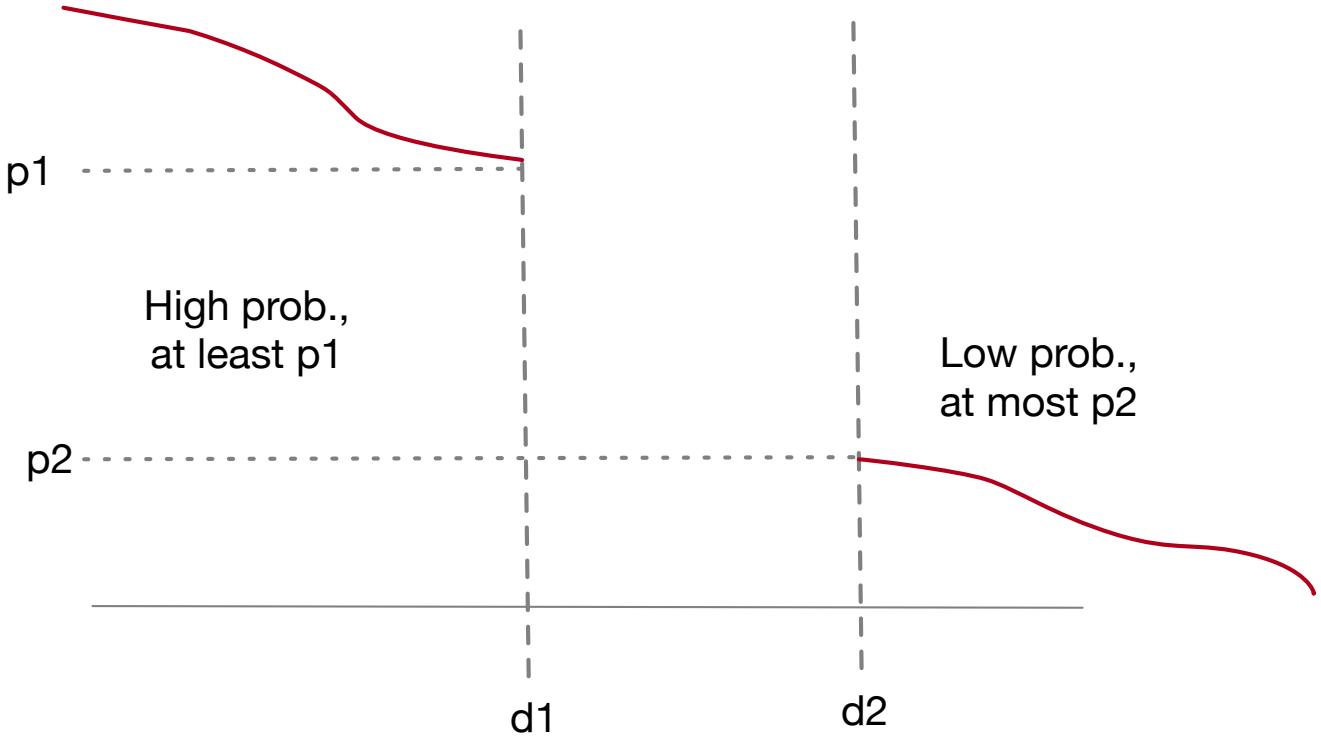
50 hash functions
(b=10, r=5)

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

Hash Functions Decide Equality

- A hash function h takes two elements x and y, and returns a decision whether x and y are candidates for comparison
  - E.g., the family of minhash functions computes minhash values and returns yes if they are the same
  - Shorthand: h(x) = h(y) means h returns yes for the pair of x and y
- Suppose we have a space S of points with a distance measure d
- A family **H** of hash functions is said to be (d1, d2, p1, p2)-sensitive if for any x and y in S:
  - if d(x, y) <= d1, prob. over all h in **H** that h(x)=h(y) is at least p1
  - if d(x, y) >= d2, prob. over all h in **H** that h(x)=h(y) is at most p2

Locality-Sensitive Families (LSF)



Expect the distance between d1 and d2 very small,
the distance between p1 and p2 very large

LSF for Jaccard Distance

- S = sets, d = Jaccard distance, **H** is formed from minhash functions for all permutations
- Pr(h(x)=h(y)) = 1 - d(x, y) = sim(x, y)
- **H** is a (1/3, 2/3, 2/3, 1/3)-sensitive family for S and d

if distance <=1/3 (similarity >= 2/3)

Prob. that minhash values agree >= 2/3)

For Jaccard similarity, minhashing gives us a (d1, d2, (1-d1), (1-d2))-sensitive family for any d1 < d2

Steepen the S-curve with “bands” technique!
AND construction like “rows in a band.”
OR construction like “many bands.”

AND/OR of Hash Functions

- Given family **H**, construct family **H'** whose members each consist of r functions from **H**

r rows in a single band!

  - AND: For $h = \{h_1, \ldots, h_r\}$ in **H'**, $h(x) = h(y)$ iff $h_i(x) = h_i(y)$ for all i
  - Theorem: if **H** is (d1, d2, p1, p2)-sensitive, then **H'** is (d1, d2, $(p1)^r$, $(p2)^r$)-sensitive.

each member of H is independently chosen

  - OR: For $h = \{h_1, \ldots, h_b\}$ in **H'**, $h(x) = h(y)$ iff $h_i(x) = h_i(y)$ for some i

combining b bands

  - Theorem: if **H** is (d1, d2, p1, p2)-sensitive, then **H'** is (d1, d2, $1-(1-p1)^b$, $1-(1-p2)^b$)-sensitive.

at least one band say yes

  - AND makes all prob. shrink, but by choosing r correctly, we can make the lower prob. (p2) approach 0 while the higher (p1) does not
  - OR makes all prob. grow, but by choosing b correctly, we can make the upper prob. (p1) approach 1 while the lower (p2) does not

