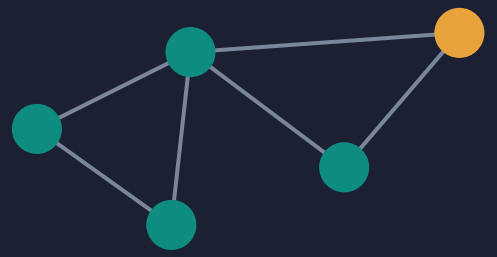
Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs

Yu. A. Malkov & D. A. Yashunin

Presenter: Yuxi Liu (Duke University)

Nov 3 2026



The problem: K-nearest neighbor search at scale

Given

- **Dataset** S of N elements, a distance δ(·,·)
- **Query** q → return the K elements of S minimizing δ(q, ·)
- *δ is not necessarily a metric: HNSW only needs to compute δ(x,y) between two elements, which sets it apart from other methods; however, the Delaunay and Relative-Neighborhood-Graph intuition, and the complexity argument are geometric. For non-metric δ, the paper's evidence is purely empirical.*

Approximate version (K-ANNS)

- **Brute force** costs O(N·d) per query; exact indexes collapse in high d
- **Allow errors,** measure them with recall@K = |found ∩ true K-NN|/K
- **Goal:** best speed / recall trade-off, plus memory and build time

94 ms

brute force per query on 1M SIFT vectors, d = 128 (Table 1)

1 ms

HNSW at recall ≥ 0.9 on the same data (Fig. 13)

O(log N)

search complexity HNSW aims for (vs. polylog for NSW)

Four families of ANN methods

1

Trees/Partition

kd-tree, VP-tree, FLANN, Annoy, IVF

Partition space; great at low d, degrade with the curse of dimensionality

2

Hashing

LSH, FALCONN

Collide nearby points; theory guarantees, but many hash loopup tables for high recall

3

Quantization

PQ, RabitQ

Compress vectors to codes; billion scale in RAM, limited recall ceiling

4

Proximity graphs

k-NN graph, NSW, HNSW, Vamana

Walk a graph toward the query; best speed/recall, memory-heavy

Families 1, 2, 4: How to narrow down candidates?
Family 3: How to store and compare vectors?

This paper: keep graphs' high-d speed, and fix their failure on low-d and clustered data, with O(log N) search

Delaunay graph: the ideal search graph

Definition

- **Voronoi cell** of p: the region closer to p than to any other data point
- **Delaunay graph:** link p and r iff their Voronoi cells share a boundary

Guarantee

- **Greedy routing always finds the exact NN:** every local minimum is the global one
- **Why:** heading from p toward q you enter a neighbor r's cell; the triangle inequality gives d(q, r) ≤ d(q, p)

Graph	Greedy → exact NN	Distances only	Degree
Delaunay	✓	✗	grows ~exp(d)
k-NN graph	✗	✓	k

Why HNSW can't just build it

1. Degree explodes with d: nearly a complete graph in high dimensions
2. Building it needs the geometry of the space, not just distances (Section 2.1)

Greedy routing and where it breaks

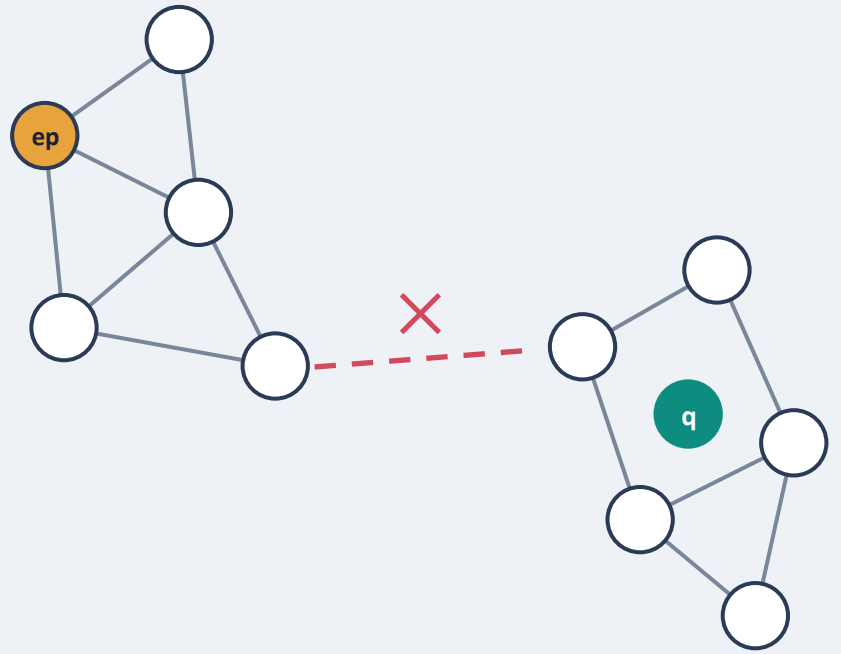
- **k-NN graph** is the usual cheap approximation. Two problems:

1. Hop count grows as a power law in N
2. Clustered data → graph loses global connectivity

Graph	Greedy → exact NN	Distances only	Degree
Delaunay	✓	✗	grows ~exp(d)
k-NN graph	✗	✓	k
HNSW layer	✗ → use ef	✓	≤ M (2M on L0)

Earlier fix: a separate coarse index (kd-tree, PQ) to pick entry points — vectors only; The upper layers of HNSW replace it!

Example: 2-NN graph on two clusters



Every point's 2 nearest neighbors are in its own cluster, so no edge crosses the gap. A greedy walk from ep can never reach q.

