

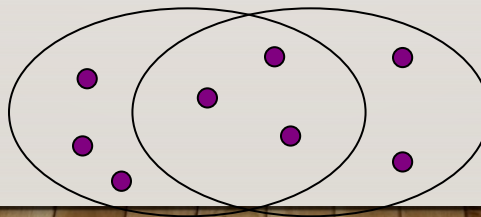


FINDING SIMILAR ITEMS



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:
$$\text{sim}(\mathbf{C}_1, \mathbf{C}_2) = |\mathbf{C}_1 \cap \mathbf{C}_2| / |\mathbf{C}_1 \cup \mathbf{C}_2|$$
 - **Jaccard distance:** $d(\mathbf{C}_1, \mathbf{C}_2) = 1 - |\mathbf{C}_1 \cap \mathbf{C}_2| / |\mathbf{C}_1 \cup \mathbf{C}_2|$



3 in intersection

8 in union

Jaccard similarity = 3/8

Jaccard distance = 5/8



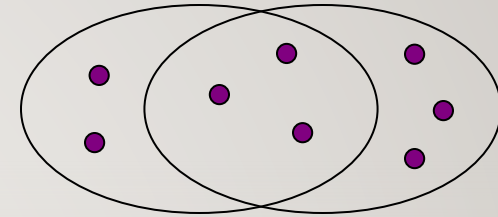
TASK: FINDING SIMILAR DOCUMENTS

- **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:** $C_1 = 10111$; $C_2 = 10011$
 - Size of intersection = **3**; size of union = **4**,
 - **Jaccard similarity** (not distance) = **3/4**
 - **Distance:** $d(C_1, C_2) = 1 - (\text{Jaccard similarity}) = 1/4$





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:** $\text{sim}(\mathbf{C}_1, \mathbf{C}_2) = ?$
 - Size of intersection = 3; size of union = 6, Jaccard similarity (not distance) = $3/6$
 - $d(\mathbf{C}_1, \mathbf{C}_2) = 1 - (\text{Jaccard similarity}) = 3/6$

	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
	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 $h(\cdot)$ 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 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** $h_{\pi}(\mathbf{C})$ = the index of the **first** (in the permuted order π) row in which column $\mathbf{C}$ has value **1**:

$$h_{\pi}(\mathbf{C}) = \min_{\pi} \pi(\mathbf{C})$$

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



THE MIN-HASH PROPERTY

- Choose a random permutation π
- Claim: $\Pr[h_\pi(C_1) = h_\pi(C_2)] = \text{sim}(C_1, C_2)$
- Why?
 - Let $\mathbf{X}$ be a doc (set of shingles), $y \in \mathbf{X}$ is a shingle
 - Then: $\Pr[\pi(y) = \min(\pi(\mathbf{X}))] = 1/|\mathbf{X}|$
 - It is equally likely that any $y \in \mathbf{X}$ 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 $y \in C_2$
 - So the prob. that **both** are true is the prob. $y \in C_1 \cap C_2$
 - $\Pr[\min(\pi(C_1)) = \min(\pi(C_2))] = |C_1 \cap C_2| / |C_1 \cup C_2| = \text{sim}(C_1, C_2)$

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

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



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:** $\text{sim}(C_1, C_2) = a/(a + b + c)$
- Then:** $\Pr[h(C_1) = h(C_2)] = \text{Sim}(C_1, C_2)$
 - 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: $\Pr[h_\pi(C_1) = h_\pi(C_2)] = \text{sim}(C_1, C_2)$
- 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-HASH SIGNATURES

- **Pick $K=100$ random permutations of the rows**
- Think of $\text{sig}(\mathbf{C})$ as a column vector
- $\text{sig}(\mathbf{C})[i]$ = according to the i -th permutation, the index of the first row that has a 1 in column C

$$\text{sig}(\mathbf{C})[i] = \min (\pi_i(\mathbf{C}))$$

- **Note:** The sketch (signature) of document C is small $\sim \mathbf{100 \text{ bytes!}}$
- **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

