NEAREST NEIGHBOR SEARCH ALGORITHMS

SOURANGSHU BHATTACHARYA

CSE, IIT KHARAGPUR

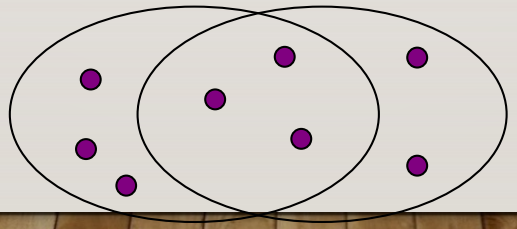
WEB: HTTPS://CSE.IITKGP.AC.IN/~SOURANGSHU/

EMAIL: SOURANGSHU@CSE.IITKGP.AC.IN

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** of two **sets** is the size of their intersection divided by the size of their union:
    $sim(C_1, C_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$
  - **Jaccard distance:** $d(C_1, C_2) = 1 - |C_1 \cap C_2| / |C_1 \cup C_2|$

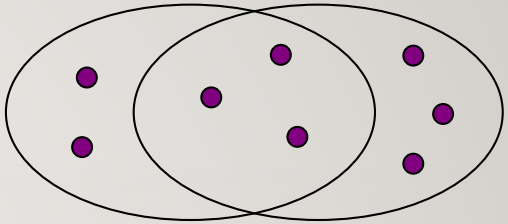


3 in intersection
8 in union
Jaccard similarity= 3/8
Jaccard distance = 5/8

TASK: 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

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:** $\mathbf{C_1}$ = 10111; $\mathbf{C_2}$ = 10011
  - Size of intersection **= 3**; size of union **= 4**,
  - **Jaccard similarity** (not distance) **= 3/4**
  - **Distance: 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**

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

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

- **Hash docs into buckets. Expect that "most" pairs of near duplicate docs hash into the same bucket!**

MIN-HASHING

- Imagine the rows of the boolean matrix permuted under **random permutation π**
- Define a **"hash" function $h_\pi(C)$** = the index of the **first** (in the permuted order π) row in which column ***C*** has value **1**:

$$h_\pi(C) = min_\pi \pi(C)$$

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

THE MIN-HASH PROPERTY

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

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

    $\pi(y) = \min(\pi(C_2))$ if $y \in C_2$

    One of the two cols had to have 1 at position ***y***
  - So the prob. that **both** are true is the prob. $y \in C_1 \cap C_2$
  - $\Pr[\min(\pi(C_1))=\min(\pi(C_2))]=|C_1 \cap C_2|/|C_1 \cup C_2| = sim(C_1, C_2)$

FOUR TYPES OF ROWS

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

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

  - **a** = # rows of type A, etc.
- **Note: $sim(C_1, C_2) = a/(a+b+c)$**
- **Then: 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
  - If it's a type-*A* row, then $h(C_1) = h(C_2)$

    If a type-*B* or type-*C* row, then not

SIMILARITY FOR SIGNATURES

- We know: $\mathbf{Pr[h_\pi(C_1) = h_\pi(C_2)] = sim(C_1, C_2)}$
- Now generalize to multiple hash functions

- **The *similarity of two signatures* is the fraction of the hash functions in which they agree**

- **Note:** Because of the Min-Hash property, the similarity of columns is the same as the expected similarity of their signatures

MIN-HASHING EXAMPLE

Permutation π

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

Input matrix (Shingles x Documents)

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

Signature matrix *M*

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

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

MIN-HASH 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*

$$sig(C)[i] = \min(\pi_i(C))$$

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

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

LOCALITY SENSITIVE HASHING

LSH: FIRST CUT

- **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 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**

CANDIDATES FROM MIN-HASH

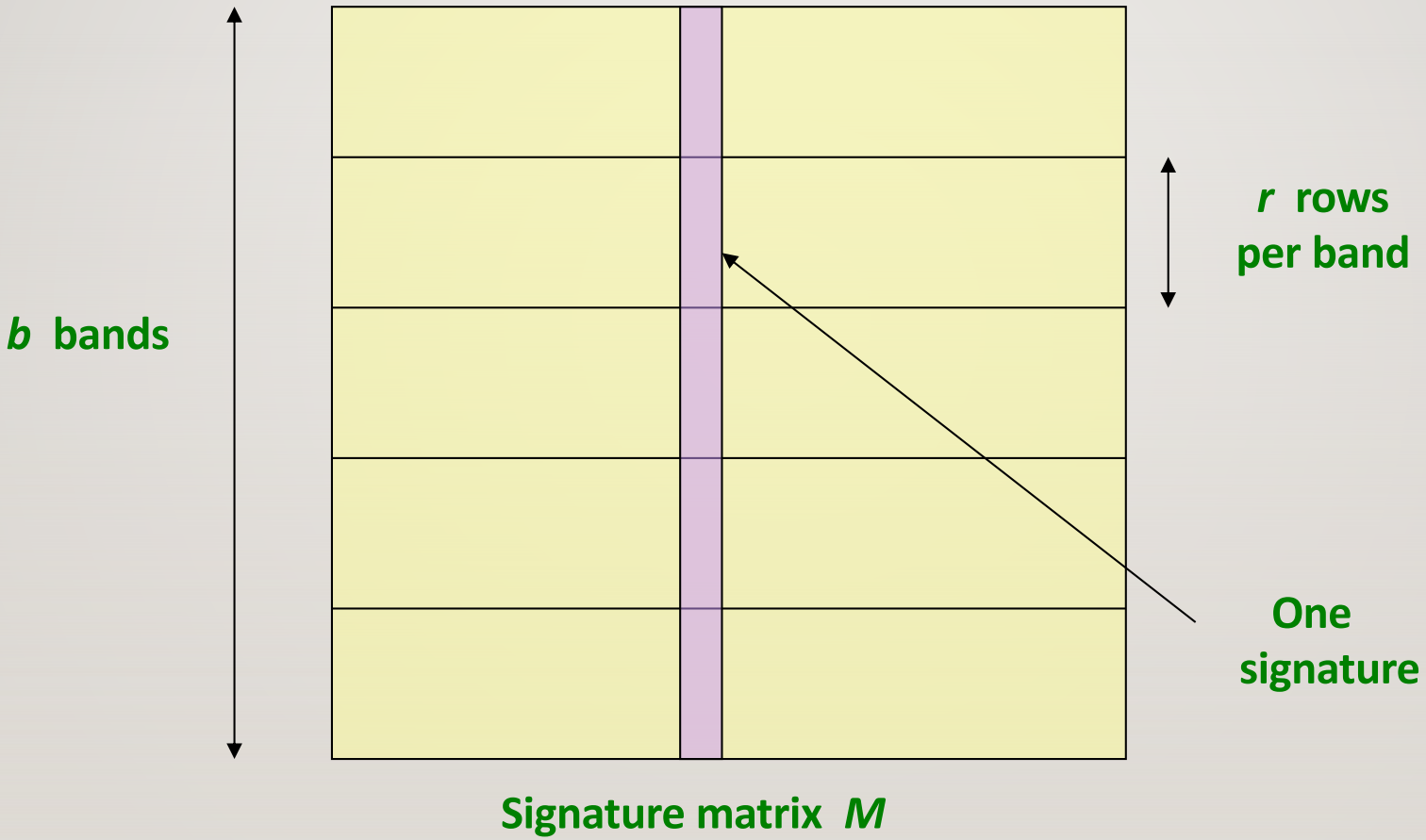
- **Pick a similarity threshold *s* (0 < s < 1)**

