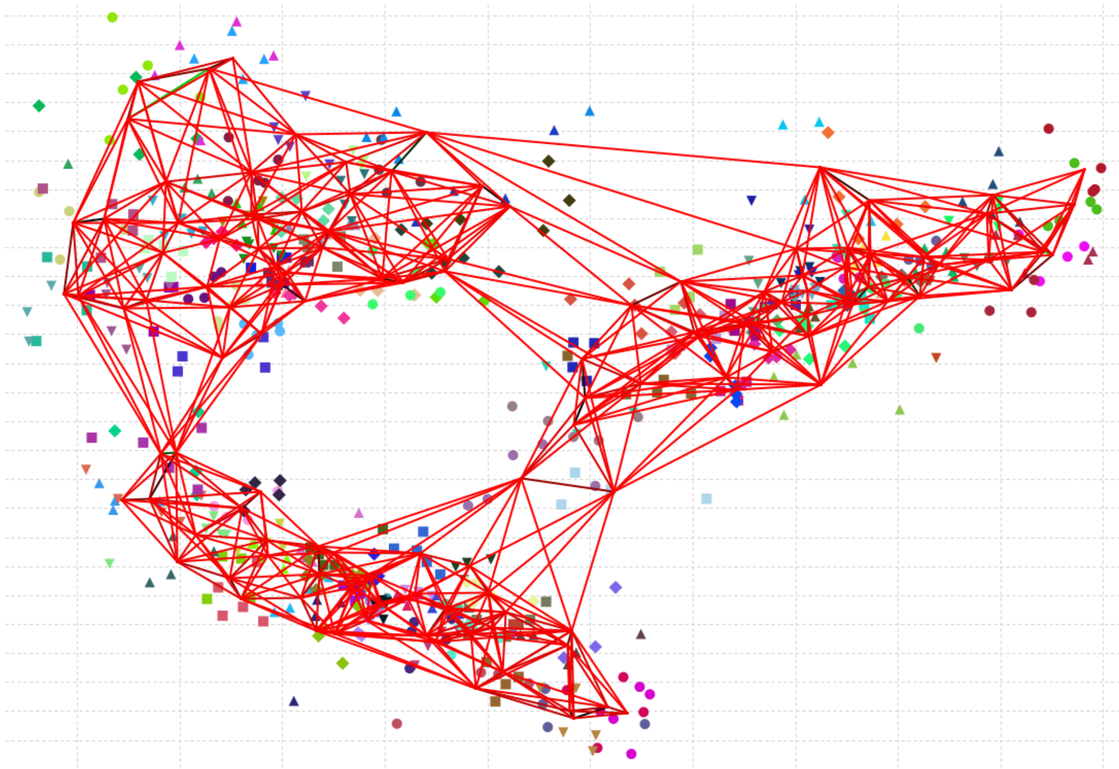
Deep dive HNSW

George Panchuk
HSE AI Fall 2025

github.com/joein/vector-search-course

HNSW Recap

- Represent vectors as nodes in a proximity graph
- Search by walking the graph, moving toward closer neighbours
- Greedy search explores only a small part of the graph
- Works well in high dimensions, unlike trees
- High memory usage (full vectors + graph links)
- Might be slow or expensive to build
- Examples: HNSW, NSG, Vamana (DiskANN)