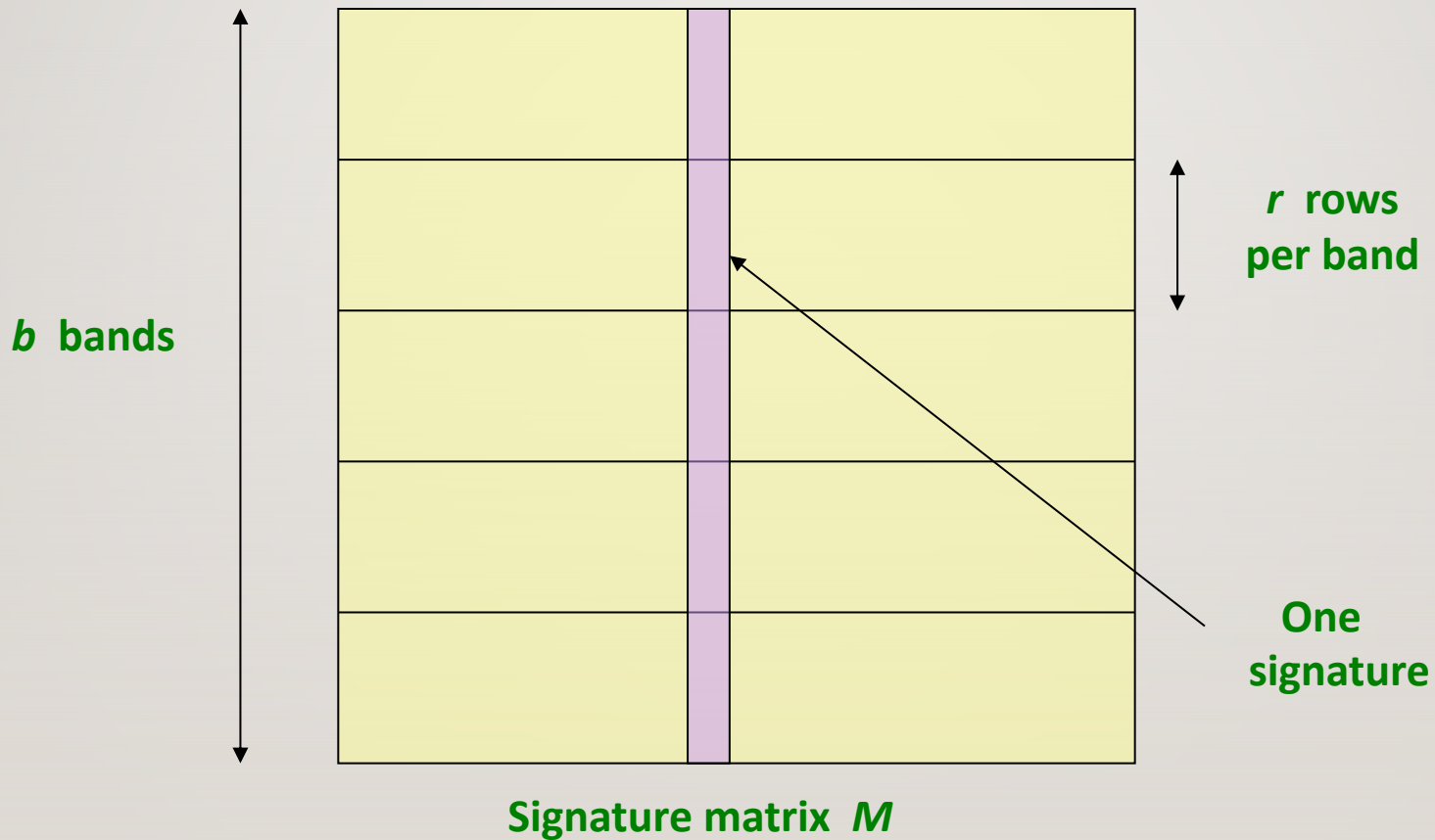


LSH FOR MIN-HASH

- **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

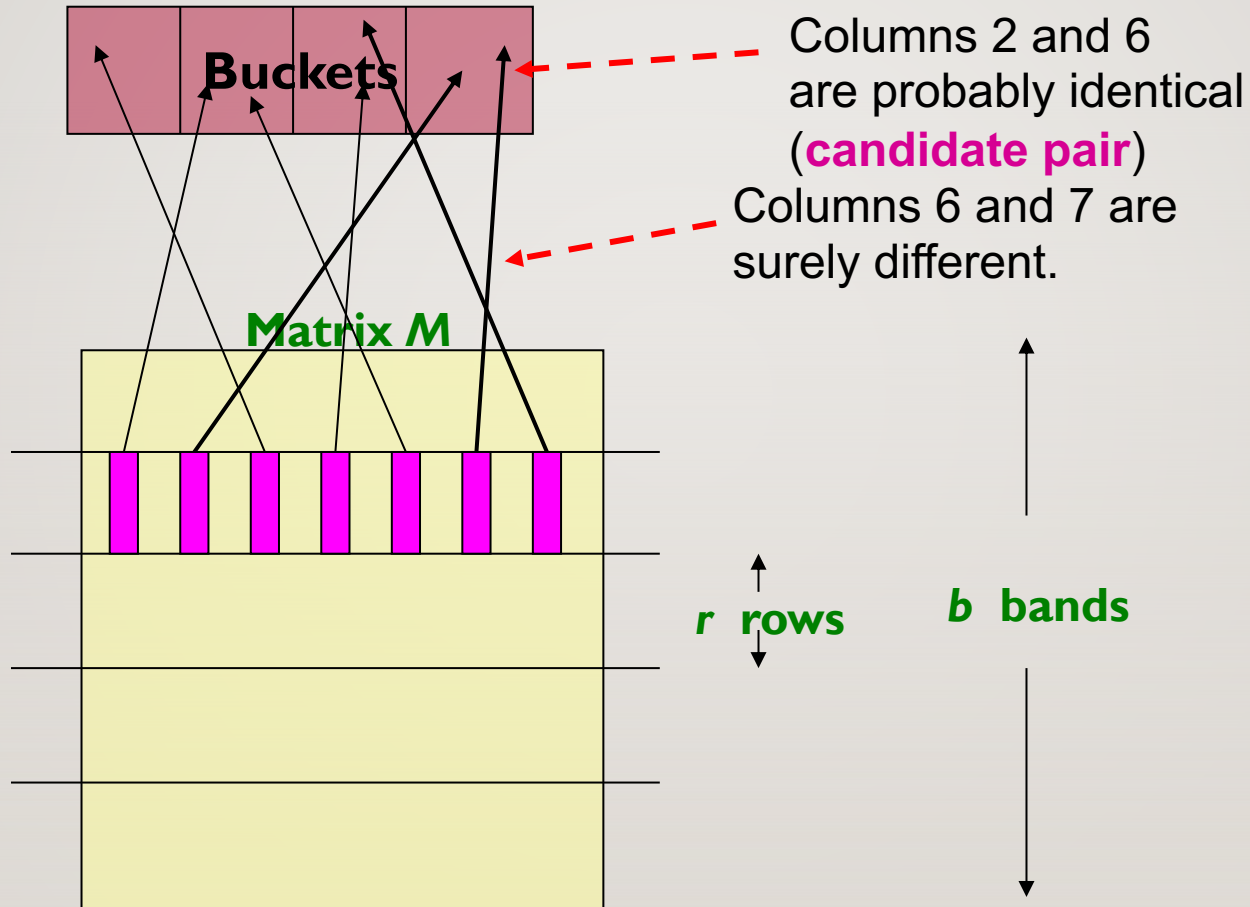




PARTITION M INTO BANDS

- 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





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 $\geq s=0.8$ similarity, set $b=20, r=5$
- **Assume:** $\text{sim}(C_1, C_2) = 0.8$
 - Since $\text{sim}(C_1, C_2) \geq s$, 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