NSW: navigable small world graphs (the predecessor)

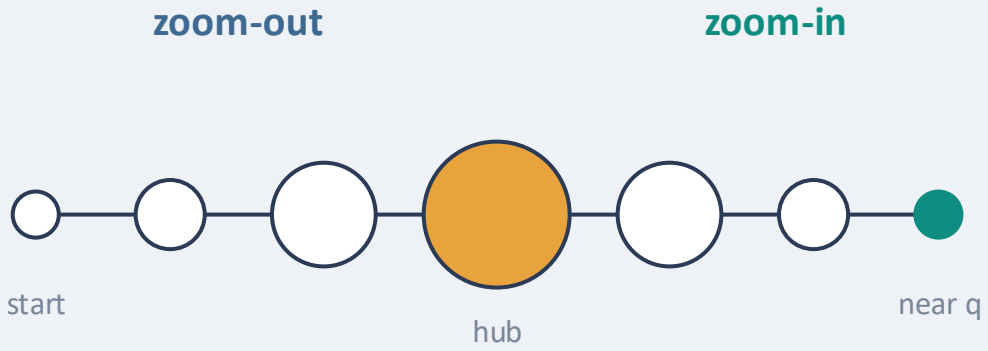
- **Build:** insert in random order; link each point both ways to its M closest already-inserted points
- **Early links** become long-range "highways" as the graph grows → navigable, no prior knowledge of the data distribution needed
- **Problem:** search is only polylogarithmic, and fails badly on low-d / clustered data

cost ≈ #hops × degree on path

$O(\log N) \times O(\log N) = O(\log^2 N)$

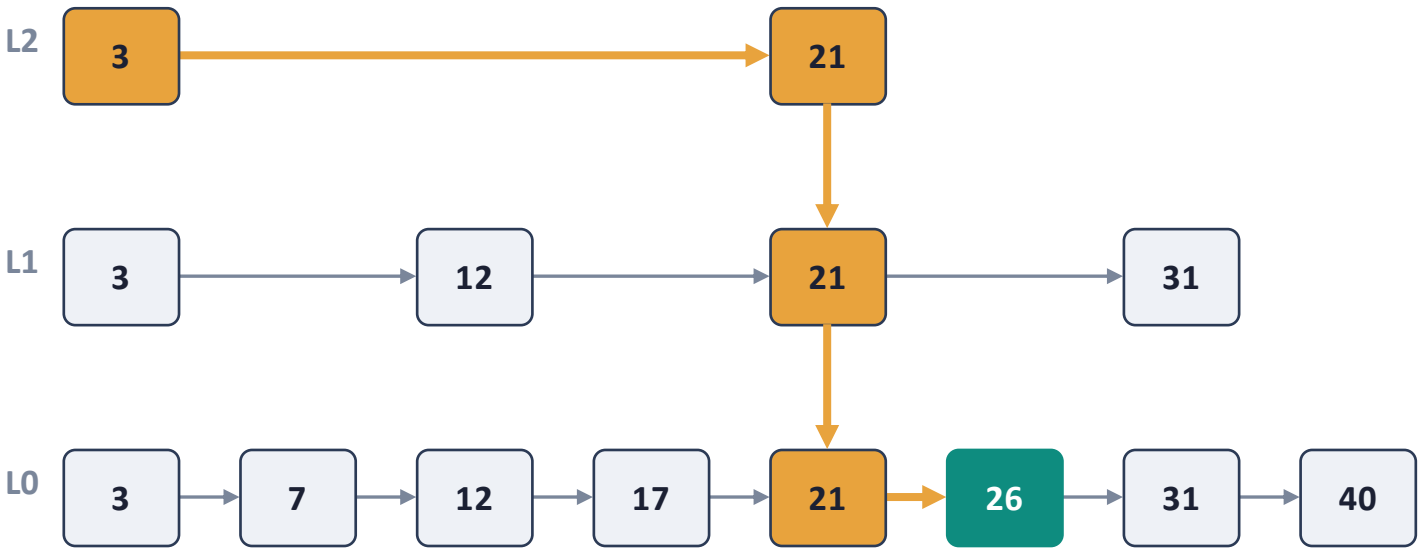
(empirical scaling, Section 3)

A greedy route in NSW (circle size ≈ node degree)



Hubs on the path get more links as N grows, so each hop costs more. HNSW's idea: keep every node's degree constant and put long links on separate layers.

Key idea: a skip list made of proximity graphs



Example: search 26 → 3 → 21 ↓ (31 > 26) ↓ → 26

Each key is promoted with probability p, so the expected search cost is O(log N)

Skip list → HNSW

- **Linked list** → proximity graph on each layer
- **Promote w.p. p** → exponentially decaying levels: P(level >= l) = p^l
- **Walk right, drop** → greedy search, drop at local minimum
- **Layer 0** holds every element
- **Top-layer head** → one fixed entry point