HNSW Recap

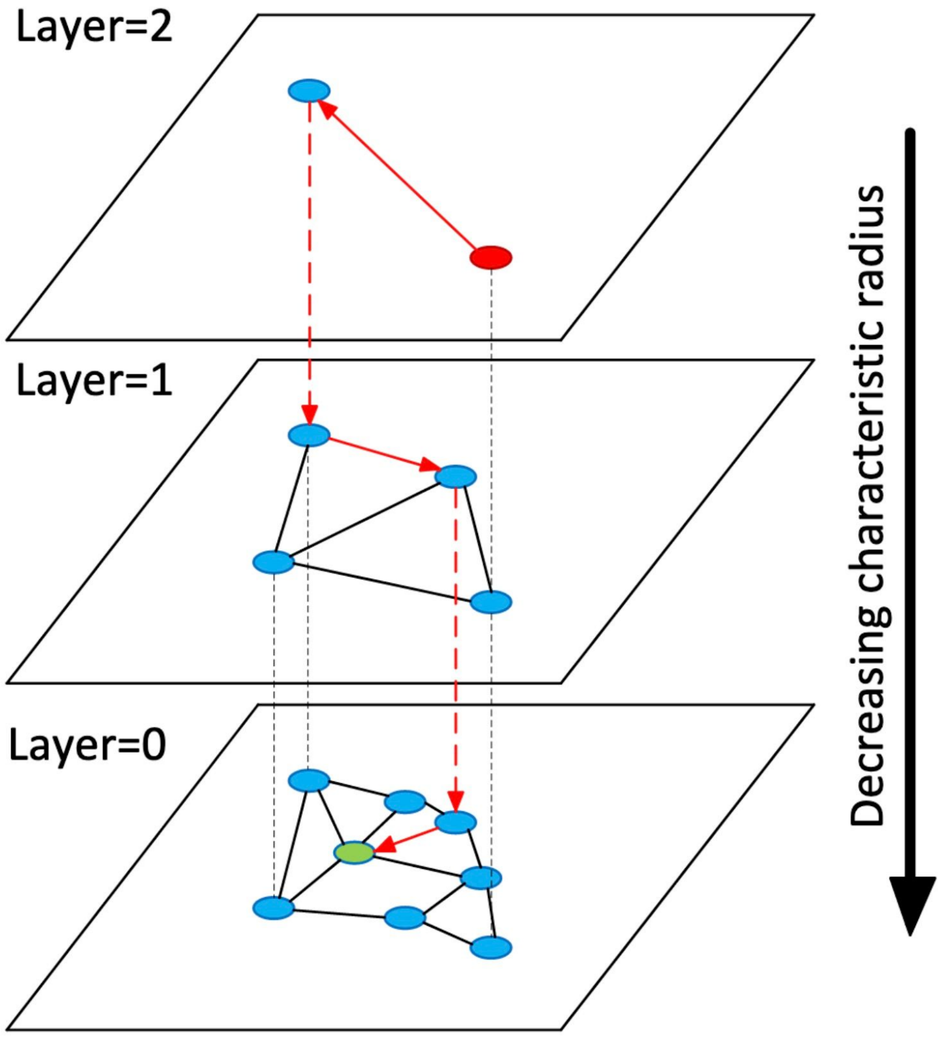
Overview

Idea:

- Build a multi-layer proximity graph where higher layers provide long-range jumps, and lower layers provide fine-grained local neighbours.
- Search starts at the top and descends, always moving toward closer nodes

Key properties:

- Extremely fast, low-latency search
- Can achieve higher recall than the other non-graph based methods
- Works well in high-dimensional spaces

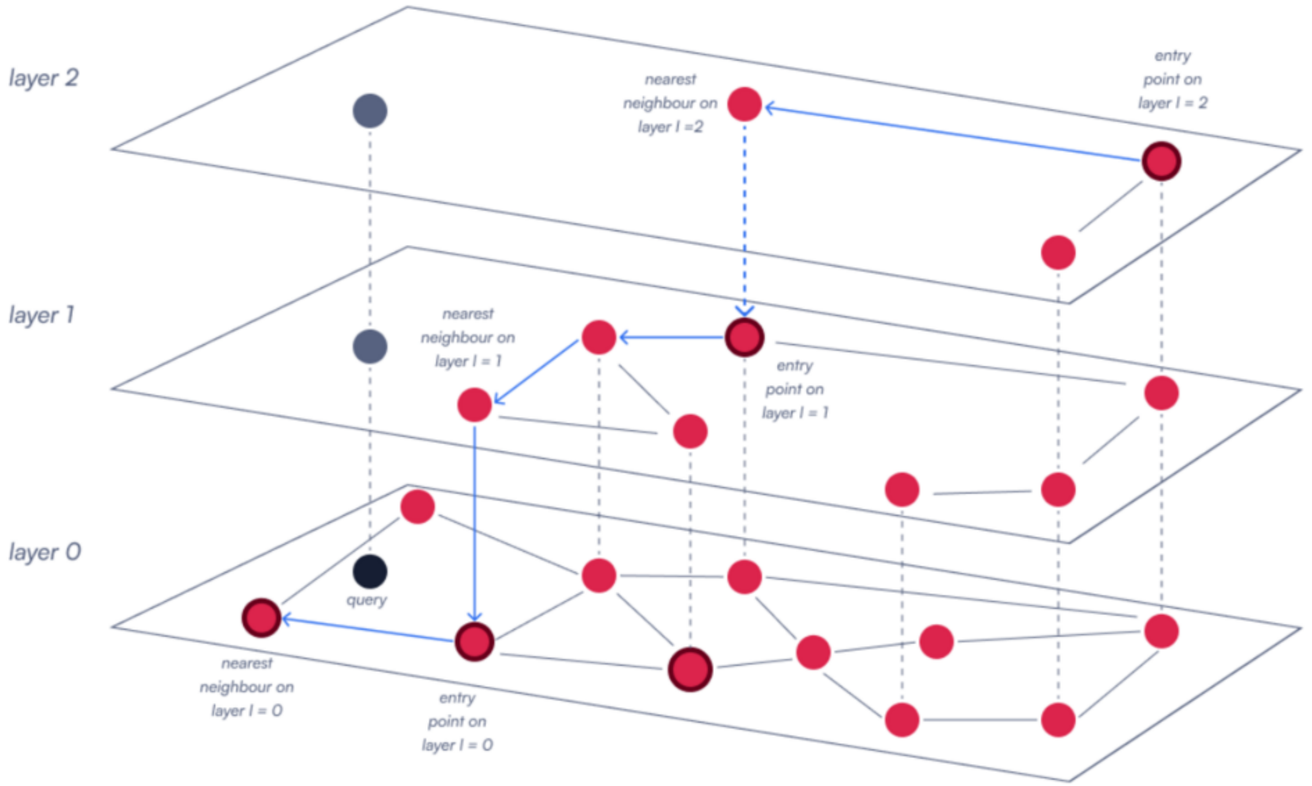


Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs

HNSW

Construction

- Assign each point a random maximum layer l by using $\lfloor -ln(unif(0;1)) * m_L \rfloor$, if l is higher than current max level L, set points as an entry point for HNSW
- m_L (and M_{max0}) help to maintain the small world navigability
- Insert top-down: greedy search at each level until nearest entry point is found, on layers L to l+1 - with *ef=1, and ef=efConstruction* on layers from l to 0
- Insert a node on each level starting from *l*, each node can have up to M edges
- At level 0, M_{max0} is used instead of M, practically $M_{max0} = 2 \cdot M$
- Prune node connections if it has more than specified M number