- 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 their signatures

LSH FOR MIN-HASH

- **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**

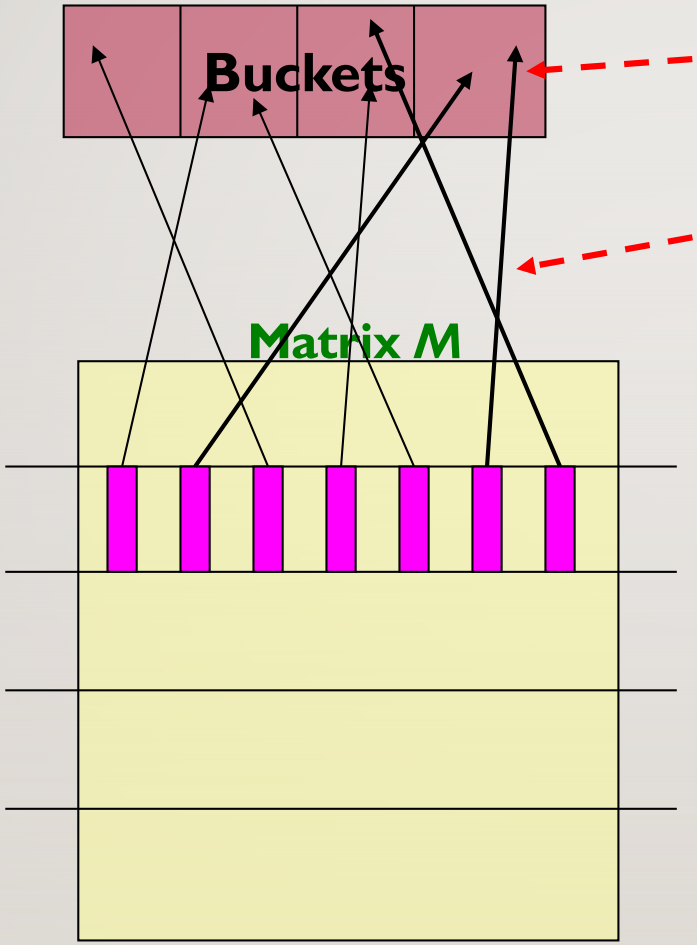
PARTITION *M* INTO *B* BANDS



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



Columns 2 and 6 are probably identical (**candidate pair**)

Columns 6 and 7 are surely different.

r rows

b 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

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

- **Find pairs of ≥ *s*=0.8 similarity, set b=20, r=5**
- **Assume:** sim(C_1, C_2) = 0.8
  - Since sim(C_1, C_2) ≥ **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

- **Find pairs of $\geq$ *s*=0.8 similarity, set b=20, r=5**
- **Assume:** sim(C_1, C_2) = 0.3
  - Since sim(C_1, C_2) < **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**

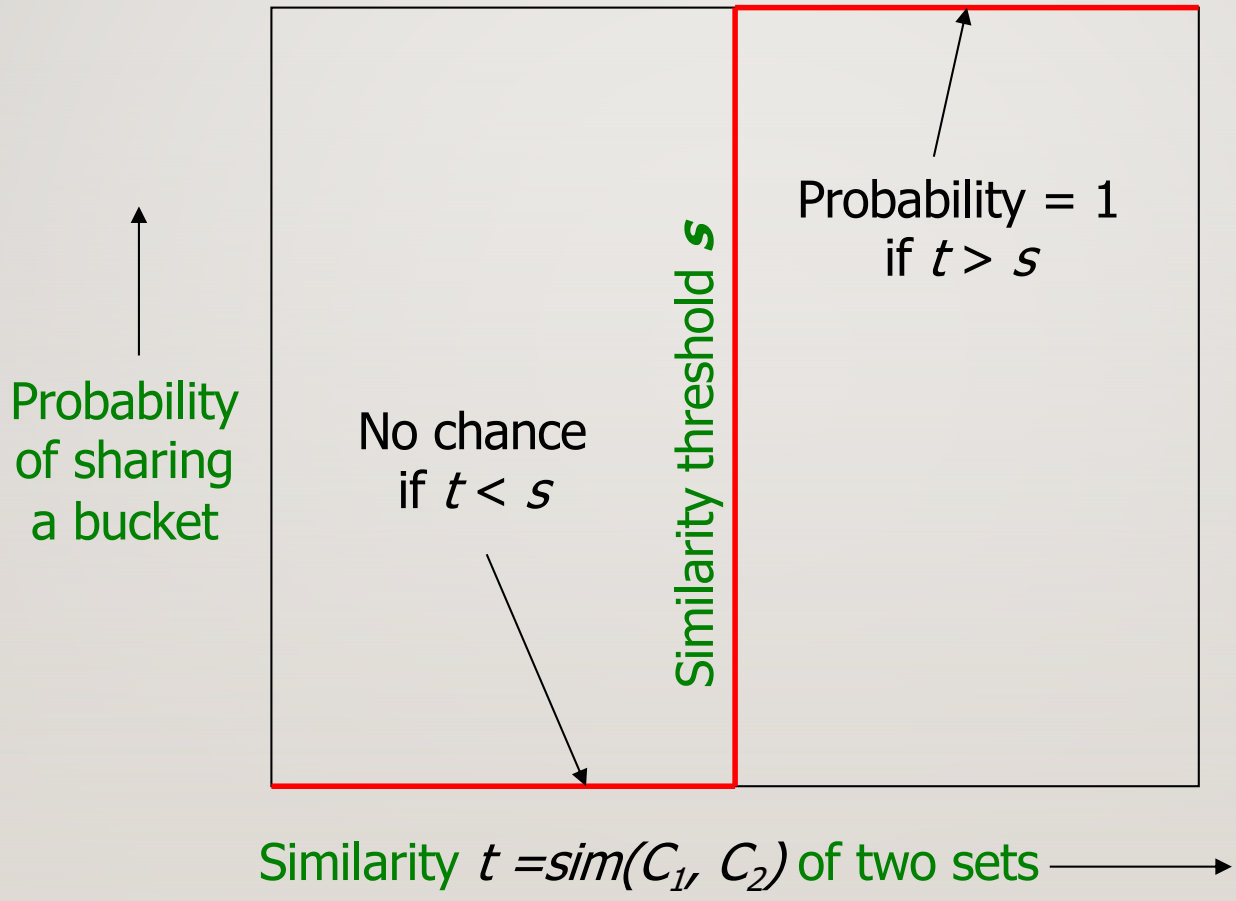
LSH INVOLVES A TRADEOFF

- **Pick:**
  - The number of Min-Hashes (rows of ***M***)
  - The number of bands ***b***, and
  - The number of rows ***r*** per band