Composing Constructions

- As for the signature matrix, we can use the AND construction followed by the OR construction
  - or vice-versa
  - or any sequence of AND's and OR's alternating
- AND-OR Composition: Each of the two prob. p is transformed into $1-(1-p^r)^b$
  - "S-curve"
  - E.g., r=4, b=4

p	$1-(1-p^4)^4$
0.2	0.0064
0.3	0.032
0.4	0.0985
0.5	0.2275
0.6	0.426
0.7	0.6666
0.8	0.8785
0.9	0.986

E.g., Transforms a (0.2,0.8,0.8,0.2)-sensitive family into a (0.2,0.8,0.8785,0.0064)-sensitive family

Composing Constructions

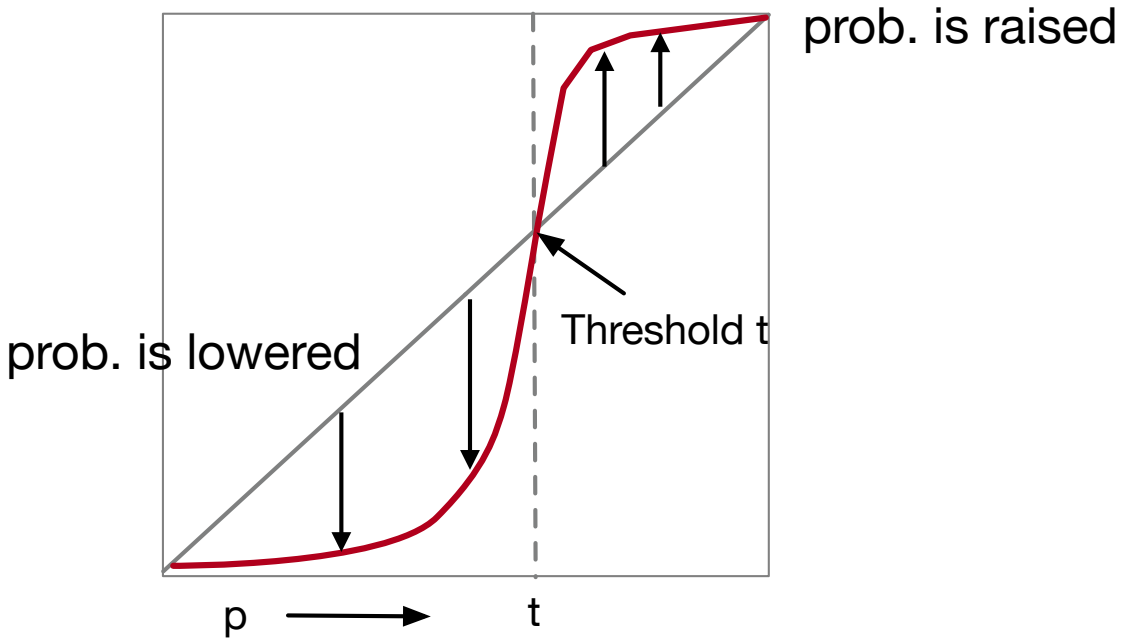
- OR-AND Composition: Each of the two prob. p is transformed into $(1-(1-p)^b)^r$
  - the same S-curve, mirrored horizontally and vertically
  - E.g., b=4, r=4

p	$(1-(1-p)^4)^4$
0.1	0.014
0.2	0.1215
0.3	0.3334
0.4	0.574
0.5	0.7725
0.6	0.9015
0.7	0.9680
0.8	0.9936

E.g., Transforms a (0.2,0.8,0.8,0.2)-sensitive family into a (0.2,0.8,0.9936,0.1215)-sensitive family

General Use of S-Curves

- For each S-curve $1-(1-p^r)^b$, there is a threshold t, for which $1-(1-t^r)^b=t$
- Above t, prob. are increased, below t, they are decreased
- You improve the sensitivity (by AND-OR construction) as long as the low prob. (p2) is less than t, and the high prob. (p1) is greater than t



Cascading Construction

- E.g., apply the (4,4) OR-AND construction followed by the (4,4) AND-OR construction
  - 256 minhash functions
  - Transforms a (0.2,0.8,0.8,0.2)-sensitive family into a (0.2,0.8,0.9999996,0.0008715)-sensitive family