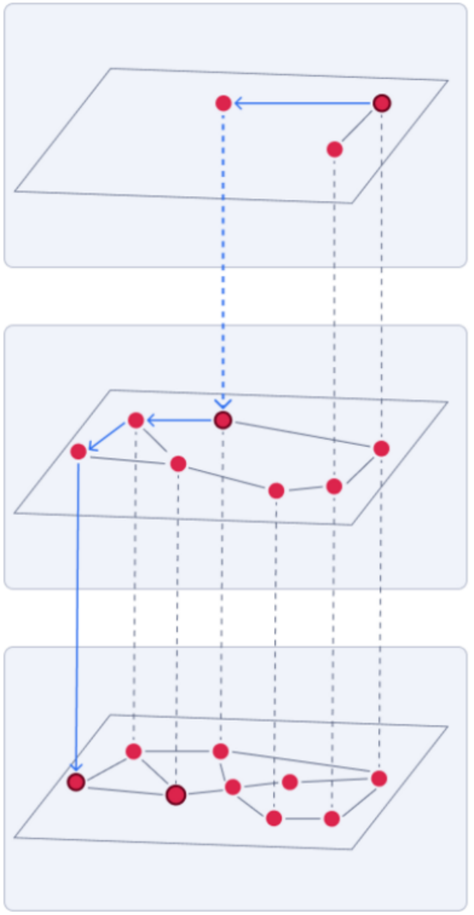


HNSW indexing fundamentals

HNSW

Search

- Start at the top layer at HNSW entrypoint
- Greedy search at each level until nearest entry point is found
- Previous layers entrypoint is the entrypoint on the next level
- On each level except ground level search with *ef=1*, then with *ef=efSearch*
- During search, keep a list of visited nodes, as well as dynamic list of nearest neighbours
- Iterate over the neighbours, if a neighbour is closer than already found nearest points, check its candidates as well
- Stop when there are no closer points in a candidates set than already selected ones



m (Edges per Node)

This controls the number of edges in the graph. A higher value enhances search accuracy but demands more memory and build time. Fine-tune this to balance memory usage and precision.

ef_construct (Index Build Range)

This parameter sets how many neighbors are considered during index construction. A larger value improves the accuracy of the index but increases the build time. Use this to customize your indexing speed versus quality.

ef (Search Range)

This determines how many neighbors are evaluated during a search query. You can adjust this to balance query speed and accuracy.

Vector Search Resource Optimisation Guide

HNSW

Pseudocode

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 connectionts 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
```

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
```

Efficient and robust ANN search using HNSW

HNSW

Pseudocode

Algorithm 3
SELECT-NEIGHBORS-SIMPLE(q, C, M)
Input: base element q, candidate elements C, number of neighbors to return M
Output: M nearest elements to q
return M nearest elements from C to q