Layers separate links by length scale

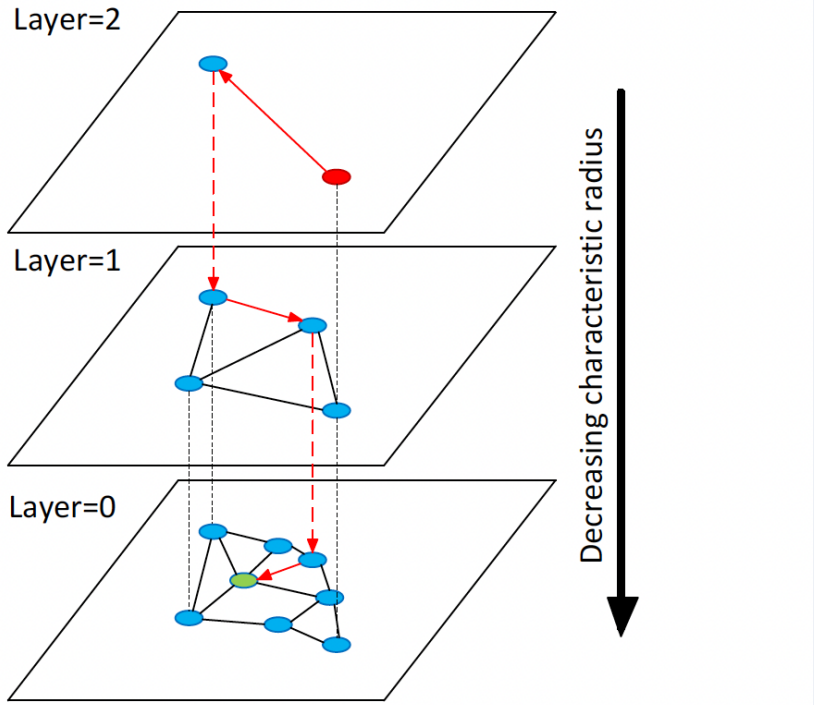


Fig. 1. Illustration of the Hierarchical NSW idea. The search starts from an element from the top layer (shown red). Red arrows show direction of the greedy algorithm from the entry point to the query (shown green).