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

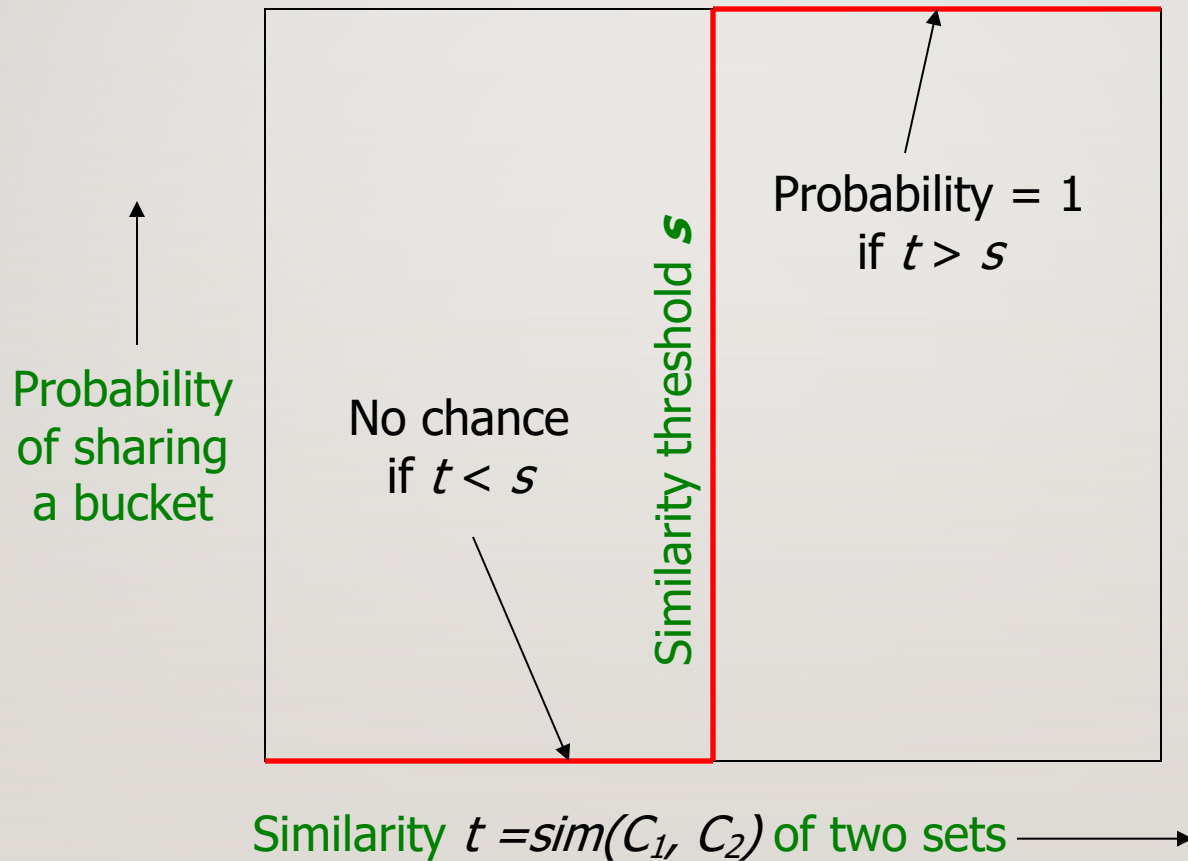
- The number of Min-Hashes (rows of M)
- The number of bands b , and
- The number of rows r per band