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
```

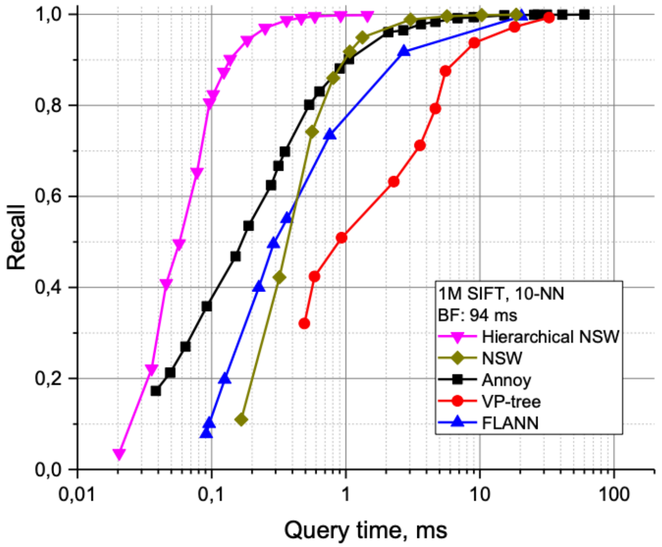
Algorithm 4
SELECT-NEIGHBORS-HEURISTIC(q, C, M, l_c, *extendCandidates*, *keepPrunedConnections*)
Input: base element q, candidate elements C, number of neighbors to return M, layer number l_c, flag indicating whether or not to extend candidate list *extendCandidates*, flag indicating whether or not to add discarded elements *keepPrunedConnections*
Output: M elements selected by the heuristic

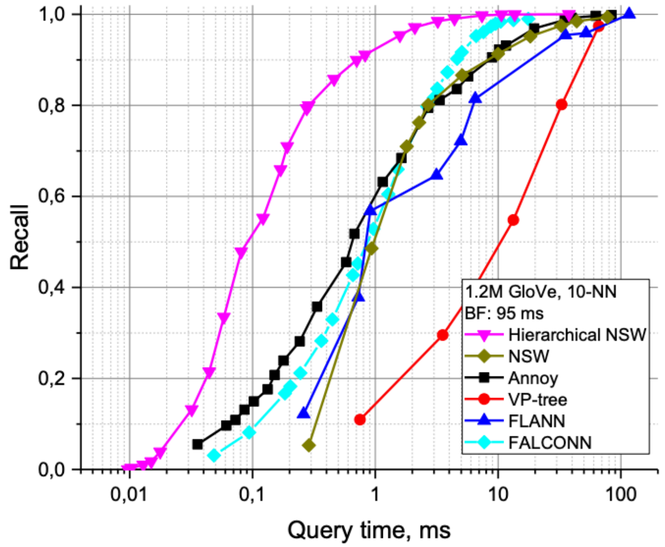
```
1  R ← ∅
2  W ← C   // working queue for the candidates
3  if extendCandidates   // extend candidates by their neighbors
4    for each e ∈ C
5      for each e_adj ∈ neighbourhood(e) at layer l_c
6        if e_adj ∉ W
7          W ← W ⋃ e_adj
8  W_d ← ∅   // queue for the discarded candidates
9  while |W| > 0 and |R| < M
10   e ← extract nearest element from W to q
11   if e is closer to q compared to any element from R
12     R ← R ⋃ e
13   else
14     W_d ← W_d ⋃ e
15 if keepPrunedConnections  // add some of the discarded