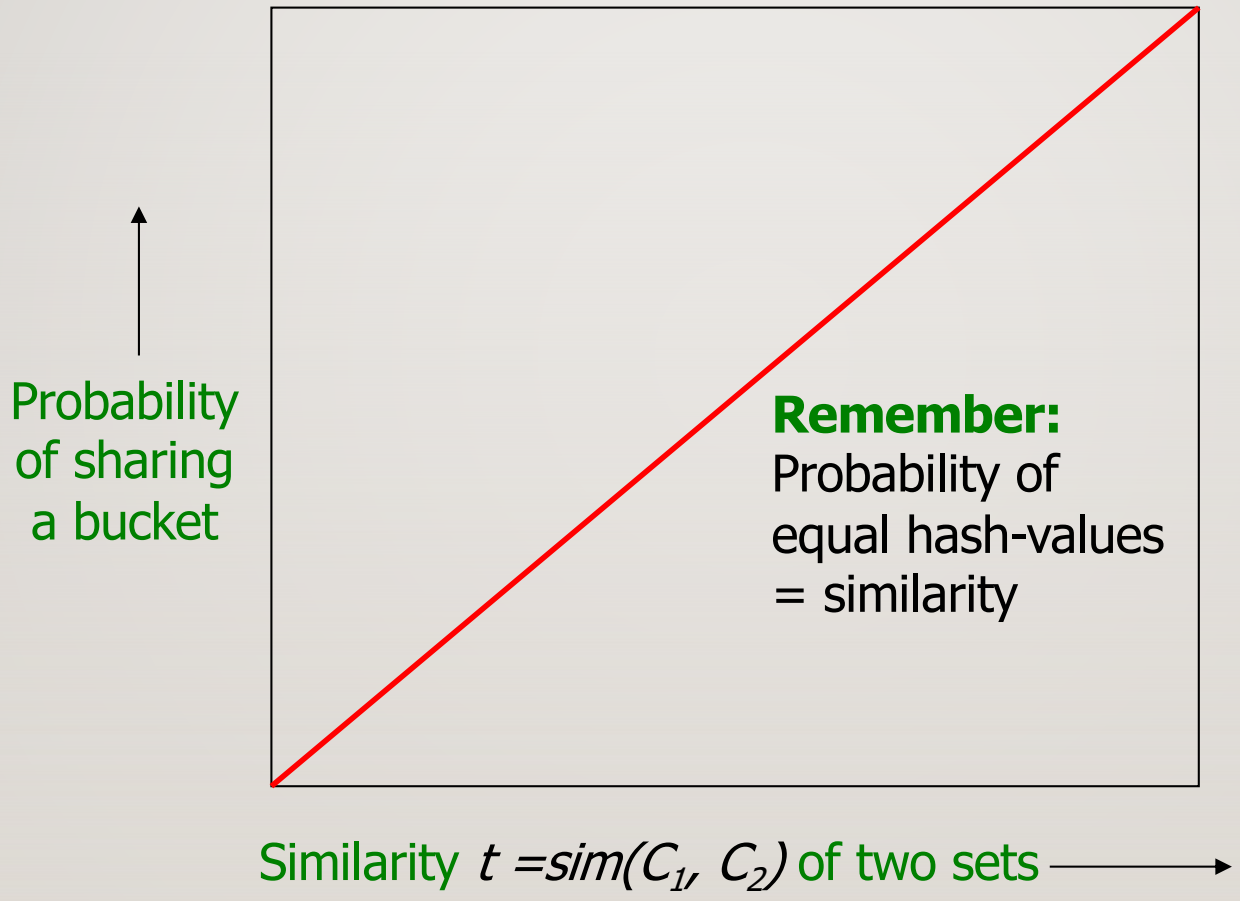
  to balance false positives/negatives

- **Example:** If we had only 15 bands of 5 rows, the number of false positives would go down, but the number of false negatives would go up