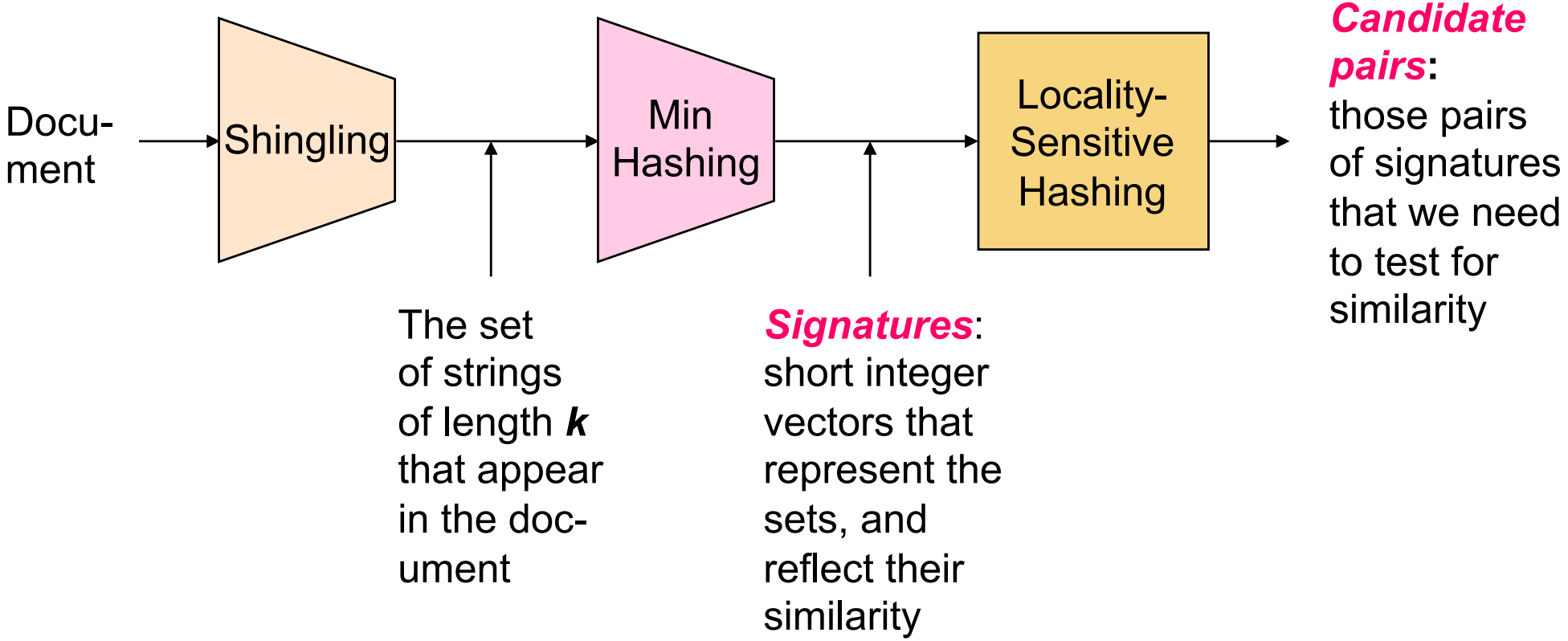
Review—Finding Similar Documents

- **Goal:** Given a large number (N in the millions or billions) of documents, 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'
- **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**

Review—Idea of Hashing

- To efficiently compare long documents, we need to
  - quickly eliminate unlikely pairs
  - hash pairs that are likely to be similar into the same bucket
  - represent the documents by their short signatures
  - only compare signatures in the same bucket

Review—Locality-Sensitive Hashing



Review—Hashing Columns (Signatures)

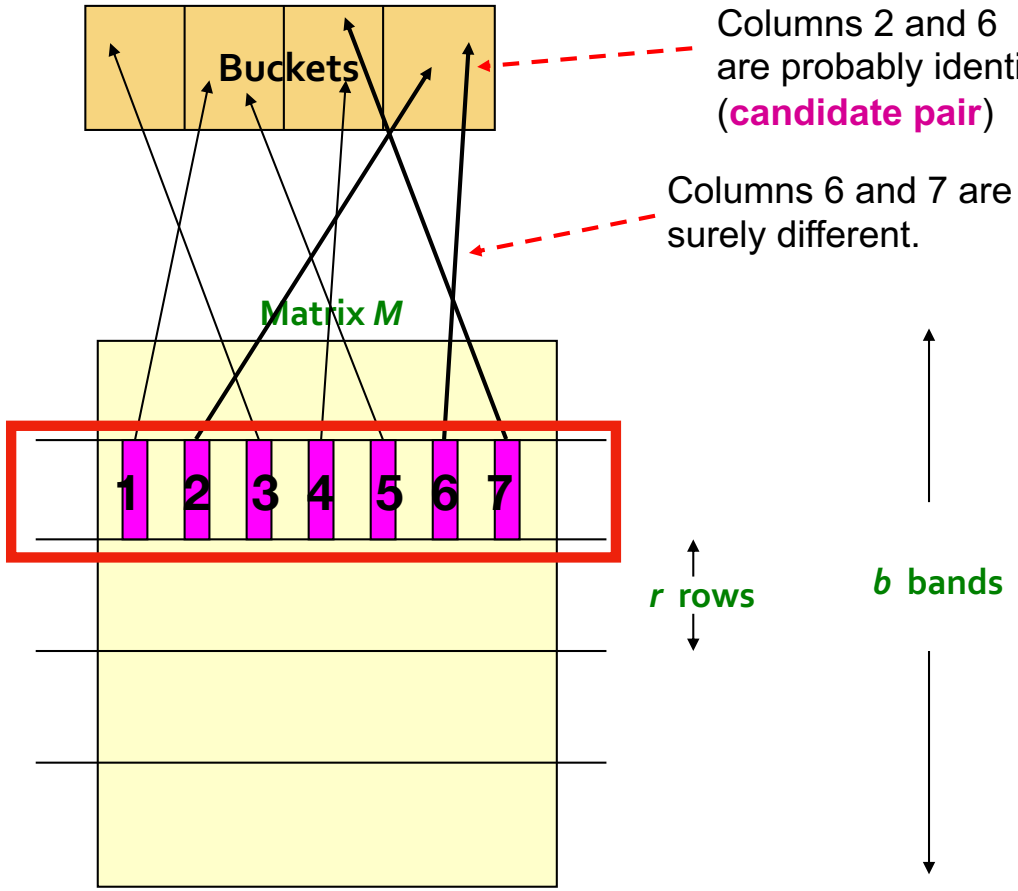
- Idea: `hash' each column C to a small **signature** h(C) s.t.:
  - h(C) is small enough that the signature fits in memory
  - sim(C1, C2) is the same as the `similarity' of signatures h(C1) and h(C2)
- Goal: **find a hash function h(·)** s.t.:
  - if sim(C1, C2) is high, then with high prob. h(C1) = h(C2)
  - if sim(C1, C2) is low, then with high prob. h(C1) ≠ h(C2)
- Hash docs into **buckets**. Expect that `most' pairs of near duplicate docs hash into the same bucket!