                             // connections from W_d
16   while |W_d| > 0 and |R| < M
17     R ← R ⋃ extract nearest element from W_d to q
18 return R
```

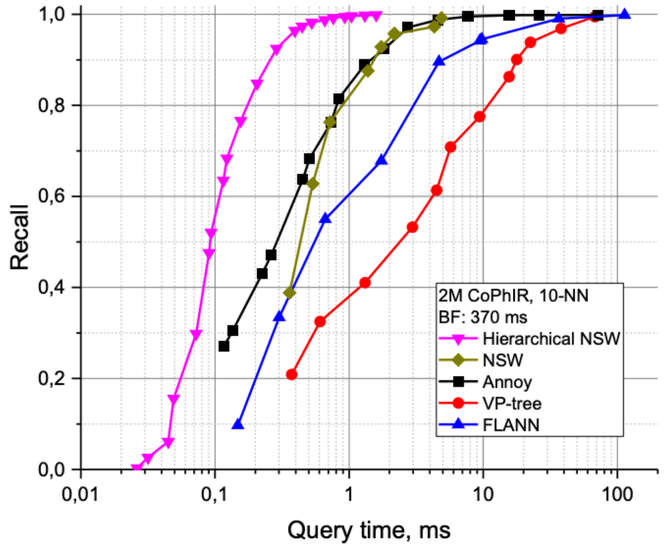
- extendCandidates (set to false by default) extends the candidate set and useful only for extremely clustered data
- keepPrunedConnections allows getting fixed number of connection per element

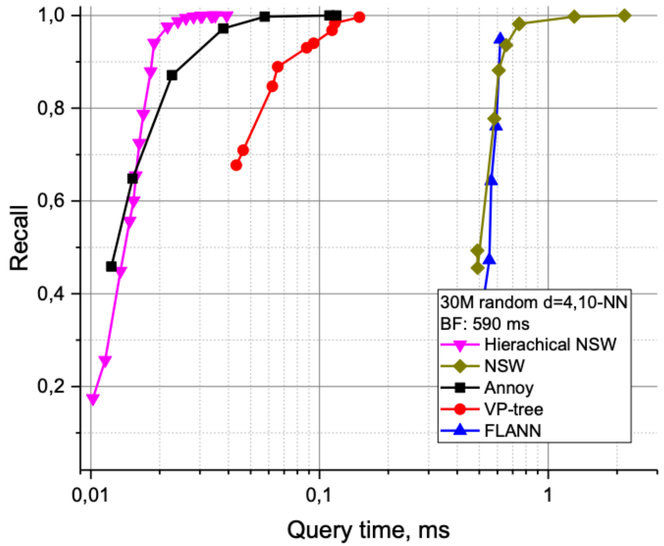
Efficient and robust ANN search using HNSW

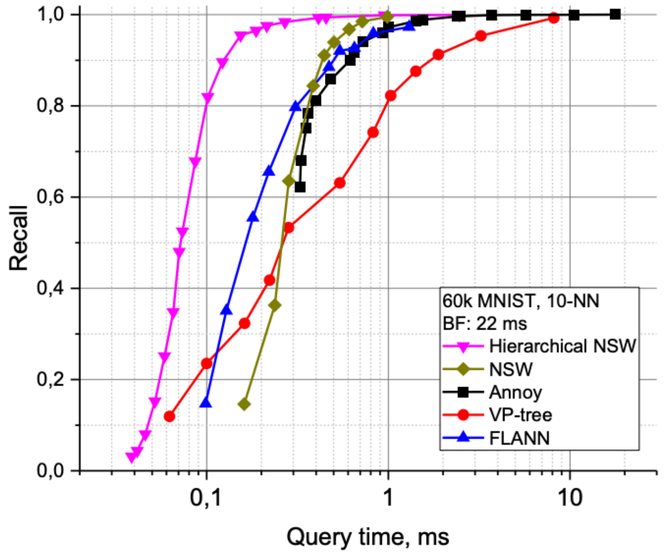
Query time & Recall comparison

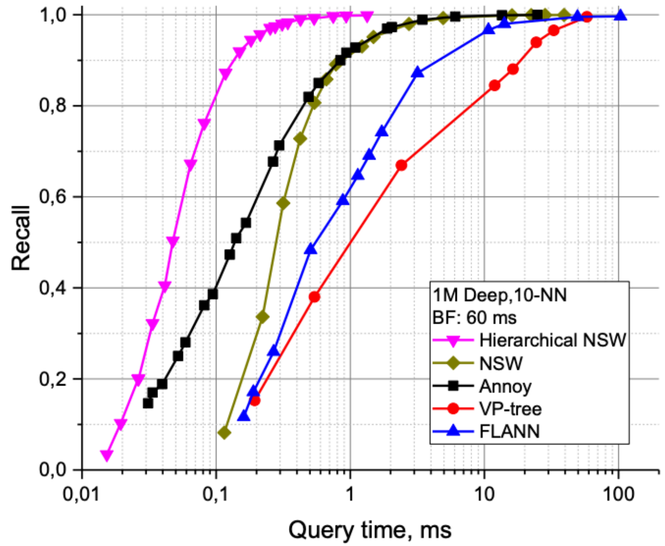












Hyperparameter benchmark

- *m* - number of edges per node, the larger the value, the higher the precision, but more space required
- *efConstruction* - number of neighbours to consider during the index building, the larger the value, the higher the precision, but the longer the indexing time
- *efSearch* - number of neighbours to consider during the search, the larger the value, the higher the precision, but the longer the search time