ANALYSIS OF LSH – WHAT WE WANT



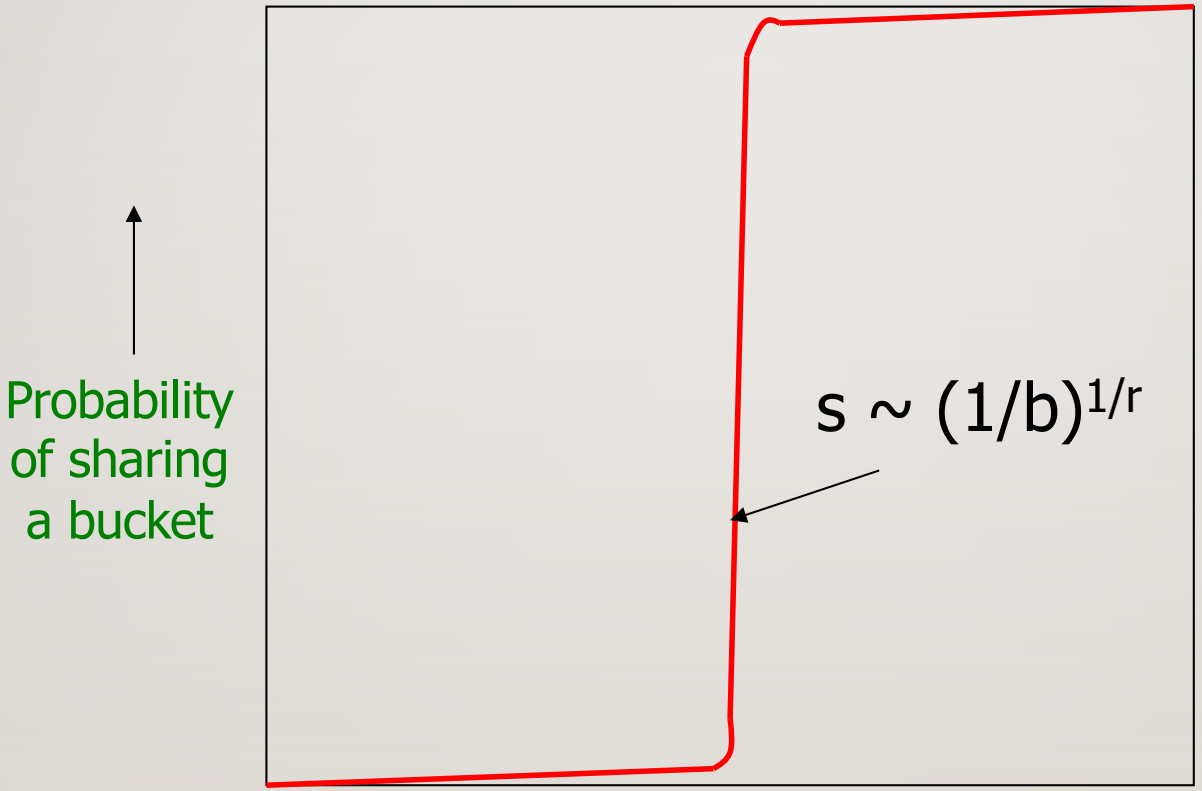
WHAT 1 BAND OF 1 ROW GIVES YOU



B BANDS, *R* ROWS/BAND

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

WHAT *B* BANDS OF *R* ROWS GIVES YOU



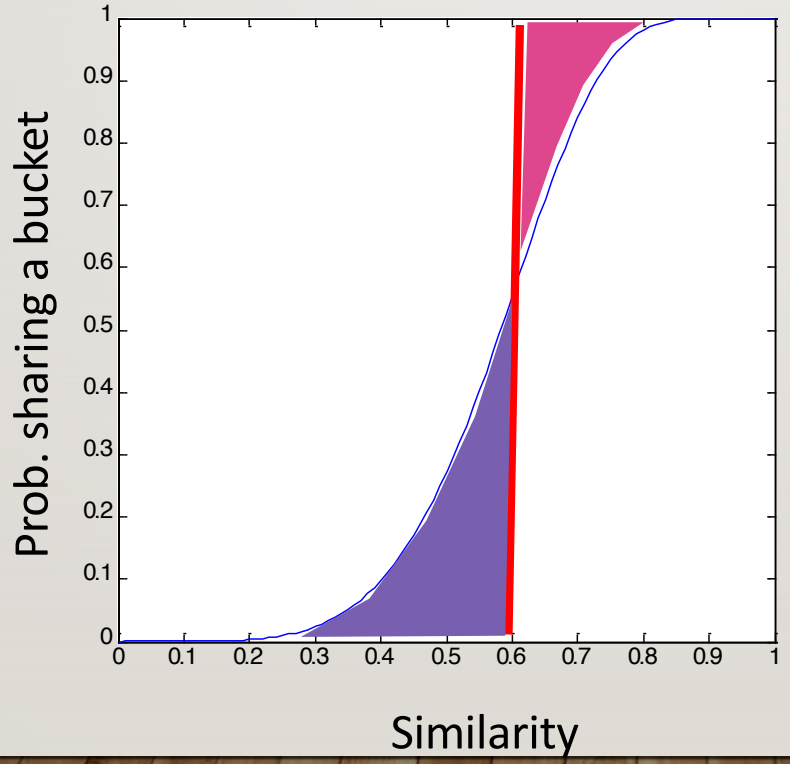
EXAMPLE: $B = 20; R = 5$

- **Similarity threshold s**
- **Prob. that at least 1 band is identical:**

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

PICKING *R* AND *B*: THE S-CURVE

- **Picking *r* and *b* to get the best S-curve**
  - 50 hash-functions (r=5, b=10)



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**

GENERALIZATION OF LSH

LOCALITY SENSITIVE HASHING

- Originally defined in terms of a similarity function [C'02]

$$\Pr_{h \in H}[h(x) = h(y)] = s(x, y)$$

- Given universe U and a similarity $s: U \times U \rightarrow [0,1]$, does there exist a prob distribution over some hash family H such that

$$s(x, y) = 1 \rightarrow x = y$$
$$s(x, y) = s(y, x)$$

LOCALITY SENSITIVE HASHING

[Indyk Motwani]

- Hash family H is *locality sensitive* if

$\Pr[h(x) = h(y)]$ is high if x is close to y

$\Pr[h(x) = h(y)]$ is low if x is far from y

- Not clear such functions exist for all distance functions

HAMMING DISTANCE

- Points are bit strings of length d
- $H(x, y) = |\{i, x_i \neq y_i\}|$ $\quad S_H(x, y) = 1 - \frac{H(x,y)}{d}$
- Define a hash function h by sampling a set of positions
  - $x = 1011010001, y = 0111010101$
  - $S = \{1,5,7\}$
  - $h(x) = 100, h(y) = 100$

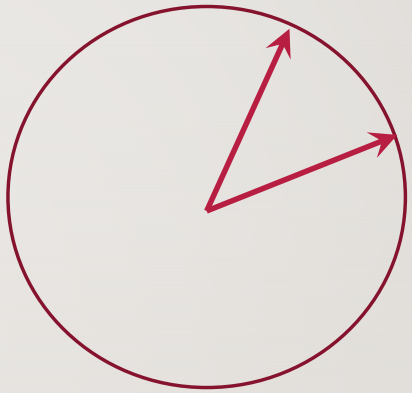
LSH FOR HAMMING DISTANCE

- The above hash family is locality sensitive, $k = |S|$

$$\Pr[h(x) = h(y)] = \left(1 - \frac{H(x,y)}{d}\right)^k$$

LSH FOR ANGLE DISTANCE

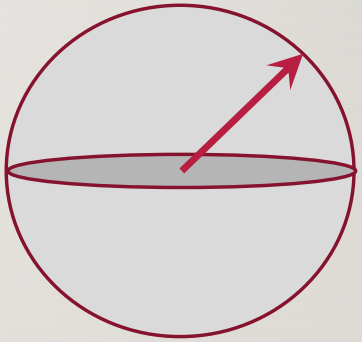
- x, y are unit norm vectors
- $d(x, y) = \cos^{-1}(x \cdot y) = \theta$
- $S(x, y) = 1 - \theta/\pi$



- Choose direction v uniformly at random
  - $h_v(x) = sign(v \cdot x)$
  - $\Pr[h_v(x) = h_v(y)] = 1 - \theta/\pi$

ASIDE: PICKING A DIRECTION U.A.R.

- How to sample a vector $x \in R^d, |x|_2 = 1$ and the direction is uniform among all possible directions



- Generate $x = (x_1, \ldots . x_d), x_i \sim N(0,1)$ iid
- Normalize $\frac{x}{|x|_2}$
  - By writing the pdf of the d-dimensional Gaussian in polar form, easy to see that this is uniform direction on unit sphere

WHICH SIMILARITIES ADMIT LSH?

- There are various similarities and distance that are used in scientific literature
  - Encyclopedia of distances DL'11

- Will there be an LSH for each one of them?
  - Similarity is LSHable if there exists an LSH for it

[slide courtesy R. Kumar]

LSHABLE SIMILARITIES

Thm: S is LSHable → 1 – S is a metric

$$d(x,y) = 0 \rightarrow x = y$$
$$d(x,y) = d(y,x)$$
$$d(x,y) + d(y,z) \geq d(x,z)$$

Fix hash function $h \in H$ and define

$$\Delta_h(A,B) = [h(A) \neq h(B)]$$
$$1 - \mathrm{S(A,B)} = \Pr_h[\Delta_h(A,B)]$$

Also

$$\Delta_h(A,B) + \Delta_h(B,C) \geq \Delta_h(A,C)$$

EXAMPLE OF NON-LSHABLE SIMILARITIES

- $d(A,B) = 1 - s(A,B)$
- Sorenson-Dice : $s(A,B) = \frac{2|A \cap B|}{|A|+|B|}$
  - Ex: $A = \{a\}, B = \{b\}, C = \{a,b\}$
  - $s(A,B) = 0, s(B,C) = s(A,C) = \frac{2}{3}$
- Overlap: $s(A,B) = \frac{|A \cap B|}{\min(|A|,|B|)}$
  - $s(A,B) = 0, s(A,C) = 1 = s(B,C)$

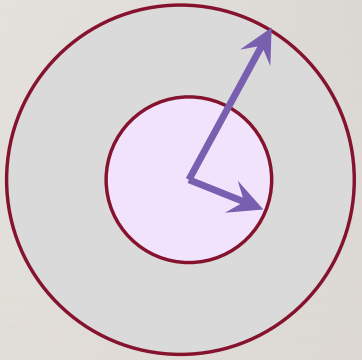
GAP DEFINITION OF LSH

- A family is (r, R, p, q) LSH if

IMRS'97, IM'98, GIM'99

$$\Pr_{h \in H}[h(x) = h(y)] \geq p \; if \; d(x, y) \leq r$$

$$\Pr_{h \in H}[h(x) = h(y)] \leq q \; if \; d(x, y) \geq R$$



Here $p > q$.