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

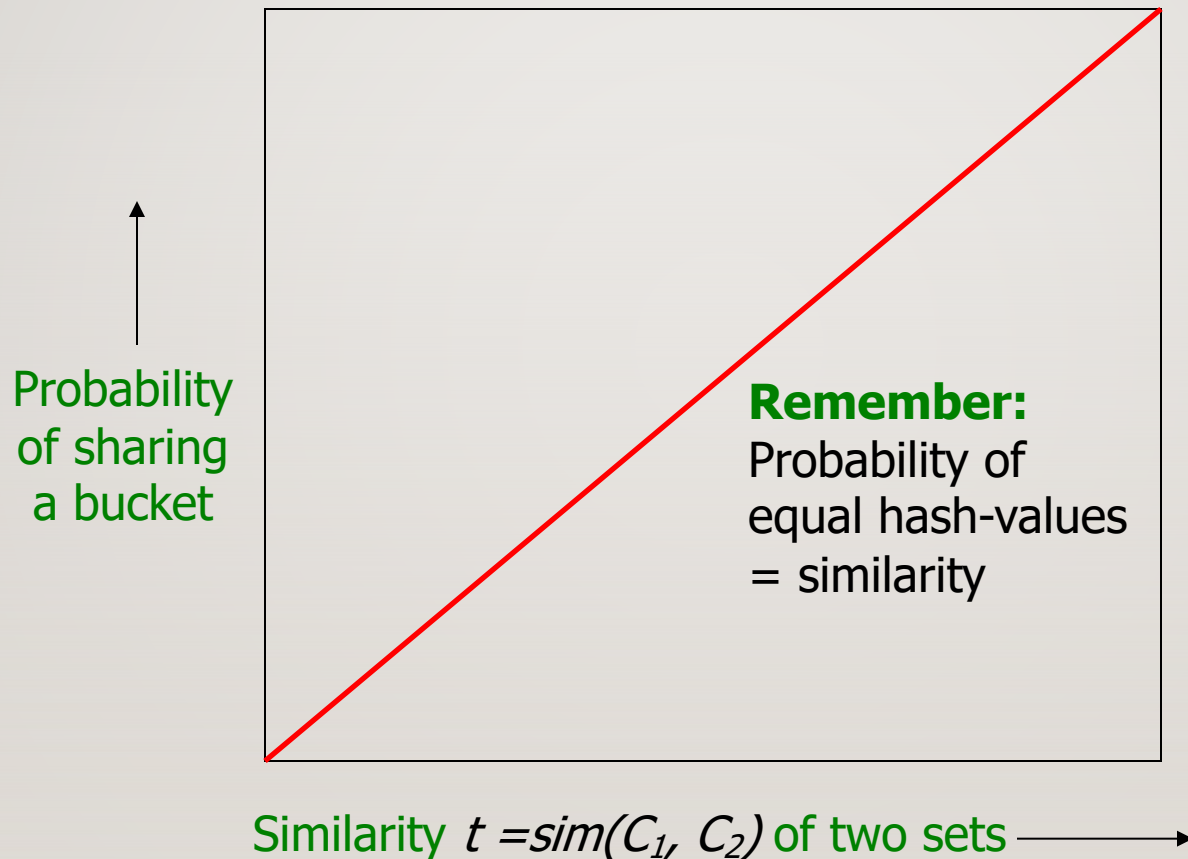


ANALYSIS OF LSH – WHAT WE WANT





WHAT 1 BAND OF 1 ROW GIVES YOU



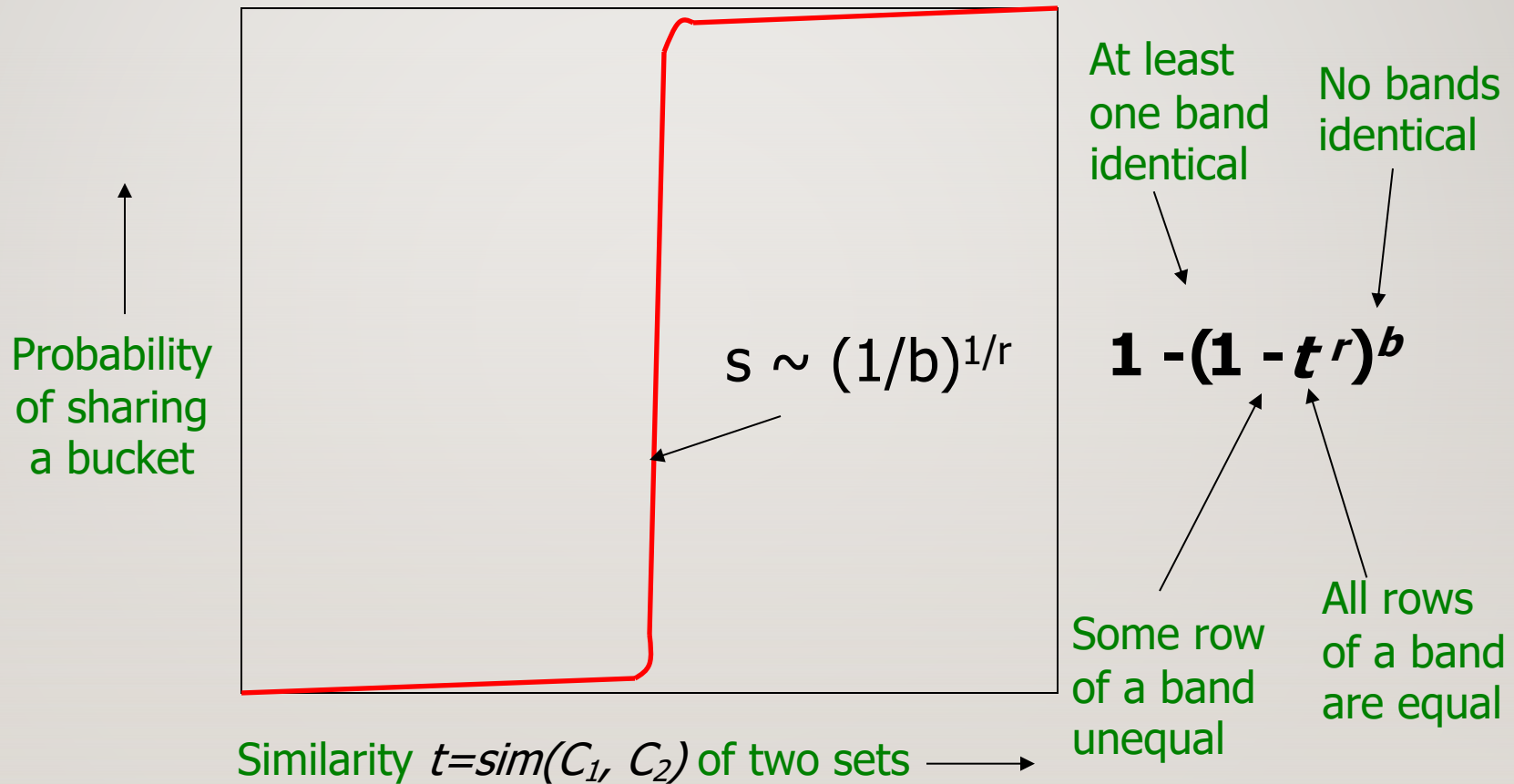


B BANDS, R ROWS/BAND

- 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: $B = 20; R = 5$