Level of a new element

$$l = \lfloor -\ln(U) \cdot mL \rfloor, \quad mL = 1/\ln M$$

$P(l \geq k) = M^{-k}$: skip list with $p = 1/M$

Example: N = 1,000,000, M = 16

Layer	Expected nodes
5 (top)	≈ 1
4	15
3	244
2	3,906
1	62,500
0	1,000,000

SEARCH-LAYER (Alg. 2): a beam search

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
```

C: members are distance-computed and not-yet expanded; and when the distance is computed, a member is "believed to be a top-ef answer".

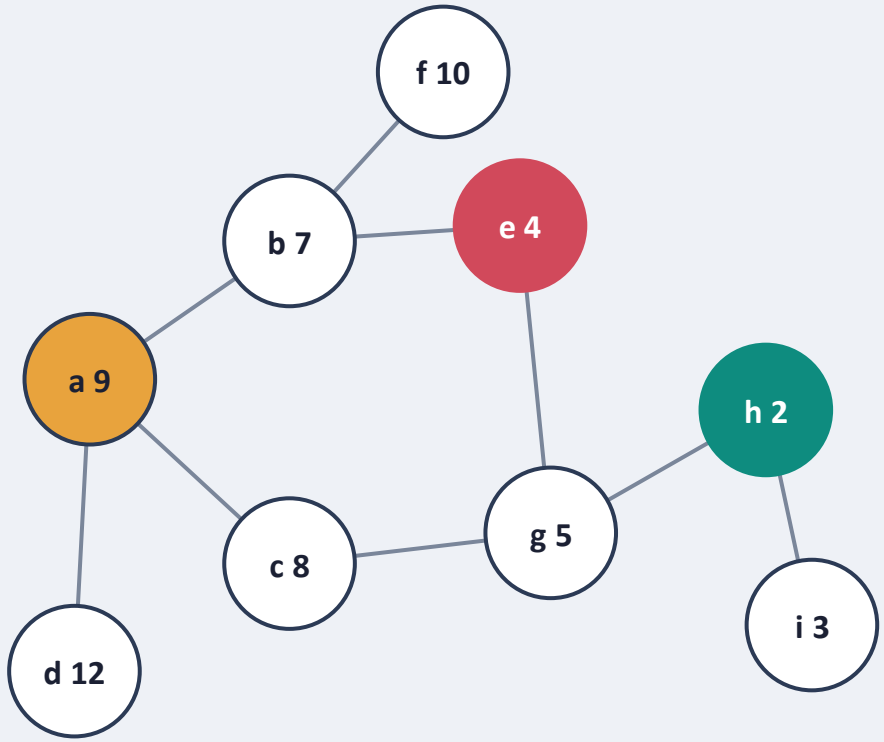
W: so far, the best top-ef answer we have found

Stop condition: the best top-ef answers (we have found so far) are all expanded

Eligible to enter both queues if "this candidate is a top-ef answer": (i) I haven't found ef candidates

Note: interestingly, |W| = ef while |C| can be > ef; despite, we are not exhausting C, but pause when W are exhausted. Actually in some current HNSW impl., the two queues are merged.

Example: why ef matters



char = node identifier, number = distance to q
entry point = a

ef = 1 (greedy)

expand	W after
a (9)	b 7
b (7)	e 4
e (4)	e 4 ← stuck, returns e ✗

ef = 3 (beam)

expand	W after (best 3)
a (9)	b 7, c 8, a 9
b (7)	e 4, b 7, c 8
e (4)	e 4, g 5, b 7 ← g kept in the beam
g (5)	h 2, e 4, g 5
h (2)	h 2, i 3, e 4
stop	next candidate c (8) > worst in W (4) → return h ✓

7 vs 9 distance evaluations: ef=1 [a, b, c, d, e, f, g]; ef=3, all.
a slightly larger beam gets past the local minimum

INSERT (Alg. 1) and K-NN-SEARCH (Alg. 5)

Layer 3	SEARCH-LAYER(ef=1)→ pass nearest down as the entry point
Layer 2	SEARCH-LAYER(ef=1)→ pass nearest down as the entry point
Layer 1	SEARCH-LAYER(ef=efConstruction) → select M neighbors → link both ways → pass W down as the entry point(s)
Layer 0	SEARCH-LAYER(ef=efConstruction) → select M neighbors → link both ways (cap 2M)

Example: inserting x with level l = 1 into an index whose top layer is L = 3

Query = insert without linking

- Layers L … l: essentially greedy descent

(in both hnswlib, Faiss, their implementations do not call SEARCH-LAYER(ef=1), but a dedicated greedy loop)

- Layer 0: SEARCH-LAYER with ef, return top K