Review—Locality-Sensitive Hashing

- **Goal**: Find documents with Jaccard similarity at least **s** (e.g. **s**=0.8)
- **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 Min-Hash matrices:
  - Hash columns of signature matrix **M** to many buckets. Each pair of documents that hashes into the same bucket is a **candidate pair**
  - 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 fraction **s** values of **i**

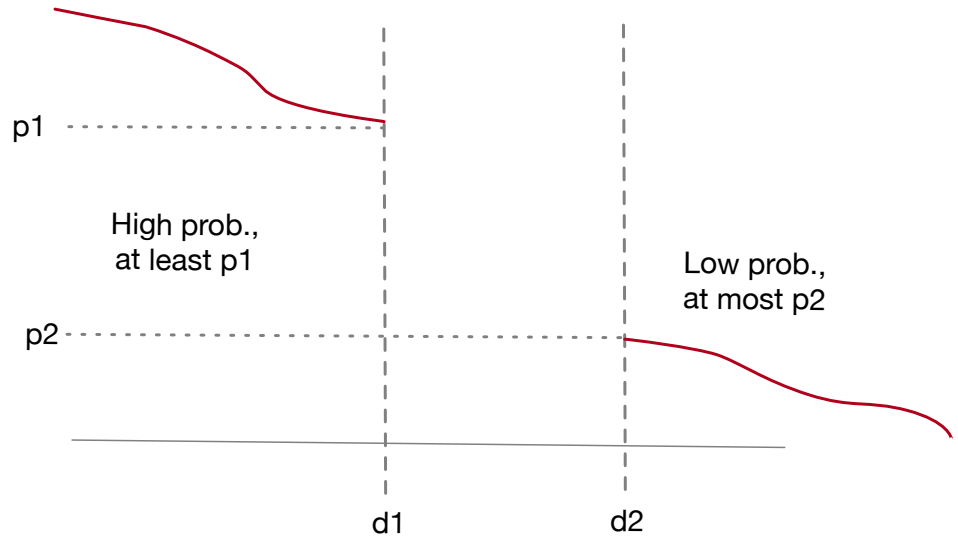
Review—LSH

- 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
- **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



Assume: there are enough buckets that columns are unlikely to hash to the same bucket unless they are identical in a band; "same bucket" means "identical in that band"

Review—Locality-Sensitive Families



Expect the distance between d1 and d2 very small, the distance between p1 and p2 very large

- A family **H** of hash functions is said to be (d1, d2, p1, p2)-sensitive if for any x and y in S:
  - if d(x, y) <= d1, prob. over all h in **H** that h(x)=h(y) is at least p1
  - if d(x, y) >= d2, prob. over all h in **H** that h(x)=h(y) is at most p2

Review—LSF for Jaccard Distance

- S = sets, d = Jaccard distance, **H** is formed from minhash functions for all permutations
- Pr(h(x)=h(y)) = 1 - d(x, y) = sim(x, y)
- **H** is a (1/3, 2/3, 2/3, 1/3)-sensitive family for S and d

if distance <=1/3 (similarity >= 2/3)

Prob. that minhash values agree >= 2/3)

For Jaccard similarity, minhashing gives us a (d1, d2, (1-d1), (1-d2))-sensitive family for any d1 < d2

Steepen the S-curve with “bands” technique!
AND construction like “rows in a band.”
OR construction like “many bands.”

- What are other hash functions except for min hash?

Locality-Sensitive Functions

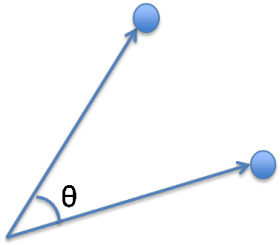
- Locality-Sensitive Hashing (LSH) Families
- LSH Family for Jaccard Distance
- LSH Family for Cosine Distance
- LSH Family for Euclidean Distance

Cosine Distance

- d is a distance measure if it is a function from pairs of points to real numbers s.t.:
  1. d(x, y) >= 0
  2. d(x, y) = 0 iff x = y
  3. d(x, y) = d(y, x)
  4. d(x, y) <= d(x, z) + d(z, y) (triangle inequality)