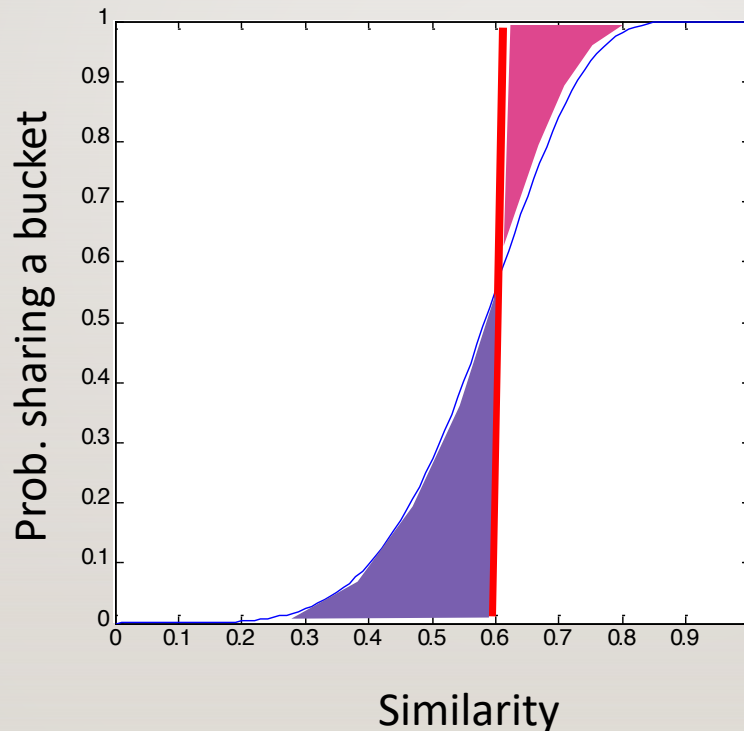
- **Similarity threshold s**
- **Prob. that at least 1 band is identical:**

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 $\Pr[h_\pi(\mathbf{C}_1) = h_\pi(\mathbf{C}_2)] = \text{sim}(\mathbf{C}_1, \mathbf{C}_2)$
 - 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 $\geq s$