HNSW and payload filtering

Why filtering is not trivial?

Not many ANN algorithms are compatible with filtering.

HNSW is one of the few of them, but search engines approach its integration in different ways

- Some use *post-filtering*, which applies filters after ANN search. It doesn't scale well as it either loses results or requires many candidates on the first stage.
- Other use *pre-filtering*, which requires a binary mask of the whole dataset to be passed into the ANN algorithm. It is also not scalable, as the mask size grows linearly with the dataset size

HNSW and payload filtering

Why filtering is not trivial?

Not many ANN algorithms are compatible with filtering.

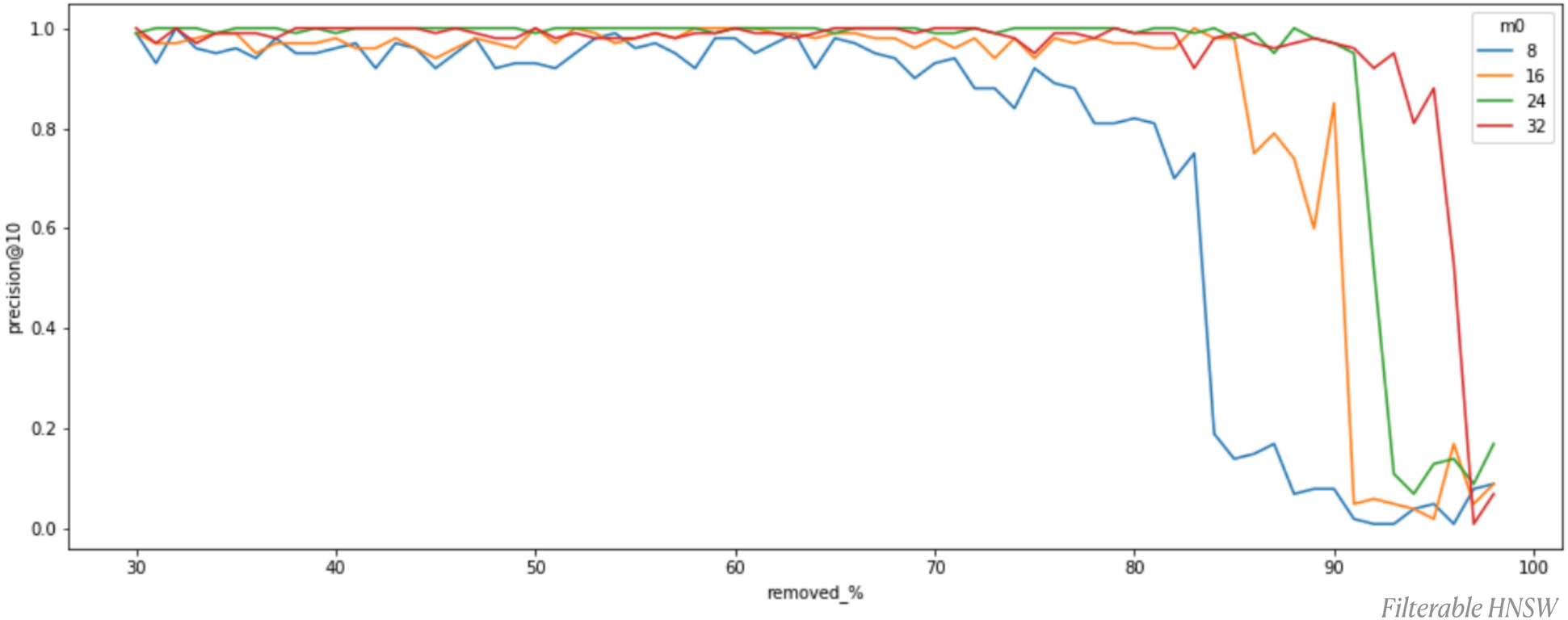
HNSW is one of the few of them, but search engines approach its integration in different ways

- Some use *post-filtering*, which applies filters after ANN search. It doesn't scale well as it either loses results or requires many candidates on the first stage.