GAP LSH

- All the previous constructions satisfy the gap definition
  - Ex: for $JS(S,T) = \frac{|S \cap T|}{|S \cup T|}$

$$JD(S,T) \leq r \rightarrow JS(S,T) \geq 1 - r \rightarrow \Pr[h(S) = h(T)] = JS(S,T) \geq 1 - r$$

$$JD(S,T) \geq R \rightarrow JS(S,T) \leq 1 - R \rightarrow \Pr[h(S) = h(T)] = JS(S,T) \leq 1 - R$$

Hence is a $(r, R, 1-r, 1-R)$ LSH

L2 NORM

- $d(x, y) = \sqrt{(\sum_i (x_i - y_i)^2}$
- $u =$ random unit norm vector, $w \in R$ parameter, $b \sim Unif[0, w]$

- $h(x) = \lfloor \frac{u \cdot x + b}{w} \rfloor$
- If $|x - y|_2 < \frac{w}{2}$, $\Pr[h(x) = h(y)] \geq \frac{1}{3}$
- If $|x - y|_2 > 4w, \Pr[h(x) = h(y)] \leq \frac{1}{4}$

SOLVING THE NEAR NEIGHBOUR

- (r, c) −near neighbour problem
  - Given query point q, return all points p such that $d(p, q) < r$ and none such that $d(p, q) > cr$
  - Solving this gives a subroutine to solve the "nearest neighbour", by building a data-structure for each r , in powers of $(1 + \epsilon)$

HOW TO ACTUALLY USE IT?

- Need to amplify the probability of collisions for "near" points

BAND CONSTRUCTION

- AND-ing of LSH
  - Define a composite function $H(x) = (h_1(x), \ldots h_k(x))$
  - $\Pr[H(x) = H(y)] = \Pi_i \Pr[h_i(x) = h_i(y)] = \Pr[h_1(x) = h_1(y)]^k$
- OR-ing
  - Create L independent hash-tables for $H_1, H_2, \ldots H_L$
  - Given query x, search in $\cup_j H_j(x)$

EXAMPLE

	S_1	S_2	S_3	S_4
A	1	0	1	0
B	1	0	0	1
C	0	1	0	1
D	0	1	0	1
E	0	1	0	1
F	1	0	1	0
G	1	0	1	0

	S1	S2	S3	S3
h1	1	2	1	2
h2	2	1	3	1

	S1	S2	S3	S3
h3	3	1	2	1
h4	1	3	2	2

WHY IS THIS BETTER?

- Consider x, y with $\Pr[h(x) = h(y)] = 1 - d(x, y)$

- Probability of not finding y as one of the candidates in $\cup_j H_j(x)$

$$1 - \left(1 - (1 - d)^k\right)^L$$

CREATING AN LSH

- Query x
- If we have a (r, cr, p, q) LSH
- For any y, with $|x - y| < r$,
  - Prob of y as candidate in $\cup_j H_j(x) \geq 1 - \left(1 - p^k\right)^L \geq 1 - \frac{1}{e}$
- For any $z, |x - z| > cr$,
  - Prob of z as candidate in any fixed $H_j(x) \leq q^k$
  - Expected number of such $z \leq Lq^k \leq L = n^\rho$
  - $\rho < 1$

$$\rho = \frac{\log(p)}{\log(q)} \quad L = n^\rho \quad k = \log(n)/\log\left(\frac{1}{q}\right)$$

RUNTIME

- Space used = $n^{1+\rho}$
- Query time = $n^{\rho} \times (k + d)$ [time for k-hashes & brute force comparison]

- We can show that for Hamming, angle etc, $\rho \approx \frac{1}{c}$
  - Can get 2-approx near neighbors with $O(\sqrt{n})$ neighbour comparisons

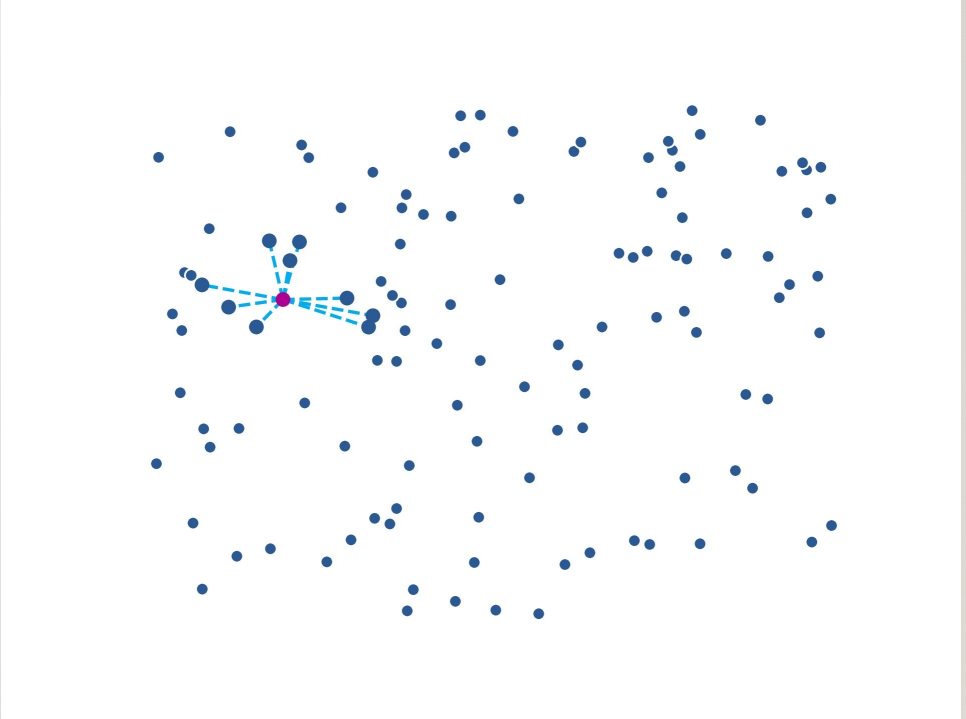
LSH: THEORY VS PRACTICE

- In order to design LSH in practice, the theoretical parameter values are only a guidance
  - Typically need to search over the parameter space to find a good operating point
  - Data statistics can provide some guidance.