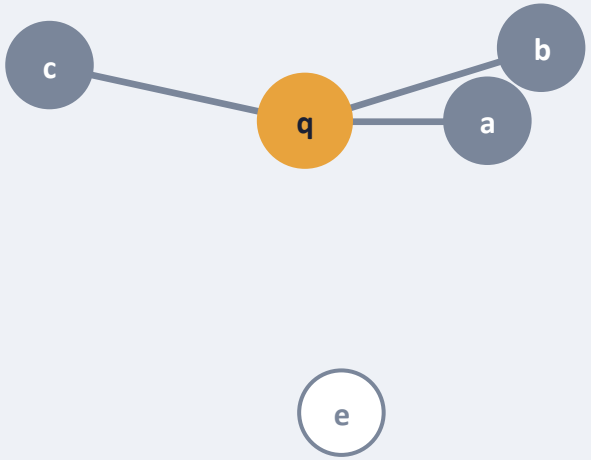
Details that matter

- The whole candidate list W is passed down as entry points
- Overfull neighbor → shrink with the same heuristic, so the graph is directed
- If l > L, x becomes the new entry point

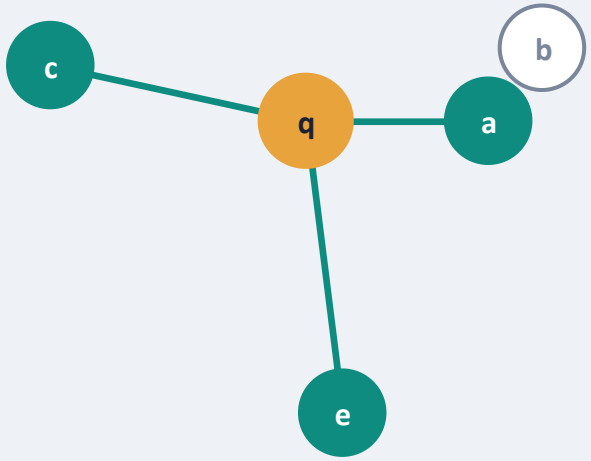
Neighbor selection: closest vs. diverse (Alg. 3 vs. Alg. 4)

efConstruction usually 200, while M is usually 32 or 64; how to select which M neighbors from the 200?

Simple: 3 closest → a, b, c



Heuristic → a, c, e



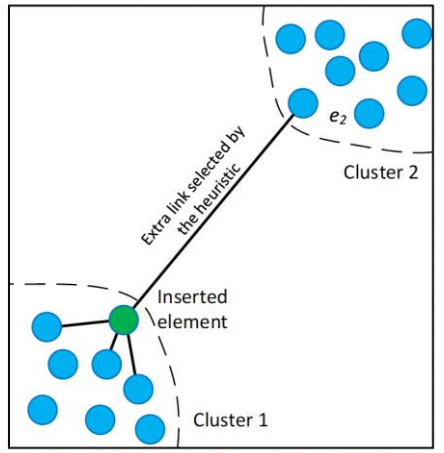


Fig. 2. Illustration of the heuristic used to select the graph neighbors for two isolated clusters. A new element is inserted on the boundary of Cluster 1. All of the closest neighbors of the element belong to the Cluster 1, thus missing the edges of Delaunay graph between the clusters. The heuristic, however, selects element e_2 from Cluster 2, thus, maintaining the global connectivity in case the inserted element is the closest to e_2 compared to any other element from Cluster 1.

Rule: scan candidates nearest-first and keep e only if d(e, q) < d(e, r) for every kept r. *(RNG-style pruning)*
Example: b is dropped because d(b,a) < d(b,q), so b is already reachable through a.
Two optional flags: (i) consider one-hop neighbors in addition; (ii) add back some discarded neighbors if < M.

Ablation study of mL choice

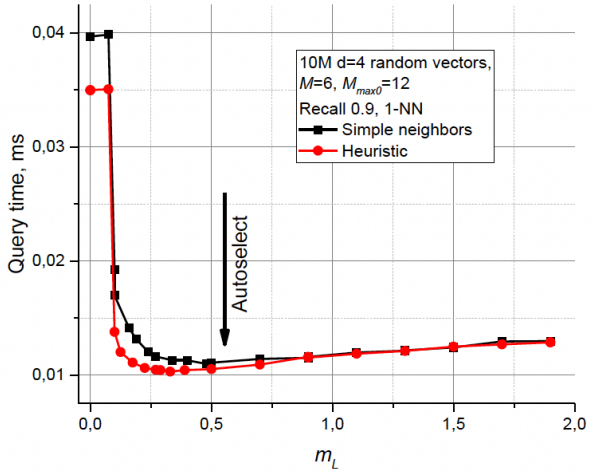


Fig. 3. Plots for query time vs m_L parameter for 10M random vectors with d=4. The autoselected value 1/ln(M) for m_L is shown by an arrow.

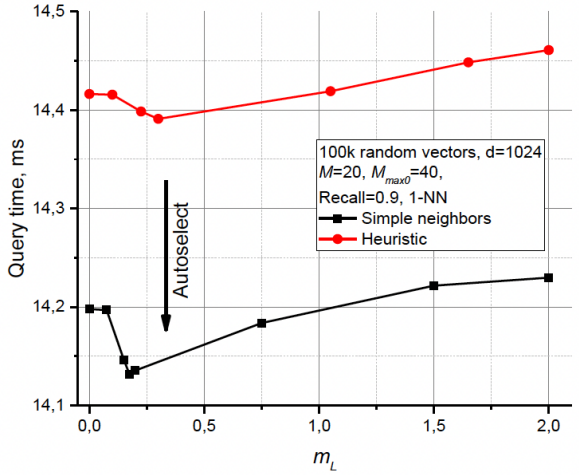


Fig. 4. Plots for query time vs m_L parameter for 100k random vectors with d=1024. The autoselected value 1/ln(M) for m_L is shown by an arrow.

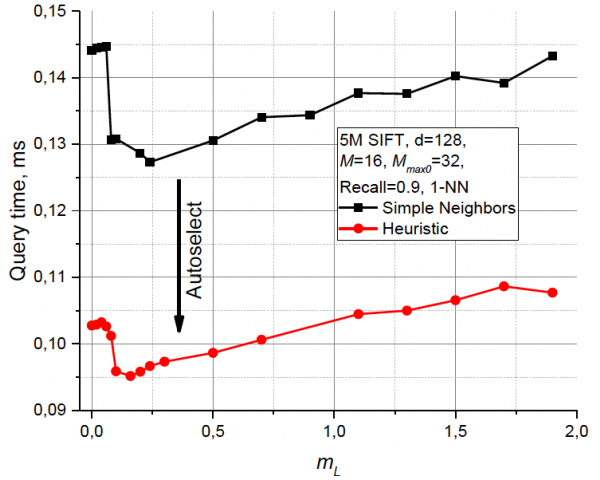


Fig. 5. Plots for query time vs m_L parameter for 5M SIFT learn dataset. The autoselected value 1/ln(M) for m_L is shown by an arrow.

1

mL = 1/ln M is close to optimal. Big win at low d, small win at high d (When mL → 0, the graph is flat as one level).

Ablation study of Mmax0 choice