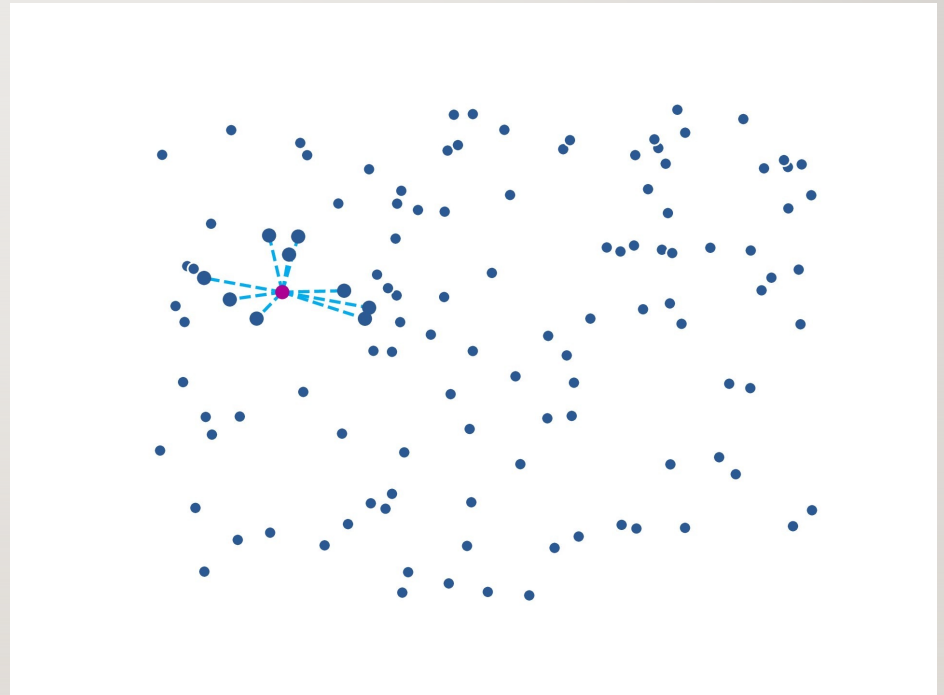


HIERARCHICAL NAVIGABLE SMALL WORLD



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**





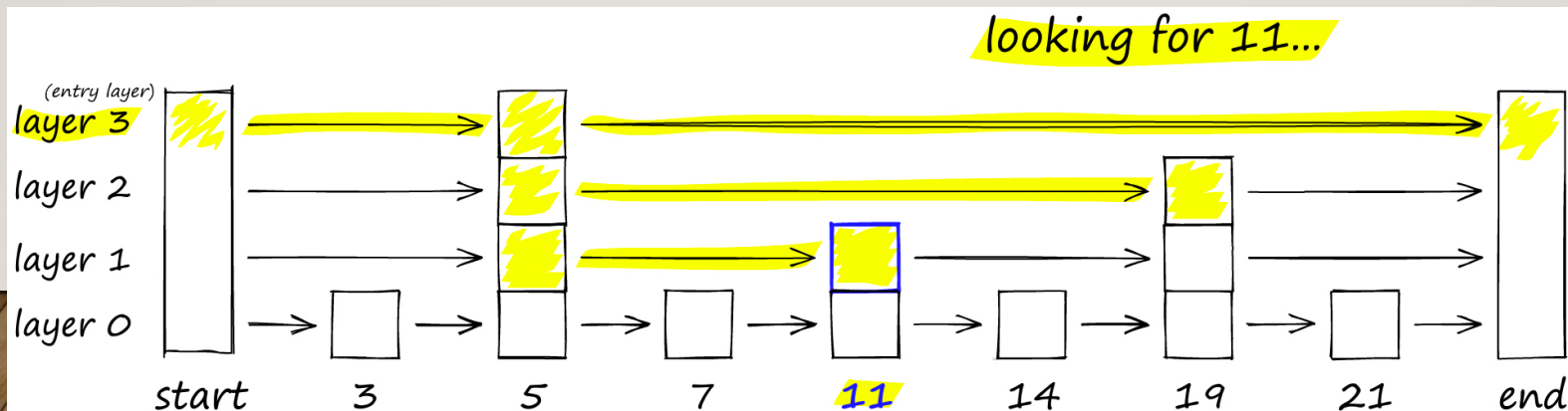
SOLUTIONS

- 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.





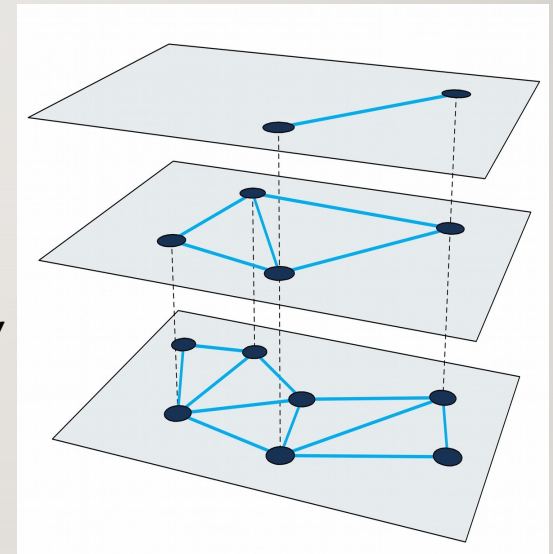
PROXIMITY GRAPH

- 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($hns w, q, K, ef$)

Input: multilayer graph $hns w$, query element q , number of nearest neighbors to return K , size of the dynamic candidate list ef

Output: K nearest elements to q

```
1  $W \leftarrow \emptyset$  // set for the current nearest elements
2  $ep \leftarrow$  get enter point for  $hns w$ 
3  $L \leftarrow$  level of  $ep$  // top layer for  $hns w$ 
4 for  $l_c \leftarrow L \dots 1$ 
5    $W \leftarrow$  SEARCH-LAYER( $q, ep, ef=1, l_c$ )
6    $ep \leftarrow$  get nearest element from  $W$  to  $q$ 
7  $W \leftarrow$  SEARCH-LAYER( $q, ep, ef, l_c=0$ )
8 return  $K$  nearest elements from  $W$  to  $q$ 
```