HIERARCHICAL NAVIGABLE SMALL WORLD

APPROXIMATE NN SEARCH

- **Data (D):**
  - Many vectors (millions or billions)
- **Input (Q):**
  - One query vector (not necessarily from **D**)
- **Output:**
  - The *k* vectors from **D** that are closest to **Q**



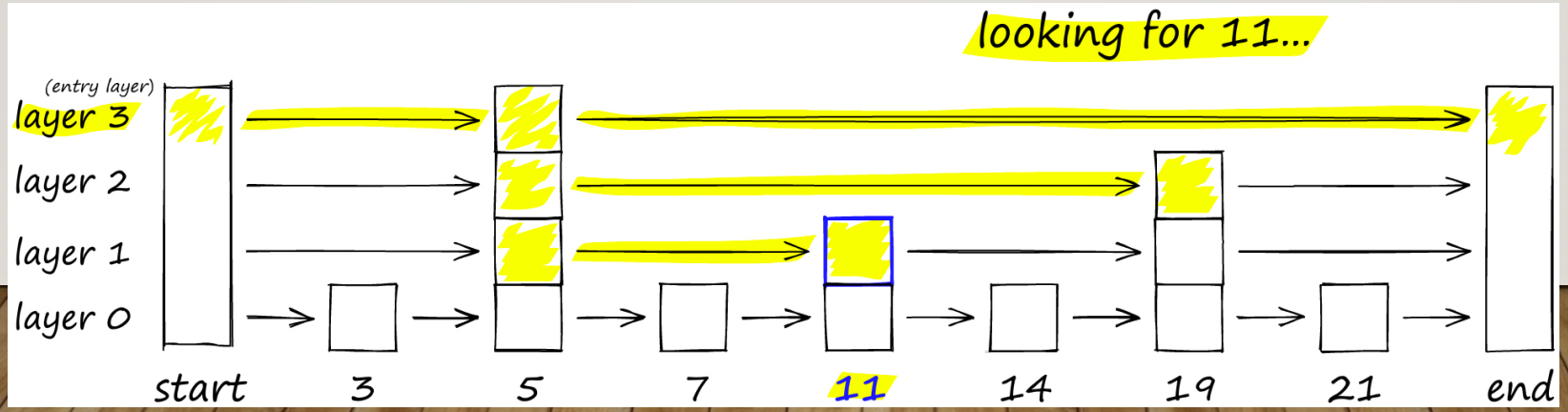
SOLUTIONS

- Locality sensitive hashing
- Space subdivision methods:
  - KD-trees
  - Slow for high-dimensional data
- Proximity Graph based methods
  - HNSW

- For index compression (not discussed):
  - Product quantization.

MOTIVATION: PROBABILITY SKIP LIST

- A linked list structure for fast search and insertion of new elements
  - Allows fast search in a sorted array.
- Several layers of linked lists.
  - First layer *skip* many intermediate nodes.
  - The number of *'skips'* decreases in lower layers
- Search: start at the highest layer follow links until you find the element greater than key.
  - Move down the layer and repeat.

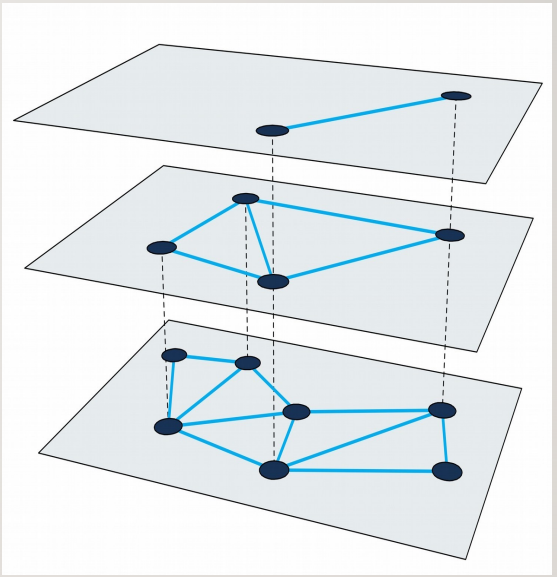


PROXIMITY GRAPH

- Vertices are datapoints
  - Edges between datapoints close to each other.
- Search is performed by browsing neighbors for each points.
  - Start with an initial point.
  - Browse to the neighbor closest to the query point
  - Stop when you have reached local minima, i.e. distance to the current node is less than distance to all neighbors
- K-nearest neighbor graph
  - The length of search path is large.
  - Not small world.

HIERARCHICAL NAVIGABLE SMALL WORLD

- The proximity graph should be:
  - Navigable Small World graph.
    - The maximum distance between any two nodes should be low.
    - PolyLogarithmic scaling during greedy traversal.
    - There are high degree nodes which are connected to many nodes.
    - Sometimes, performance degrades due to far entry point.
  - Hierarchical NSW:
    - Graphs at different levels with varying sparsity.
    - Inspired by skip lists.



HNSW - SEARCH

- Given a HNSW index for a dataset, and query q:
  1. Start searching from the top layer with the default entry point.
  2. Calculate the entry point to the lower layer from the nearest neighbor found in previous layer.
  3. Repeat from step 1.
- For searching the nearest neighbors in each layer:
  - Search the neighborhood of each point in the neighborhood of entry point.
  - Return a list of *ef* closest points to query.
- Detailed algorithm in the next slide.

HNSW - SEARCH

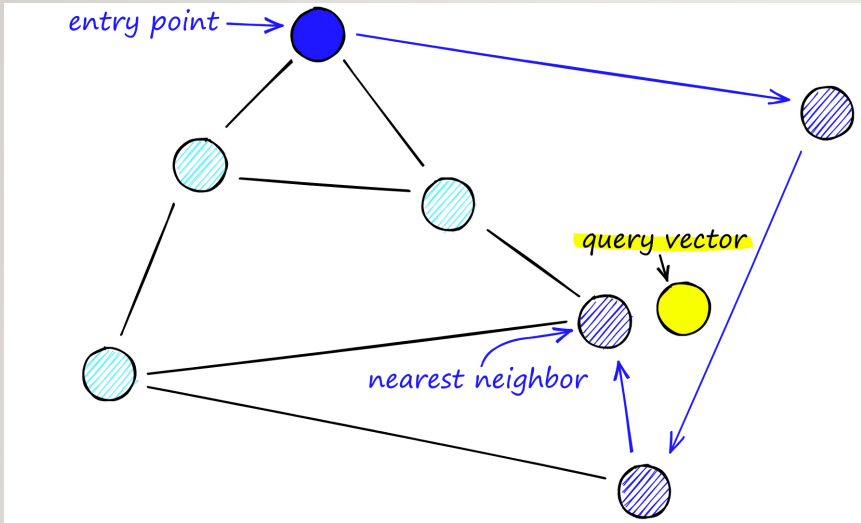
Algorithm 5
K-NN-SEARCH(*hnsw*, *q*, *K*, *ef*)
Input: multilayer graph *hnsw*, query element *q*, number of nearest neighbors to return *K*, size of the dynamic candidate list *ef*
Output: *K* nearest elements to *q*

```
1  W ← ∅   // set for the current nearest elements
2  ep ← get enter point for hnsw
3  L ← level of ep    // top layer for hnsw
4  for lc ← L … 1
5     W ← SEARCH-LAYER(q, ep, ef=1, lc)
6     ep ← get nearest element from W to q
7  W ← SEARCH-LAYER(q, ep, ef, lc =0)
8  return K nearest elements from W to q
```