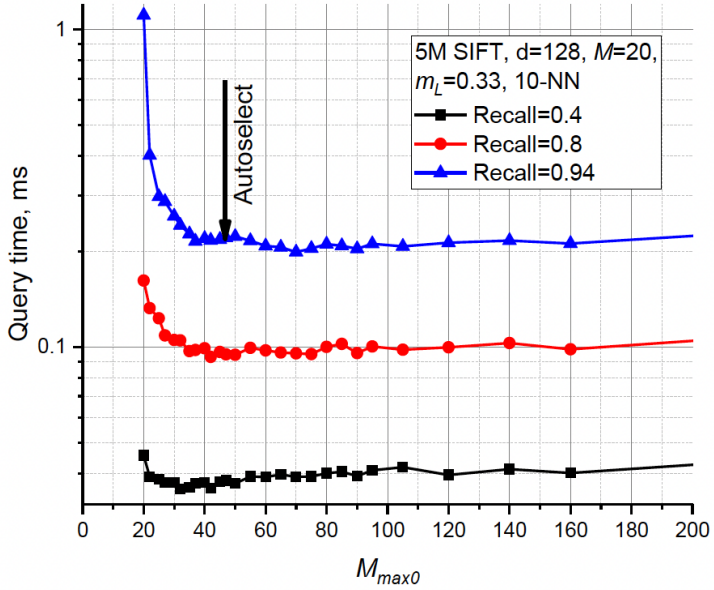


Fig. 6. Plots for query time vs M_{max0} parameter for 5M SIFT learn dataset. The autoselected value $2 \cdot M$ for M_{max0} is shown by an arrow.

2

Mmax0 = M is very slow at high recall;
2M is near optimal

Ablation study of neighbor selection

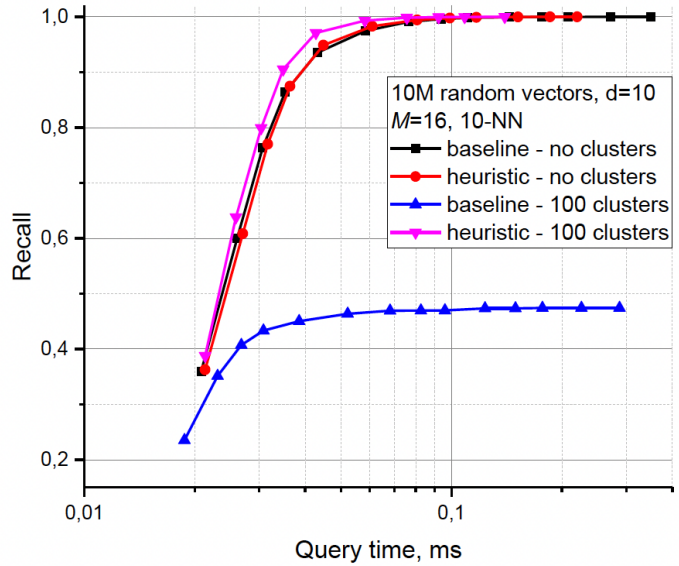


Fig. 7. Effect of the method of neighbor selections (baseline corresponds to alg. 3, heuristic to alg. 4) on clustered (100 random isolated clusters) and non-clustered d=10 random vector data.

3 Without the heuristic, recall plateaus on clustered data

Parameters: what to set, what to leave alone

Parameter	Role	Paper's advice	Evidence
M	links per new element (Mmax = M)	5 – 48; bigger for high d / high recall	Fig. 8
efConstruction	beam width during insert	big enough for ~0.95 recall at build time	Fig. 10
ef	beam width at query time (≥ K)	the speed / recall knob	Figs. 12 – 15
mL	level normalization	1/ln M (skip list p = 1/M)	Figs. 3 – 5
Mmax0	max degree on layer 0	2M	Fig. 6

Shaded rows: you tune these. Both hnswlib and Faiss libraries hard-code the others.

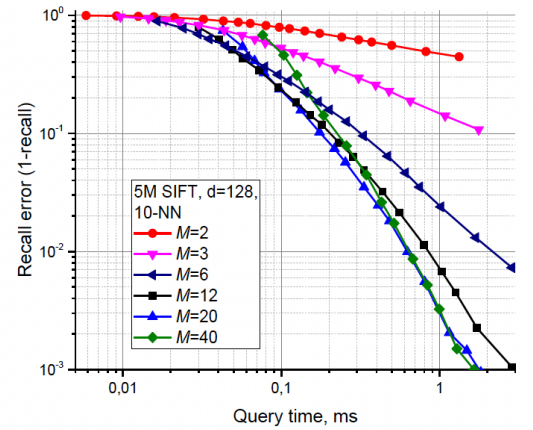


Fig. 8. Plots for recall error vs query time for different parameters of *M* for Hierarchical NSW on 5M SIFT learn dataset.

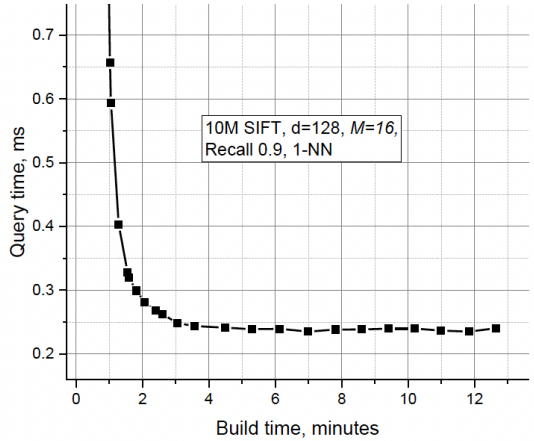


Fig. 10. Plots of the query time vs construction time tradeoff for Hierarchical NSW on 10M SIFT dataset.

Search and build cost

Assumptions (§4.2.1)

- **A1** each layer is an exact Delaunay graph → greedy finds its true nearest node
- **A2** Delaunay degree ≤ a constant C (Section 4.2.1, random Euclid data, but can be in principle violated)

Why each layer costs ≤ M − 1 expected steps

1. **Arrive at E.** The search on layer ℓ+1 ends at E, q's nearest node on ℓ+1 **(A1)**. E is the entry point on layer ℓ.
2. **Greedy only moves closer.** Every node visited on layer ℓ is closer to q than E.
3. **None of them is on ℓ+1,** or it would be q's nearest there instead of E. Sorted by distance to q, E is the first layer-ℓ node that is also on ℓ+1.