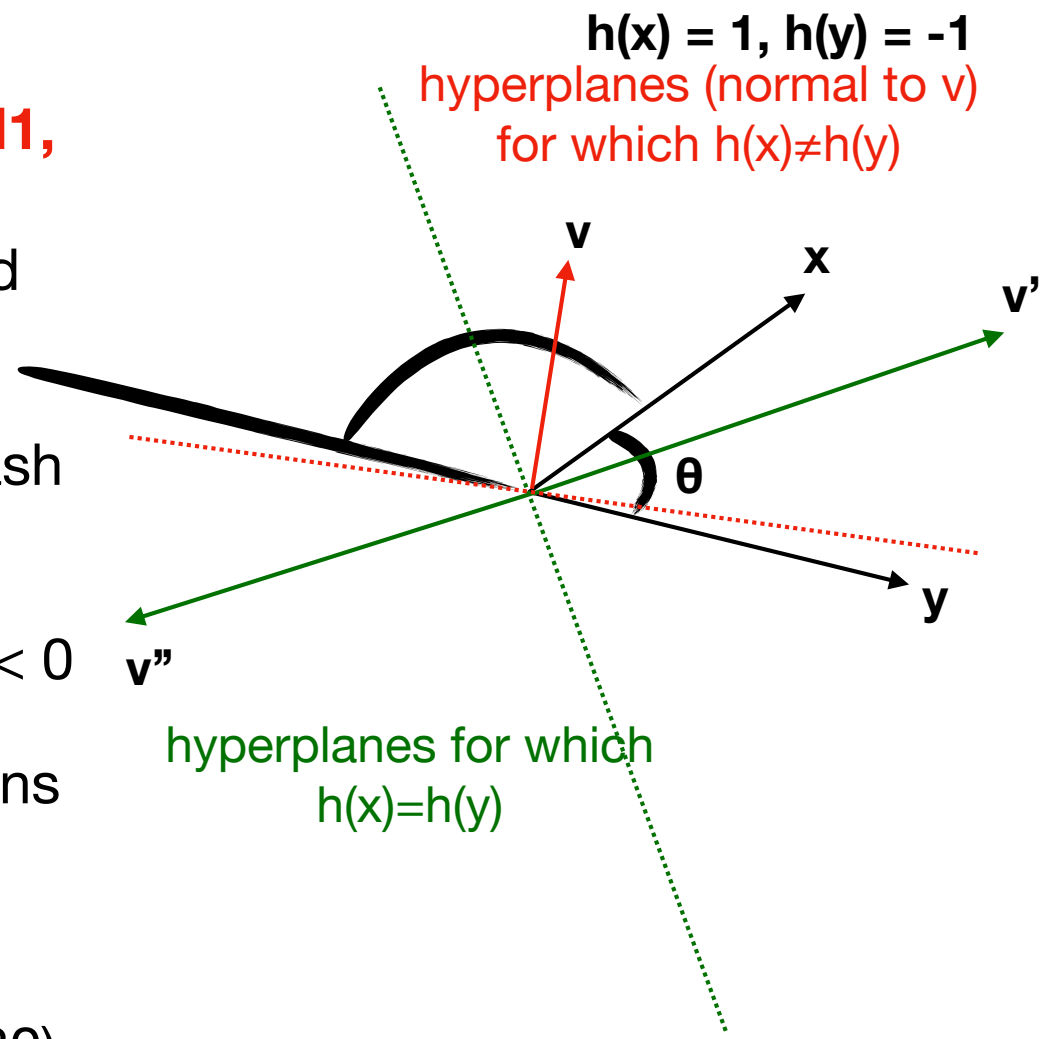
- Cosine distance = angle between vector x and y:
  1. d(x, y) is in the range of 0 to 180
  2. d(x, y) = 0 iff two vectors are the same direction
  3. the angle between x and y is the same as the angle between y and x
  4. one way to rotate from x to y is to rotate to z and then to y. Sum of the two rotations >= rotation directly from x to y

$$sim(A,B) = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$



LSH Family for Cosine Distance

- For cosine distance, there is a technique analogous to minhashing for generating a **(d1, d2, (1-d1/180),(1-d2/180))-sensitive** family for any d1 and d2 — **random hyperplanes**
- Each vector v determines a hash function h_v, with two buckets
- $h_v(x) = +1$ if $v \cdot x > 0$; $= -1$ if $v \cdot x < 0$
- LS-family **H** = set of all functions derived from any vector
- Claim: Pr[h(x)=h(y)]=1 - (angle between x and y divided by 180)



Signatures for Cosine Distance

- Pick some number of vectors, and hash your data for each vector
- The result is a signature (sketch) of +1's and -1's that can be used for LSH like the minhash signatures for Jaccard distance
- The existence of the LSH-family is sufficient for amplification by AND/OR
- We need not pick from all possible vectors v to form a component of a sketch
- It suffices to consider only vectors v consisting of +1 and -1 components

Euclidean vs. Non-Euclidean

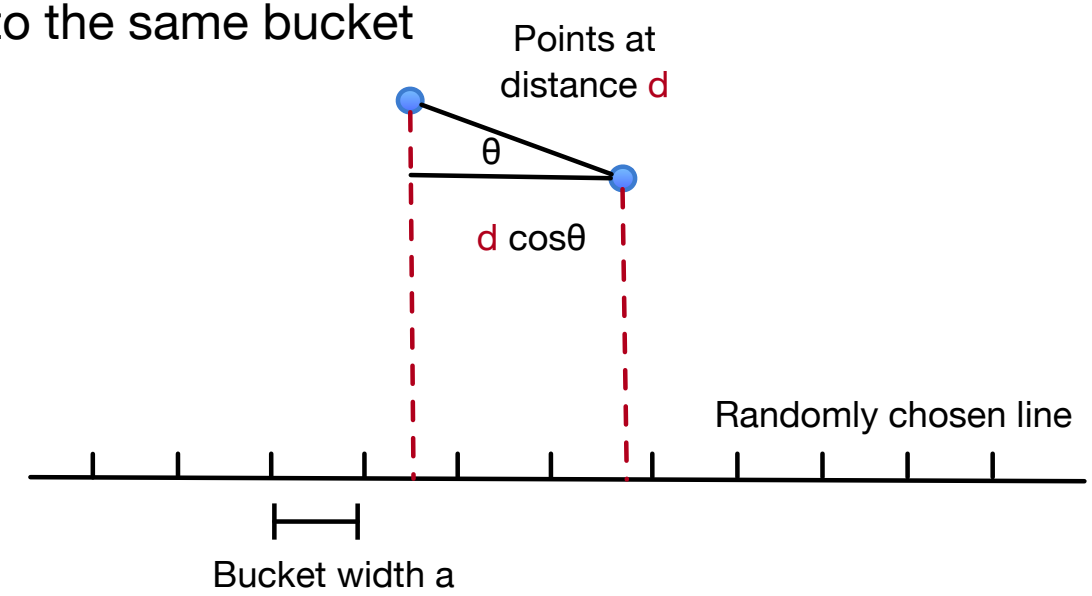
- A **Euclidean space** has some number of real-valued dimensions and “dense” points
  - there is a notion of “average” of two points
  - a Euclidean distance is based on the locations of points in such a space
- Any other space is **Non-Euclidean**
  - distance measures for non-Euclidean spaces are based on properties of points, but not their “location” in a space

LSH Family for Euclidean Distance

- Idea: hash functions correspond to randomly chosen 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

if d >> a, θ must be close to 90° for there to be any chance points go to the same bucket

if d << a, the chance the points are in the same bucket is at least 1 - d/a



LSH Family for Euclidean Distance

- If points are >= 2a apart, 60<=θ<=90 for there to be a chance that the points go in the same bucket
- at most 1/3 prob. that the randomly chosen hash function returns yes
- If points are <= a/2 apart, there is at least 1/2 chance they share a bucket
- Yields a (a/2, 2a, 1/2, 1/3)-sensitive family of hash functions

- What are the applications for locality sensitive hashing?

