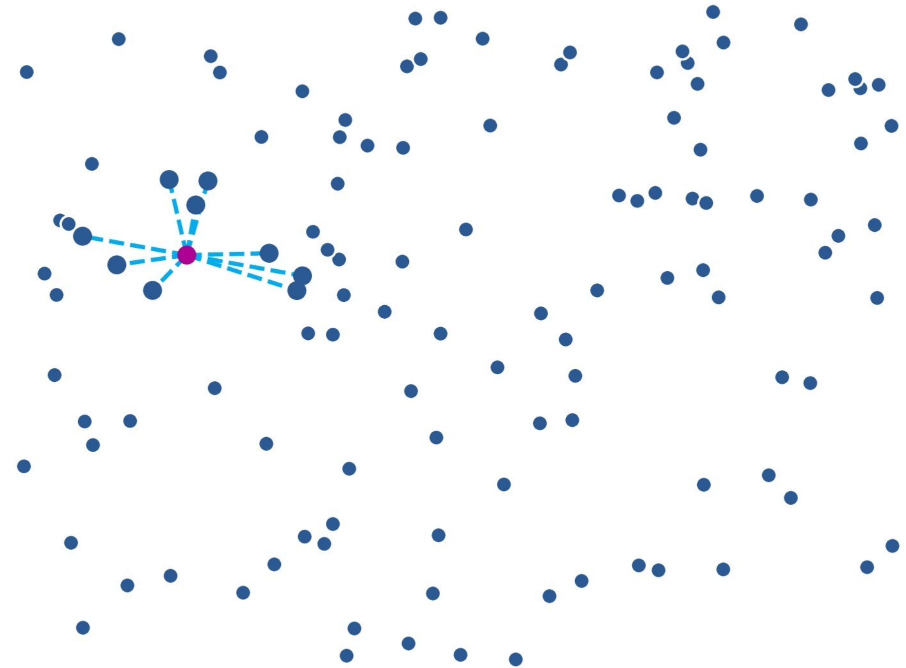
Scalable Data Mining

Sourangshu Bhattacharya

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

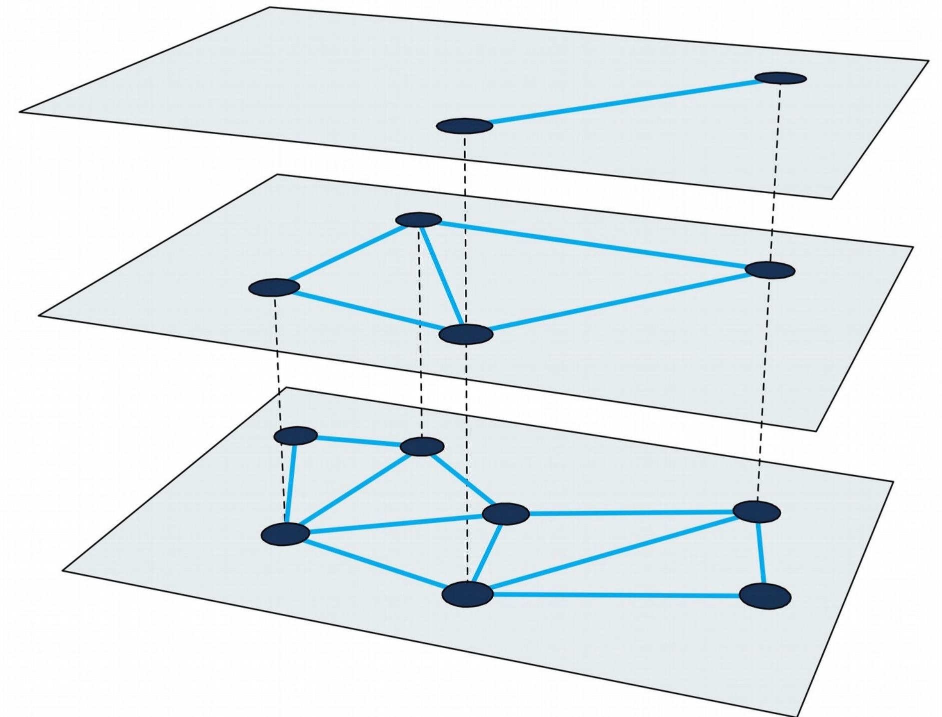
- For index compression (not discussed):
  - Product quantization.

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 $W \leftarrow \emptyset$ // set for the current nearest elements
2 $ep \leftarrow$ get enter point for *hnsw*
3 $L \leftarrow$ level of *ep* // top layer for *hnsw*
4 **for** $l_c \leftarrow L \ldots 1$
5 $\quad W \leftarrow$ SEARCH-LAYER(q, ep, ef=1, l_c)
6 $\quad ep \leftarrow$ 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 $\quad c \leftarrow$ extract nearest element from C to q
6 $\quad f \leftarrow$ get furthest element from W to q
7 $\quad$ **if** $distance(c, q) > distance(f, q)$
8 $\quad\quad$ **break** // all elements in W are evaluated
9 $\quad$ **for** each $e \in neighbourhood(c)$ at layer l_c // update C and W
10 $\quad\quad$ **if** $e \notin v$
11 $\quad\quad\quad v \leftarrow v \cup e$
12 $\quad\quad\quad f \leftarrow$ get furthest element from W to q
13 $\quad\quad\quad$ **if** $distance(e, q) < distance(f, q)$ or $|W| < ef$
14 $\quad\quad\quad\quad C \leftarrow C \cup e$
15 $\quad\quad\quad\quad W \leftarrow W \cup e$
16 $\quad\quad\quad\quad$ **if** $|W| > ef$
17 $\quad\quad\quad\quad\quad$ 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 ← ∅   // list for the currently found nearest elements
2  ep ← get enter point for hnsw
3  L ← level of ep   // top layer for hnsw
4  l ← ⌊-ln(unif(0..1))·mL⌋  // new element's level
5  for lc ← L … l+1
6    W ← SEARCH-LAYER(q, ep, ef=1, lc)
7    ep ← get the nearest element from W to q
8  for lc ← min(L, l) … 0
9    W ← SEARCH-LAYER(q, ep, efConstruction, lc)
10   neighbors ← SELECT-NEIGHBORS(q, W, M, lc) // alg. 3 or alg. 4
11   add bidirectionall connections from neighbors to q at layer lc
12   for each e ∈ neighbors   // shrink connections if needed
13     eConn ← neighbourhood(e) at layer lc
14     if │eConn│ > Mmax // shrink connections of e
                        // if lc = 0 then Mmax = Mmax0
15       eNewConn ← SELECT-NEIGHBORS(e, eConn, Mmax, lc)
                                              // alg. 3 or alg. 4
16       set neighbourhood(e) at layer lc to eNewConn
17   ep ← W
18 if l > L
19   set enter point for hnsw to q
```

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/