4. **Levels ignore position.** Each layer-ℓ node is on ℓ+1 w.p. 1/M, independently: coin flips down the sorted list.
5. **Count is geometric.** greedy steps on layer-ℓ <= #nodes closer to q than E = #nodes ahead of E in the sorted list, whose expected value is M - 1. Independent of N.

In practice layers are approximate, so layer 0 needs ef > 1; the log N scaling is shown empirically (Fig. 11, d = 4), not proved for high d.

Layer ℓ, sorted by distance to q



Flip coin with p = 1/M. So Exp[count] = (1 - *p*)/p = M - 1

Search cost

$O(\log N)$	layers (top level ≈ log_M N)
$\times O(1)$	steps per layer (≤ M − 1)
$\times O(1)$	distances per step (degree ≤ Mmax, **A2**)
$= O(\log N)$	expected, under **A1 + A2**

Build (§4.2.2): insert = one search + linking on its layers; expected layers per element is a constant (Eq. 1) → O(N log N)

Memory cost

What one element stores

1. **Layer 0:** every element, up to Mmax0 = 2M links
2. **Upper layers:** on layer ≥ k w.p. M^{-k}, so E[#upper layers] = Σ M^{-k} = 1/(M − 1); up to M links each → ≈ 1 link on average
3. **Link size:** 4 B per neighbor id (int32, up to ~4 billion elements)
4. **The vector:** d × 4 B for fp32, stored once

$$\text{bytes / element} \approx 4 \cdot (2M + M/(M-1)) + 4d \approx 4 \cdot (2M + 1) + 4d$$

M	6	8	16	32	48
Bytes for links (i.e., M / (M - 1) for upper layers)	53 B	69 B	132 B	260 B	388 B
Paper's bound (i.e., M/ln M for upper layers)	61 B	79 B	151 B	293 B	434 B

M = 16, d = 128, per-element:



132 B
graph links

512 B
vector (128 × fp32)

SIFT1M ≈ 0.64 GB
, before small per-element headers

- Graph overhead grows linearly in M
- At moderate M the raw vectors dominate: more memory than PQ indexes (Table 3)

Paper (§4.2.3): (Mmax0 + mL·Mmax)·4 B. mL = 1/ln M overestimates E[#upper layers] = 1/(M − 1)

HNSW vs. NSW

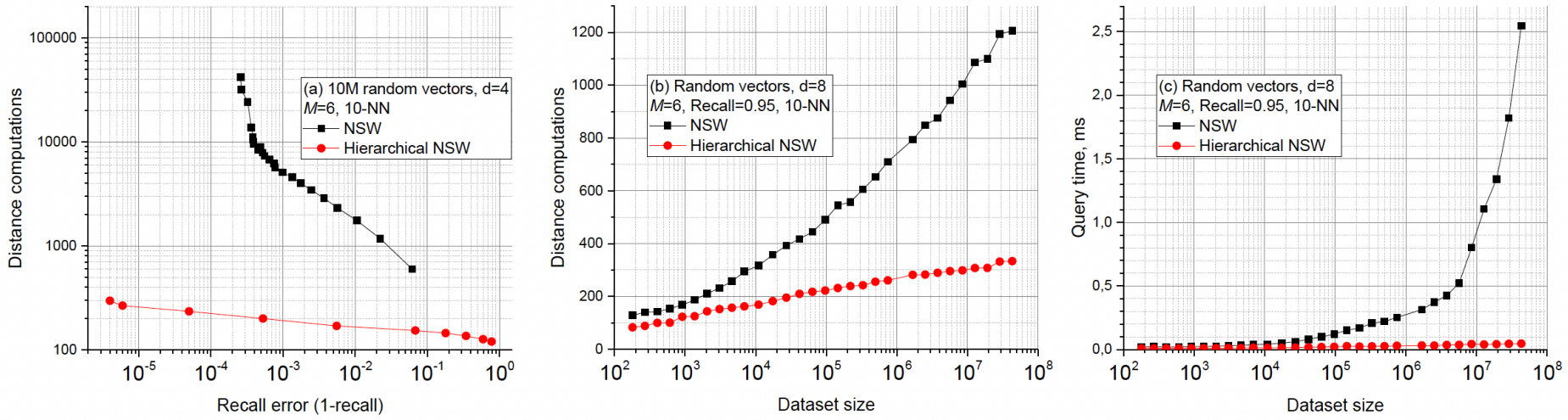


Fig. 12. Comparison between NSW and Hierarchical NSW: (a) distance calculation number vs accuracy tradeoff for a 10 million 4-dimensional random vectors dataset; (b-c) performance scaling in terms of number of distance calculations (b) and raw query(c) time on a 8-dimensional random vectors dataset.