- Other use *pre-filtering*, which requires a binary mask of the whole dataset to be passed into the ANN algorithm. It is also not scalable, as the mask size grows linearly with the dataset size

Filterable HNSW

How can we modify it?

- What if we simply apply the filter criteria to the nodes of the graph?
- It might work, if criteria do not correlate with vector semantics



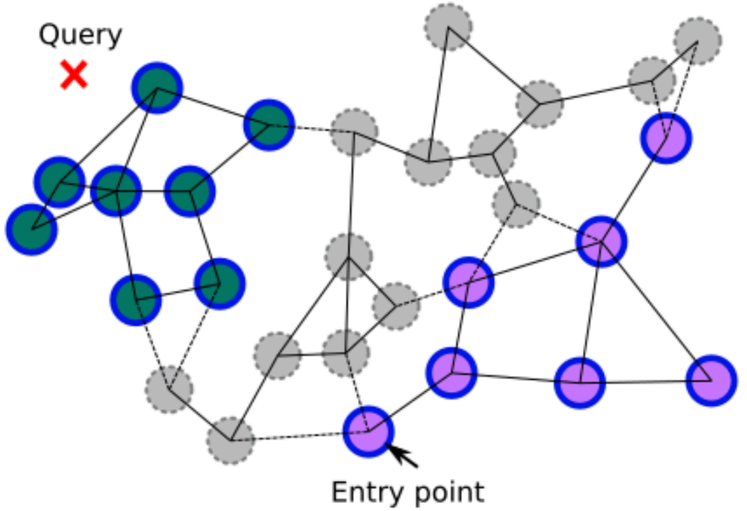
Filterable HNSW

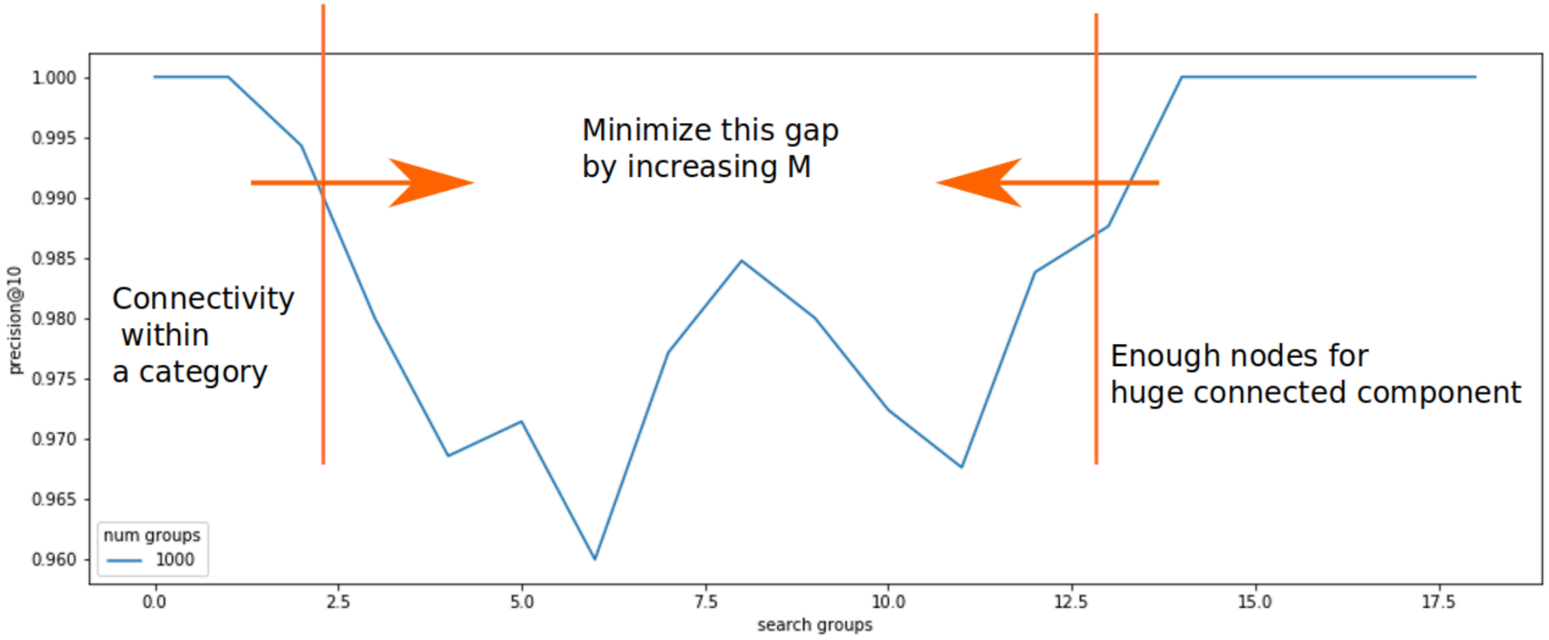
Percolation theory becomes applicable: Given a random graph of n nodes and an average degree k.

If we remove randomly a fraction 1-p of nodes and leave only a fraction p, there exists a critical percolation threshold pc = $\frac{1}{k}$ below which the network becomes fragmented, while above pc a giant connected component exists

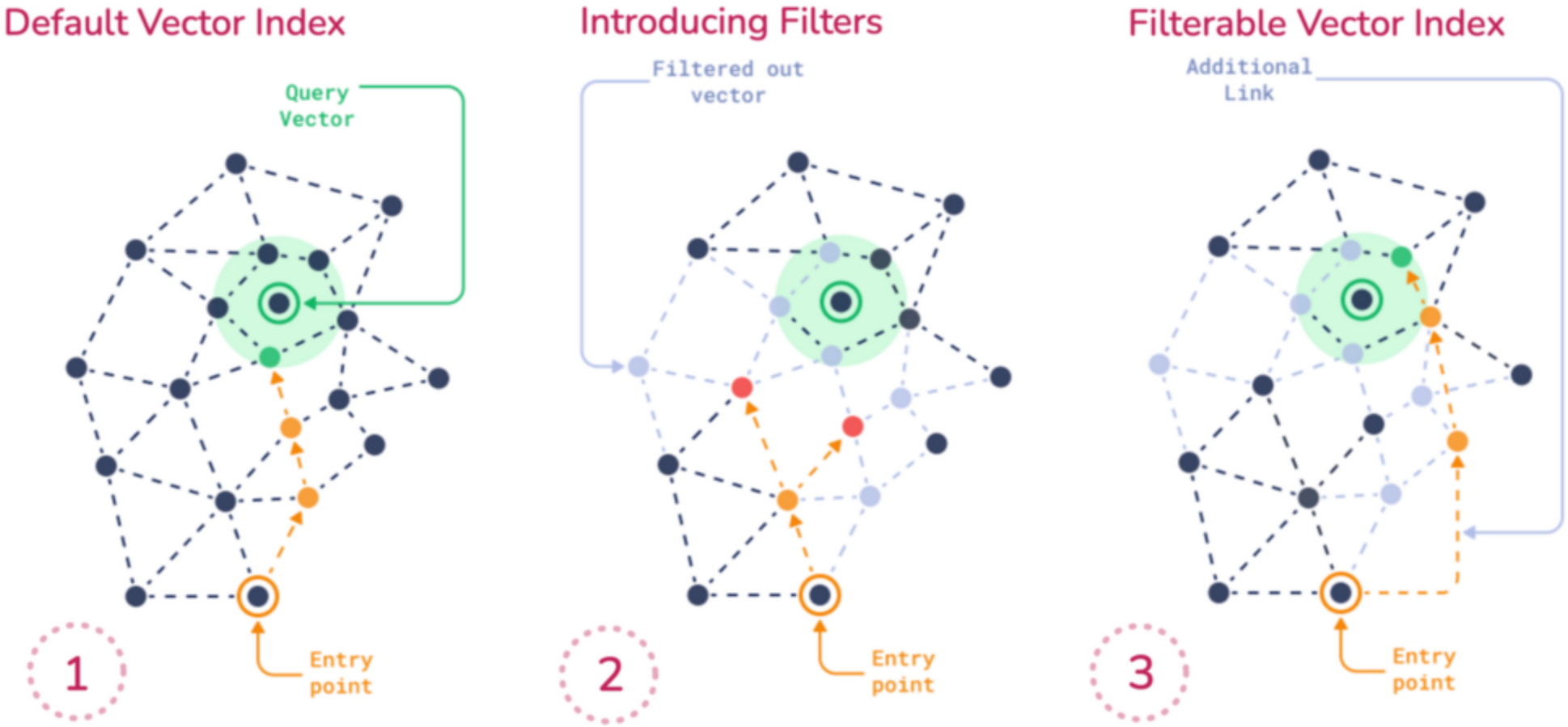
Filterable HNSW

- Filtering out nodes can make the graph disconnected
- Graph can be extended with additional links based on the payload structure (e.g. add nodes between item categories)
- Different scenarios might require different modifications
- Composite filters are difficult to handle