Algorithm 2
SEARCH-LAYER(*q*, *ep*, *ef*, l_c)
Input: query element *q*, enter points *ep*, number of nearest to *q* elements to return *ef*, layer number l_c
Output: *ef* closest neighbors to *q*

```
1  v ← ep      // set of visited elements
2  C ← ep      // set of candidates
3  W ← ep      // dynamic list of found nearest neighbors
4  while |C| > 0
5     c ← extract nearest element from C to q
6     f ← get furthest element from W to q
7     if distance(c, q) > distance(f, q)
8        break   // all elements in W are evaluated
9     for each e ∈ neighbourhood(c) at layer lc   // update C and W
10       if e ∉ v
11          v ← v ⋃ e
12          f ← get furthest element from W to q
13          if distance(e, q) < distance(f, q) or |W| < ef
14             C ← C ⋃ e
15             W ← W ⋃ e
16             if |W| > ef
17                remove furthest element from W to q
18 return W
```

HNSW – SEARCH LAYER



Algorithm 2

SEARCH-LAYER(q, ep, ef, l_c)

Input: query element q, enter points ep, number of nearest to q elements to return ef, layer number l_c

Output: ef closest neighbors to q

```
1  v ← ep      // set of visited elements
2  C ← ep      // set of candidates
3  W ← ep      // dynamic list of found nearest neighbors
4  while |C| > 0
5    c ← extract nearest element from C to q
6    f ← get furthest element from W to q
7    if distance(c, q) > distance(f, q)
8      break   // all elements in W are evaluated
9    for each e ∈ neighbourhood(c) at layer l_c   // update C and W
10     if e ∉ v
11       v ← v ⋃ e
12       f ← get furthest element from W to q
13       if distance(e, q) < distance(f, q) or |W| < ef
14         C ← C ⋃ e
15         W ← W ⋃ e
16         if |W| > ef
17           remove furthest element from W to q
18 return W
```

HNSW - INSERT

- The HNSW index is formed by first creating an empty index with no levels. The parameters are:
  - Normalization factor for level generation - m_L.
  - Maximum number of connections for each datapoint per layer - M_{max}.
- Randomly select the maximum layer l at which the datapoint is inserted.
- For each layer from l to 0:
  - Find the nearest neighbors using entry point to the layer.
  - Connect the inserted point to them and shrink each of them to size M_{max}.

HNSW - INSERT

Algorithm 1

INSERT(*hnsw*, *q*, *M*, M_{max}, *efConstruction*, m_L)

Input: multilayer graph *hnsw*, new element *q*, number of established connections *M*, maximum number of connections for each element per layer M_{max}, size of the dynamic candidate list *efConstruction*, normalization factor for level generation m_L

Output: update *hnsw* inserting element *q*

1 $W \leftarrow \emptyset$ // list for the currently found nearest elements
2 *ep* ← get enter point for *hnsw*
3 $L \leftarrow$ level of *ep* // top layer for *hnsw*
4 $l \leftarrow \lfloor -\ln(unif(0..1)) \cdot m_L \rfloor$ // new element's level
5 **for** $l_c \leftarrow L \ldots l+1$
6 $W \leftarrow$ SEARCH-LAYER(*q*, *ep*, *ef*=1, l_c)
7 *ep* ← get the nearest element from *W* to *q*
8 **for** $l_c \leftarrow \min(L, l) \ldots 0$
9 $W \leftarrow$ SEARCH-LAYER(*q*, *ep*, *efConstruction*, l_c)
10 *neighbors* ← SELECT-NEIGHBORS(*q*, *W*, *M*, l_c) // alg. 3 or alg. 4
11 add bidirectionall connectionts from *neighbors* to *q* at layer l_c
12 **for** each $e \in$ *neighbors* // shrink connections if needed
13 *eConn* ← *neighbourhood*(*e*) at layer l_c
14 **if** $|eConn| > M_{max}$ // shrink connections of *e*
// if $l_c = 0$ then $M_{max} = M_{max0}$
15 *eNewConn* ← SELECT-NEIGHBORS(*e*, *eConn*, M_{max}, l_c)
// alg. 3 or alg. 4
16 set *neighbourhood*(*e*) at layer l_c to *eNewConn*
17 *ep* ← *W*
18 **if** $l > L$
19 set enter point for *hnsw* to *q*

IMPLEMENTATION NOTES

- FAISS:
  - https://github.com/facebookresearch/faiss/wiki/
  - L2 Distance based search.
  - Many indexes implemented – Flat, IVF, IndexBinaryHash.
  - Another key idea is Product quantization:
    - Find k-centroids (e.g. using k-means clustering) – expensive
    - Encode data as a binary vector by first splitting the vector dimensions and then encoding each dimension as sign of dot product with all the centroids.
    - Multi-probe can be used to reduce memory requirement by reducing k.

FAISS SEARCH: FLAT INDEX

Input: vectors

Build index:

```
import faiss
vector_dimension = vectors.shape[1]
index = faiss.IndexFlatL2(vector_dimension)
faiss.normalize_L2(vectors)
index.add(vectors)
```