Applications of LSH

- Entity Resolution
- Matching Fingerprints
- Matching Newspaper Articles

Entity Resolution

- The **entity-resolution** problem is to examine a collection of records and determine which refer to the same entity
  - Entities could be people, events, etc.
  - We want to merge records if their values in corresponding fields are similar
- Company A and B want to find out what customers they share
  - Each company had about 1 million records
  - Records had name, address, and phone. Could be different for the same person

Entity Resolution

- **Step 1:** Design a measure (`score') of how similar records are
  - e.g., deduct points for small misspellings (`Jeffrey' vs. `Jeffery') or same phone with different area code
- **Step 2:** Score all pairs of records that the LSH scheme identified as candidates; report high scores as matches
  - 3 hash functions: exact values of name, address, phone
  - compare iff records are identical in at least one
  - miss similar records with a small differences in all three fields

Entity Resolution

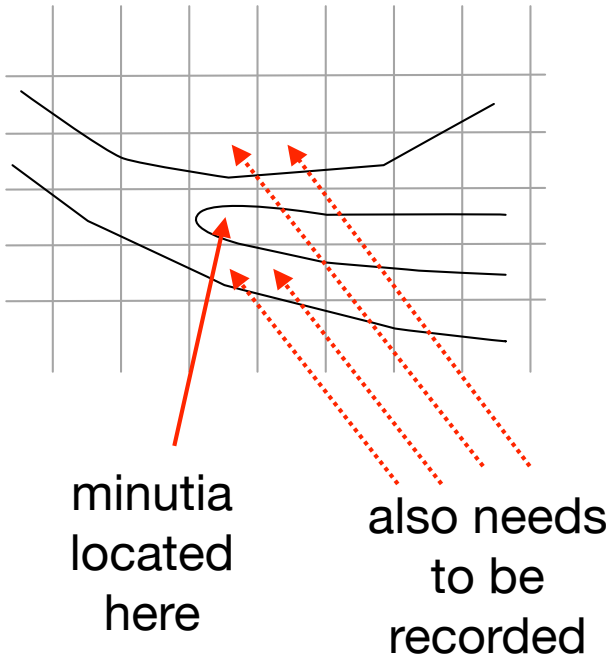
- How do we hash strings such as names so there is one bucket for each string?
  - **Idea:** sort the strings instead
  - Another option was to use a few million buckets, and compare all pairs of records within one bucket
- **Validation** of results
  - identical records has a creation date difference of 10 days
  - only looked for records created within 90 days of each other, so bogus matches has a 30-day average
  - looking at the pool of matches with a fixed score, compute the average time-difference x, fraction (30 - x)/30 of them were valid matches

Validation of Results

- Any field not used in the LSH could be used to **validate**, provided corresponding values were closer for true matches than false
    - e.g., if records has a height field, we would expect true matches to be close, false matches to be the **average difference** for random people

Matching Fingerprints

- Represent a fingerprint by the set of positions of minutiae (features of a fingerprint, e.g., points where two ridges come together or a ridge ends)
- Place a grid on a fingerprint
  - Normalize so identical prints will overlap
- Set of grid squares where minutiae are located represents the fingerprint
  - Possibly, treat minutiae near a grid boundary as if also present in adjacent grid points



- **Problem:** finding similar sets of grid squares that have minutiae
  - rows: grid squares; columns: fingerprints sets. Not sparse!

Matching Fingerprints

- No need to minhash, since the number of grid squares is not too large
- Represent each fingerprint by **a bit-vector** with one position for each square
  - 1 in only those positions whose squares have minutiae
  - Pick 1024 sets of 3 grid squares randomly
  - For each set of three squares, two fingerprints that **each** have 1 for **all three** squares are candidate pairs
    - each set of three squares creates one bucket
    - fingerprints can be in many buckets

Matching Fingerprints

- Why make sense?
  - Suppose typical fingerprints have minutiae in **20%** of the grid squares
  - Suppose fingerprints from the same finger agree in at least **80%** of their squares
  - Prob. two random fingerprints each have 1' in three given squares = $((0.2)(0.2))^3 = 0.000064$ six independent event that a grid square has a minutia
  - Prob. two fingerprints from the same finger each have 1's in three given squares = $((0.2)(0.8))^3 = 0.004096$
  - Prob. for at least one of 1024 sets of three points = $1 - (1-0.004096)^{1024} = 0.985$

1.5% false negatives

1st print has a minutia in its square

2nd print also has a minutia in that square

6.3% false positives

For random fingerprints: $1-(1-0.000064)^{1024}=0.063$

Matching Newspaper Articles

- **Problem:** the same article, say from associated press, appears on the web site of many newspapers, but looks quite different
  - each newspaper surrounds the article with logos, ads, links to other articles …
  - a newspaper may also `crop' the article
- LSH substitute: candidates are articles of similar length
- Observe that news articles have a lot of stop words, while ads do not
  - "I recommend that you buy XXX for your laundry." vs "Buy XXX"
  - Define a shingle to be a stop word and the next two following words
  - By requiring each shingle to have a stop word, they biased the mapping to pick more shingles from the articles than from ads