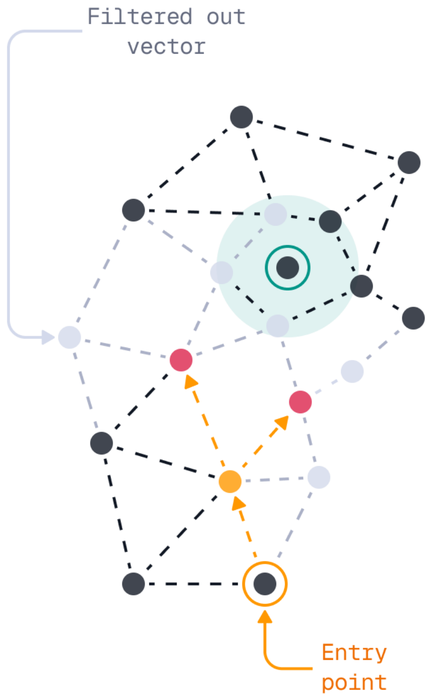
Filterable HNSW

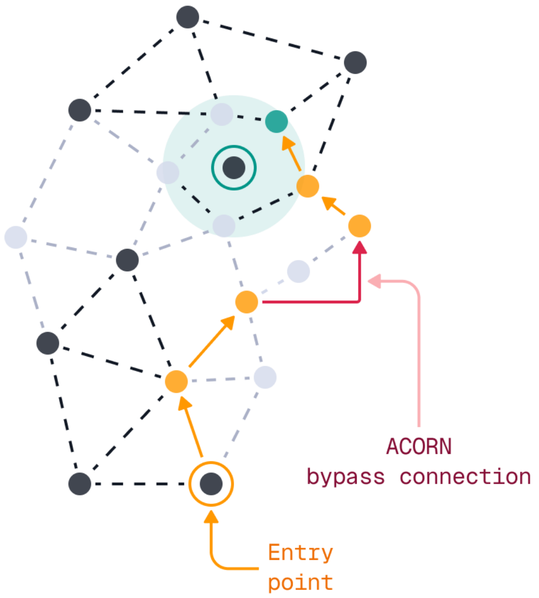


Vector search filtering

ACORN-1

- Combination of high cardinality filters can make graph disconnected
- It is impractical to build additional links for every possible combination of filters in advance
- ACORN-1 traverses not only the direct neighbours (the first hop) in the graph, but also neighbours of neighbours (the second hop)
- Improves accuracy at the cost of performance

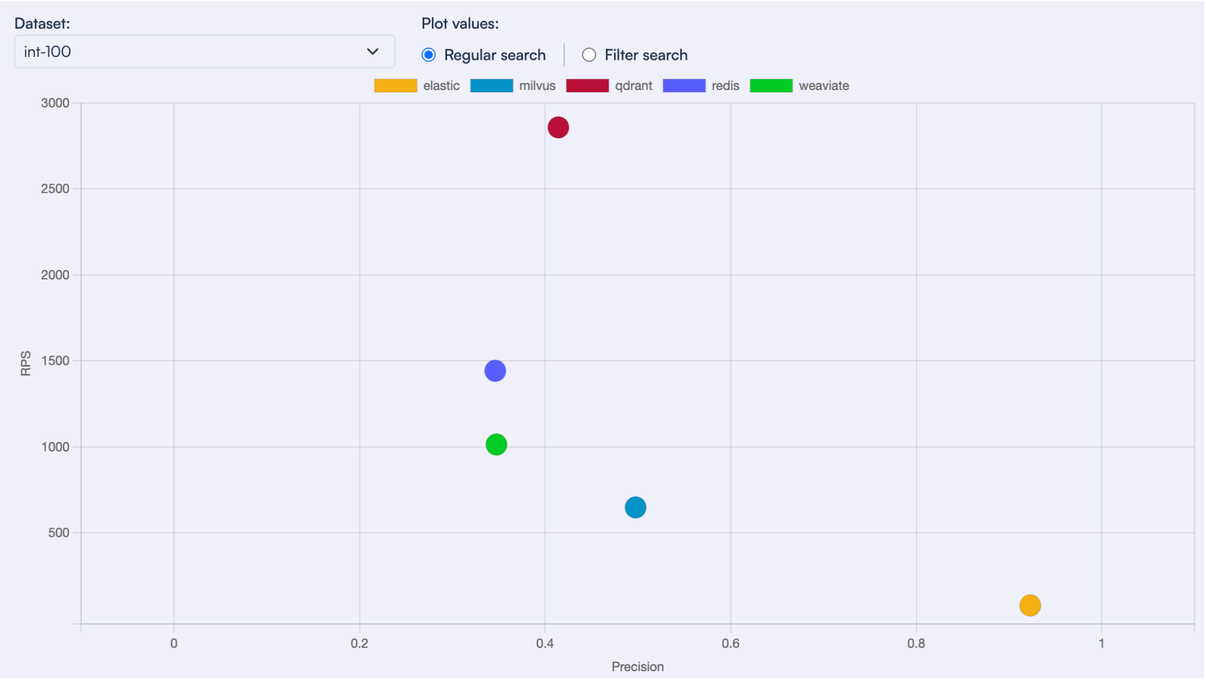


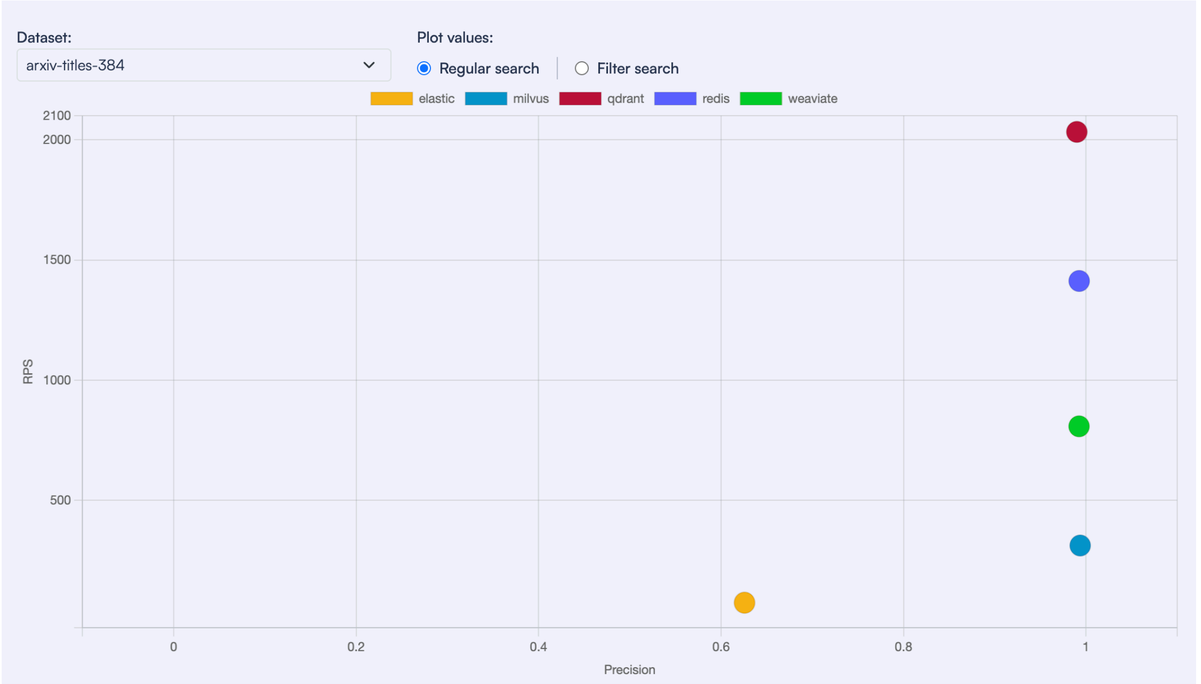


Qdrant 1.16 release blog

Benchmarks for filtered search [old]

- Algorithm performance with and without filter might be drastically different, both recall and QPS might change significantly
- There might be all possible outcomes, e.g. speed boost, which might happen when the filter is restrictive enough to avoid using ANN. Speed downturn - when building a large filtering mask is required, or accuracy collapse - when the graph becomes unconnected





Vector Database Benchmarks

Questions?