Algorithm 2

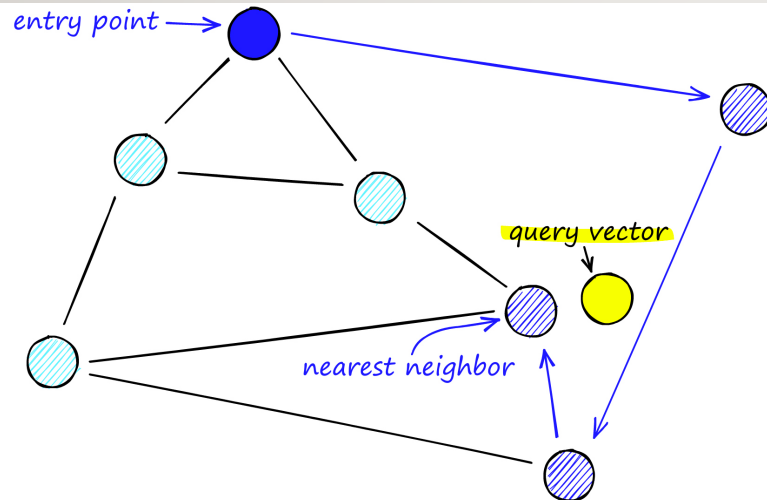
SEARCH-LAYER(q, ep, ef, l_c)

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  $v \leftarrow ep$  // set of visited elements
2  $C \leftarrow ep$  // set of candidates
3  $W \leftarrow ep$  // dynamic list of found nearest neighbors
4 while  $|C| > 0$ 
5    $c \leftarrow$  extract nearest element from  $C$  to  $q$ 
6    $f \leftarrow$  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  $e \in neighbourhood(c)$  at layer  $l_c$  // update  $C$  and  $W$ 
10    if  $e \notin v$ 
11       $v \leftarrow v \cup e$ 
12       $f \leftarrow$  get furthest element from  $W$  to  $q$ 
13      if  $distance(e, q) < distance(f, q)$  or  $|W| < ef$ 
14         $C \leftarrow C \cup e$ 
15         $W \leftarrow W \cup e$ 
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  $v \leftarrow ep$  // set of visited elements
2  $C \leftarrow ep$  // set of candidates
3  $W \leftarrow ep$  // dynamic list of found nearest neighbors
4 while  $|C| > 0$ 
5    $c \leftarrow$  extract nearest element from  $C$  to  $q$ 
6    $f \leftarrow$  get furthest element from  $W$  to  $q$ 
7   if  $\text{distance}(c, q) > \text{distance}(f, q)$ 
8     break // all elements in  $W$  are evaluated
9   for each  $e \in \text{neighbourhood}(c)$  at layer  $l_c$  // update  $C$  and  $W$ 
10    if  $e \notin v$ 
11       $v \leftarrow v \cup e$ 
12       $f \leftarrow$  get furthest element from  $W$  to  $q$ 
13      if  $\text{distance}(e, q) < \text{distance}(f, q)$  or  $|W| < ef$ 
14         $C \leftarrow C \cup e$ 
15         $W \leftarrow W \cup e$ 
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  $ep \leftarrow$  get enter point for hnsw
3  $L \leftarrow$  level of  $ep$  // top layer for hnsw
4  $l \leftarrow \lfloor -\ln(\text{unif}(0..1)) \cdot m_L \rfloor$  // new element's level
5 for  $l_c \leftarrow L \dots l+1$ 
6    $W \leftarrow$  SEARCH-LAYER(q, ep, ef=1,  $l_c$ )
7    $ep \leftarrow$  get the nearest element from W to q
```

```
8 for  $l_c \leftarrow \min(L, l) \dots 0$ 
9    $W \leftarrow$  SEARCH-LAYER(q, ep, efConstruction,  $l_c$ )
10   $neighbors \leftarrow$  SELECT-NEIGHBORS(q, W, M,  $l_c$ ) // alg. 3 or alg. 4
11  add bidirectional connections from  $neighbors$  to q at layer  $l_c$ 
12  for each  $e \in neighbors$  // shrink connections if needed
13     $eConn \leftarrow$  neighbourhood(e) at layer  $l_c$ 
14    if  $|eConn| > M_{max}$  // shrink connections of e
15      // if  $l_c = 0$  then  $M_{max} = M_{max0}$ 
16       $eNewConn \leftarrow$  SELECT-NEIGHBORS(e,  $eConn$ ,  $M_{max}$ ,  $l_c$ )
17      // alg. 3 or alg. 4
18      set neighbourhood(e) at layer  $l_c$  to  $eNewConn$ 
19   $ep \leftarrow W$ 
20 if  $l > L$ 
21   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.



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/>