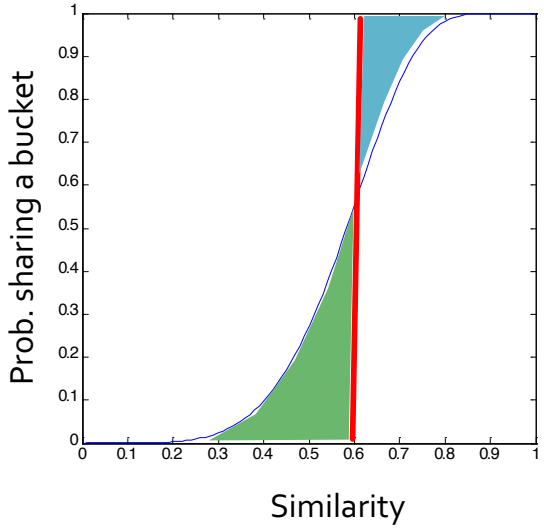
So far

- Locality-Sensitive Hashing
- Applications of Locality-Sensitive Hashing
- Distance Measures
- Locality-Sensitive Functions
- Methods for High Degrees of Similarity

Methods for High Degree of Similarity

Until now, LSH technique works well when the Jaccard similarity <= 80%. When sets are at a very high Jaccard similarity, we have other techniques with no false negatives.

- Length-Based Filtering
- Prefix-Based Indexing
- Position/Prefix-Based Indexing
- Suffix Length



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

Setting: Sets as Strings

- Represent sets by strings (lists of symbols):
  - order the universal set
  - represent a set by the string of its elements in sorted order
  - if the universal set is k-shingles, there is a natural lexicographic order
  - think of each shingle as a single symbol
    - e.g., the 2-shingling of abcad {ab, bc, ca, ad} is represented by the list [ab, ad, bc, ca]
  - a better way: order words lowest-frequency-first
    - index documents based on the early words in their lists

Jaccard and Edit Distance

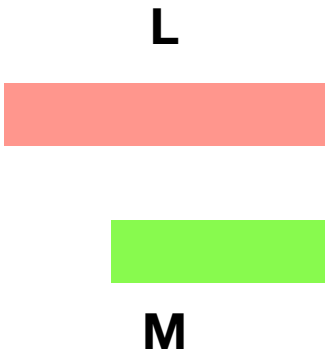
- Suppose two sets have Jaccard distance J and are represented by strings s1 and s2. Let the LCS (least common sequence) of s1 and s2 have length C and the edit distance be E. Then:
  - 1-J=C/(C+E)
  - J = E/(C+E)

works because these strings never repeat a symbol, and symbols appear in the same order

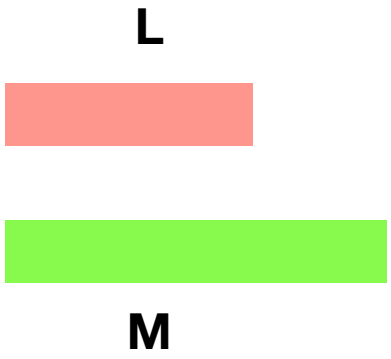
Length-Based Indexes

- Create an index on the length of strings

set A —> string A, length L
set B —> string B, length M
A is Jaccard distance J from B only if L(1-J) <= M <= L/(1-J)
E.g. if 1-J = 90% (Jaccard similarity), them M is between 90% and 111% of L



1-J = M/L
M = L(1-J)

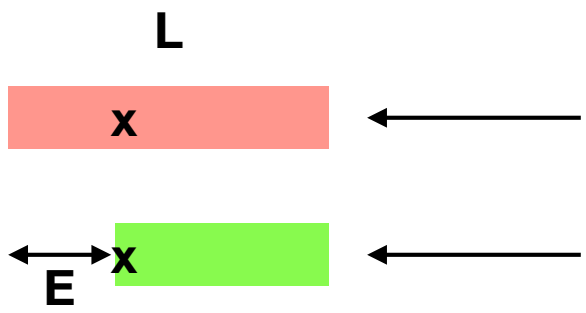


1-J =L/M
M = L/(1-J)

Given a string of length L, **we only need to look for candidates in the range L(1-J) to L/(1-J)**

Prefix-Based Indexing

- If two strings are 90% similar, they must share some symbol in their prefixes whose length is just above 10% of the length of each string
- We can base an index on symbols in just **the first ⌊JL+1⌋ positions** of a string of length L



Extreme case: 2nd string has none of the first E symbols of the 1st string, but they agree thereafter

- If two strings do not share any of the first E symbols, then J >= E/L
- Thus, E = JL, but any larger E is impossible
- Index E + 1 positions

Prefix-Based Indexing

- Think of a bucket for each possible symbol
- Each string of length L is placed in the bucket for each of its first $\lfloor JL+1 \rfloor$ positions
- Given a probe string s of length L, with J the limit on Jaccard distance:

  for (each symbol **a** among the first $\lfloor JL+1 \rfloor$ positions of **s**)
  look for other strings in the bucket for a;