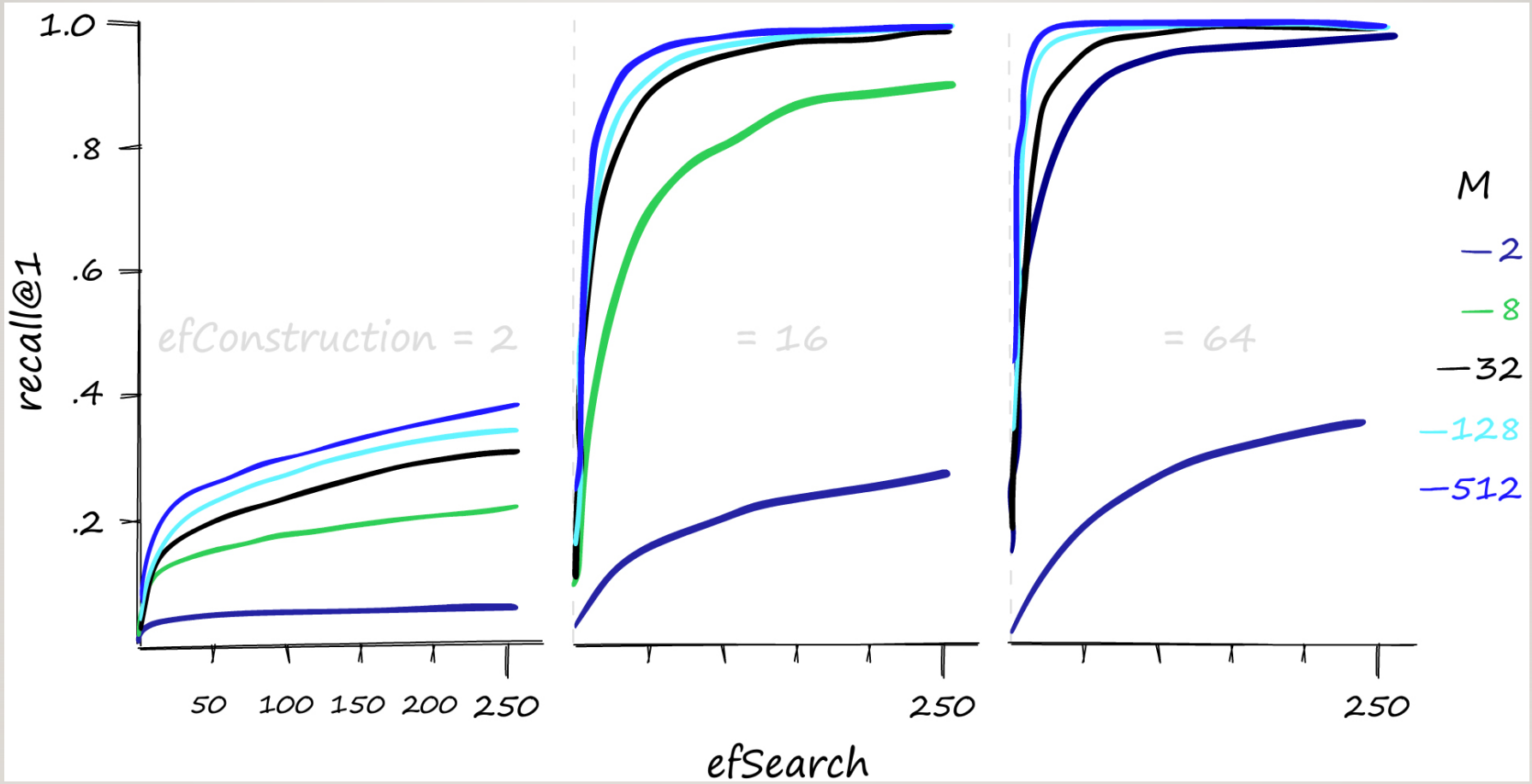
Search:

```
k = index.ntotal
distances, ann = index.search(query_vector, k=k)
```

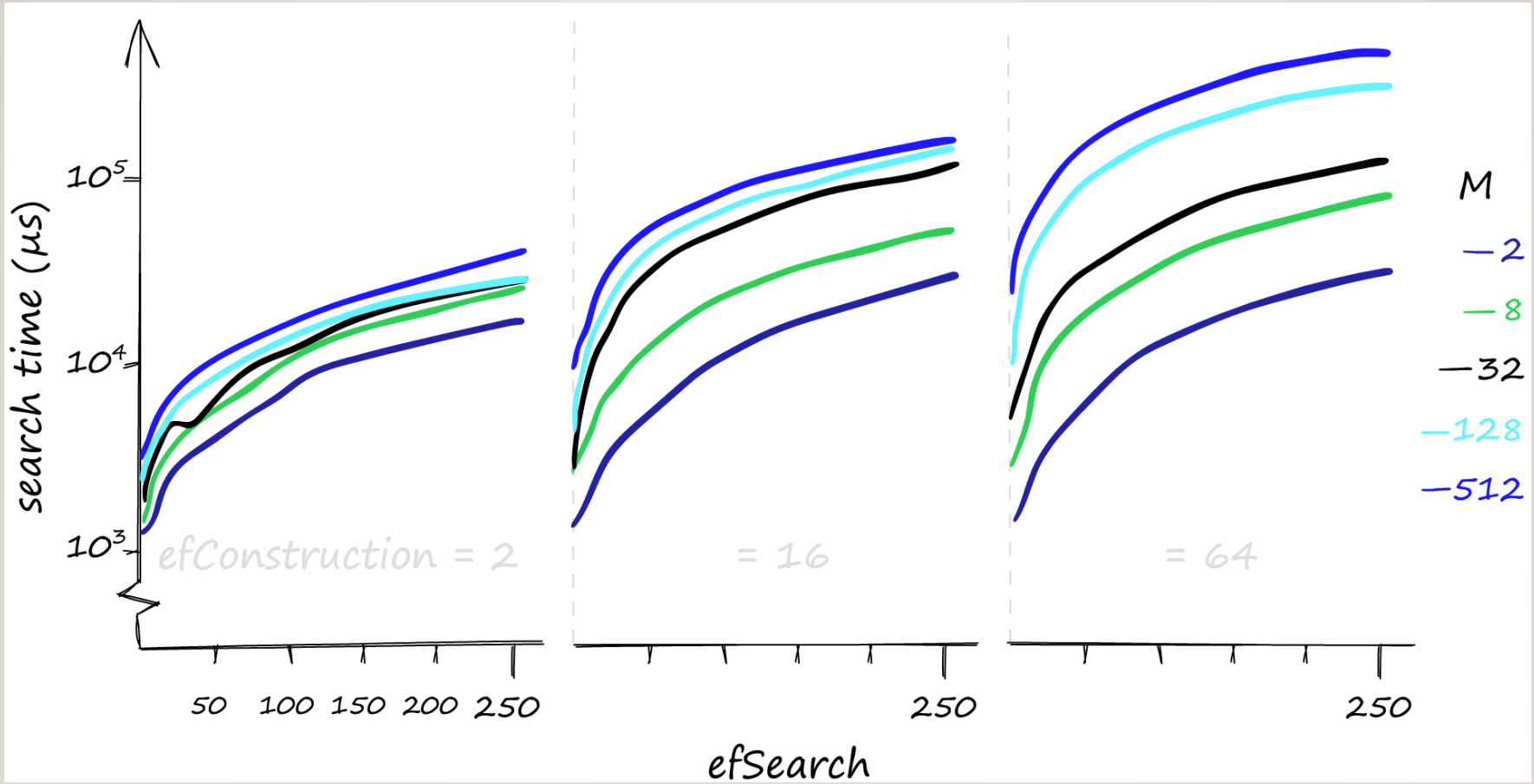
FAISS: HNSW

```
d = 128 # vector size
M = 32 #maximum connectivity of vertices
index = faiss.IndexHNSWFlat(d, M)
index.hnsw.efConstruction = efConstruction
index.add(xb)
# after adding our data the level has been set
print(index.hnsw.max_level)
index.hnsw.efSearch = efSearch
# and now we can search
index.search(xq[:1000], k=1)
```

HNSW PARAMETERS



HNSW PARAMETERS



REFERENCES

- Malkov, Yu A., and Dmitry A. Yashunin. "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs." *IEEE transactions on pattern analysis and machine intelligence* 42, no. 4 (2018): 824-836.

- Blog article: https://www.pinecone.io/learn/series/faiss/hnsw/

THANKS

QUESTIONS?

Email: sourangshu@cse.iitkgp.ac.in