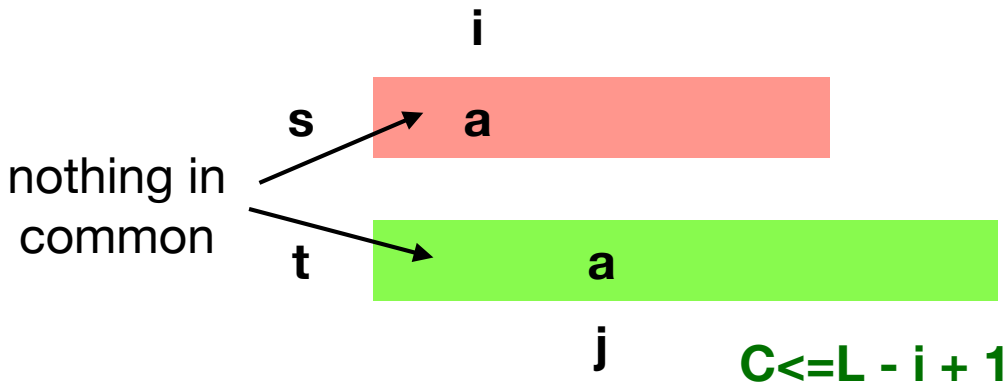
- E.g., let J=0.2
  - String abcdef is indexed under **a** and **b**
  - String acdfg is indexed under **a** and **c**
  - String bcde is indexed under **b**
  - If search for strings similar to cdef, we need look only in the bucket for **c**

Positions/Prefixes-Based Indexing

- Consider the strings s = acdefghijk and t = bcdefghijk, and assume SIM = 0.9. What are the buckets do s and t placed in?
  - s is indexed under a and c; t is indexed under b and c
- Can we do less comparison?
  - Since c is the second symbol of both, we know there will be two symbols, a and b in this case, that are in the union of the two sets but not in the intersection
  - Even if s and t are identical from c to the end, their intersection is 9 symbols and their union is 11; thus SIM(s, t) = 9/11, which is less than 0.9

Positions/Prefixes-Based Indexing

- Consider whether the first common symbol appear close enough to the fronts of both strings
- If position i of probe string s is the first position to match a prefix position of string t, and it matches position j, then the edit distance between s and t is at least i + j -2 **E>=i + j -2**



- The LCS of s and t is no longer than L-i+1, where L is the length of s
- If J is Jaccard distance, remember J = E/(E+C)
- Thus, (i+j-2)/(L+j-1) <= E/(E+C) $\Longrightarrow$ j <= (JL-J-i+2)/(1-J)

  (E+C)/E = 1 + C/E

Positions/Prefixes-Based Indexing

- Create a 2-attribute index on (symbol, position)
- If string s has symbol a as the i-th position of its prefix (first $\lfloor JL+1 \rfloor$ positions), add s to the bucket (a, i)
- Given probe string s, we only need to find a candidate once. So we
  - visit positions i of s in numerical order, assuming there have been no matches for earlier positions

```
for (i=1; i<=J*L+1; i++){
        let s have a in position i;
        for (j=1; j<=(J*L-J-i+2)/(1-J); j++)
                compare s with strings in bucket (a, j);
}
```

Positions/Prefixes-Based Indexing

- Suppose J=0.2
- Given probe string adegjkmprz, L=10, and the prefix is ade
- For the i-th position of the prefix, we must look at buckets where j <= (JL-J-i+2)/(1-J)=(3.8-i)/0.8
- For i=1: j<=3; for i=2: j<=2; for i=3: j<=1
- Look in the following buckets: (a, 1), (a, 2), (a, 3), (d, 1), (d, 2), (e, 1)
- Suppose string t is in none of these buckets
- Then the edit distance E is at least 3 (s and t share a, d, e, …)
- The LCS length C cannot be longer than s, i.e., 10
- Thus, J = E/(E+C) >= 3/13 > 0.2
- Need not compare s with t which is not in the six buckets!

Positions/Prefixes/Suffix Length Indexing

- We can add to our index a summary of what follows the positions being indexed. Help us eliminate candidate matches without comparing entire strings
- Idea: index on three attributes:
  - Character at a prefix position
  - Number of that position
  - Length of the suffix = number of positions in the entire string to the right of the given position
- s = acdefghijk, SIM = 0.9, what would the buckets that s is indexed under (symbol, position, suffix length)?
  - (a, 1, 9) and (c, 2, 8)

Positions/Prefixes/Suffix Length Indexing

- Given probe string s, we find string t because its j-th position matches the i-th position of s. The suffices of s and t have lengths k and m respectively
  - **A lower bound** on edit distance E is
    - i + j - 2
    - |k-m| = absolute difference of the lengths of the suffixes of s and t
  - **An upper bound** on the length C of the LCS is 1 + min(k, m)
- Letting J be Jaccard distance, J = E/(E+C), substitute the above values, we have
  - i + j -2 + |k-m| <= [J(1 + min(k,m))] / (1 - J)

Positions/Prefixes/Suffix Length Indexing

- Create a 3-attribute index on (symbol, position, suffix-length)
- if string s has symbol a as the i-th position of its prefix, and the length of the suffix relative to that position is k, add s to the bucket (a, i, k)
- Consider string s = abcde with J=0.2
- Prefix length = 2
- Index in: (a, 1, 4) and (b, 2, 3)
- To find candidate matches for a probe string s of length L, with required similarity J, visit the positions of s's prefix in order
- If position i has symbol a and suffix length k, look in index bucket (a, j, m) for all j and m s.t.
  - $i + j - 2 + |k-m| <= [J(1 + \min(k,m))]/(1 - J)$
  - look in (a, 1, 3), (a, 1, 4), (a, 1, 5), (a, 2, 4), (b, 1, 3)
  - for i = 1, note k = 4, we want $j -1 + |4-m| <= [0.2(\min(4,m) + 1)]/0.8$

Summary

- Three index schemes
  - symbol
  - symbol + position
  - symbol + position + suffix length
- The number of buckets grows as we add dimensions to the index, but the total size of the buckets remains the same
  - because each string is placed in $\lfloor JL+1 \rfloor$ buckets

Reading

- Jure Leskovec, Anand Raj, Jeff Ullman, “Mining of Massive Datasets,” Cambridge University Press, Chapter 3