Fewer distance computations than NSW, and the gap grows with N and with recall

Recall vs. query latency

Top-left is better

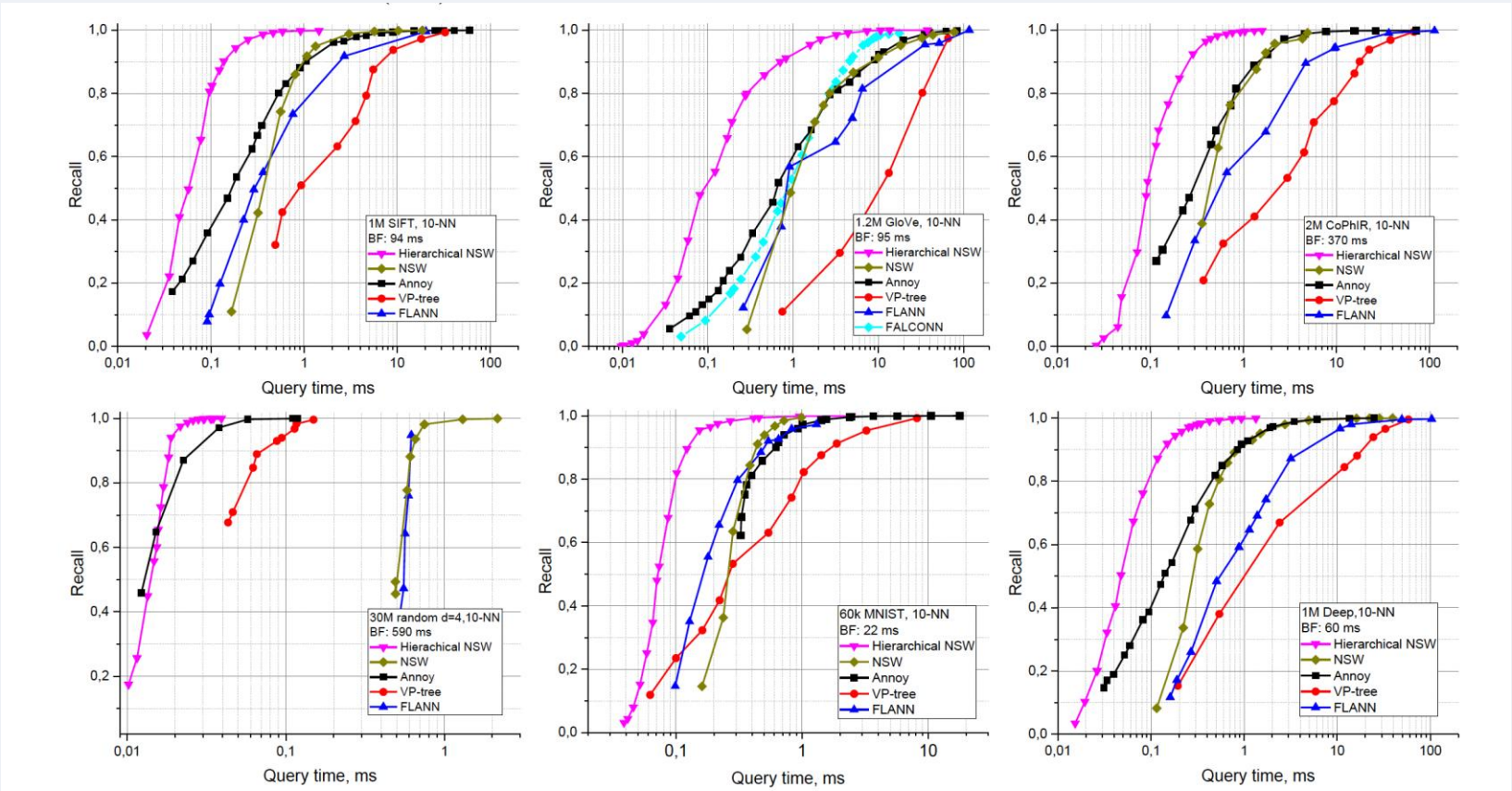


Fig. 13. Results of the comparison of Hierarchical NSW with open source implementations of K-ANNS algorithms on five datasets for 10-NN searches. The time of a brute-force search is denoted as the BF.

Wins clearly on every real dataset compared with NSW (graph-based) and three tree/partition-based indices; only slightly ahead of Annoy at d = 4

Non-metric Spaces

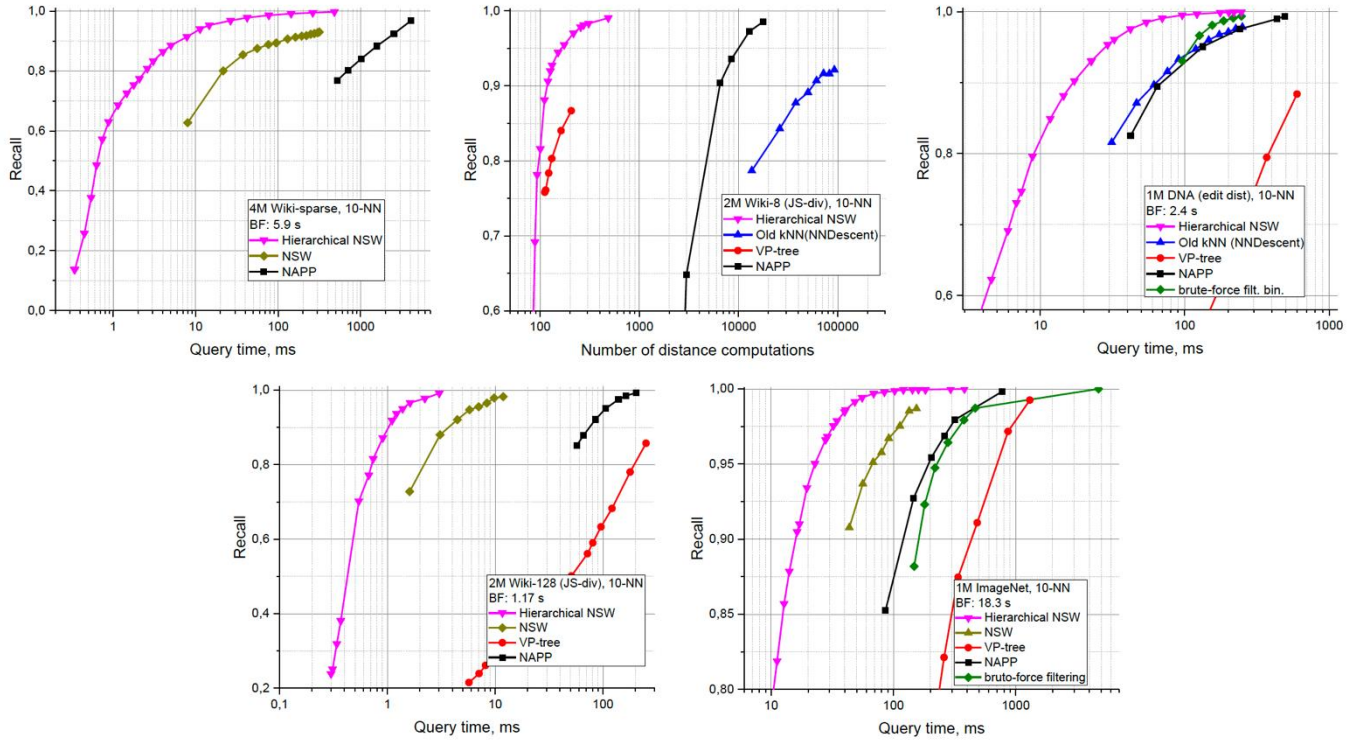


Fig. 14. Results of the comparison of Hierarchical NSW with general space K-ANNS algorithms from the Non Metric Space Library on five datasets for 10-NN searches. The time of a brute-force search is denoted as the BF.

TABLE 2.
Used datasets for repetition of the Non-Metric data tests subset.

Dataset	Description	Size	d	BF time	Distance
Wiki-sparse	TF-IDF (term frequency–inverse document frequency) vectors (created via GENSIM [58])	4M	10^5	5.9 s	Sparse cosine
Wiki-8	Topic histograms created from sparse TF-IDF vectors of the wiki-sparse dataset (created via GENSIM [58])	2M	8	-	Jensen–Shannon (JS) divergence
Wiki-128	Topic histograms created from sparse TF-IDF vectors of the wiki-sparse dataset (created via GENSIM [58])	2M	128	1.17 s	Jensen–Shannon (JS) divergence
ImageNet	Signatures extracted from LSVRC-2014 with SQFD (signature quadratic form) distance [59]	1M	272	18.3 s	SQFD
DNA	DNA (deoxyribonucleic acid) dataset sampled from the Human Genome 5 [34].	1M	-	2.4 s	Levenshtein

HNSW vs. PQ

TABLE 3.
Parameters for comparison between Hierarchical NSW and Faiss on a 200M subset of 1B SIFT dataset.

Algorithm	Build time	Peak memory (runtime)	Parameters
Hierarchical NSW	5.6 hours	64 Gb	*M*=16, *efConstruction*=500 (1)
Hierarchical NSW	42 minutes	64 Gb	*M*=16, *efConstruction*=40 (2)
Faiss	12 hours	30 Gb	*OPQ64, IMI2x14, PQ64* (1)
Faiss	11 hours	23.5 Gb	*OPQ32, IMI2x14, PQ32* (2)

Faiss IVF:
+ OPQ..: Optimized PQ Rotation
+ IMI..: Coarse partition (i.e., IVF): Inverted Multi-Index
+ PQ..: Encoding (compression)

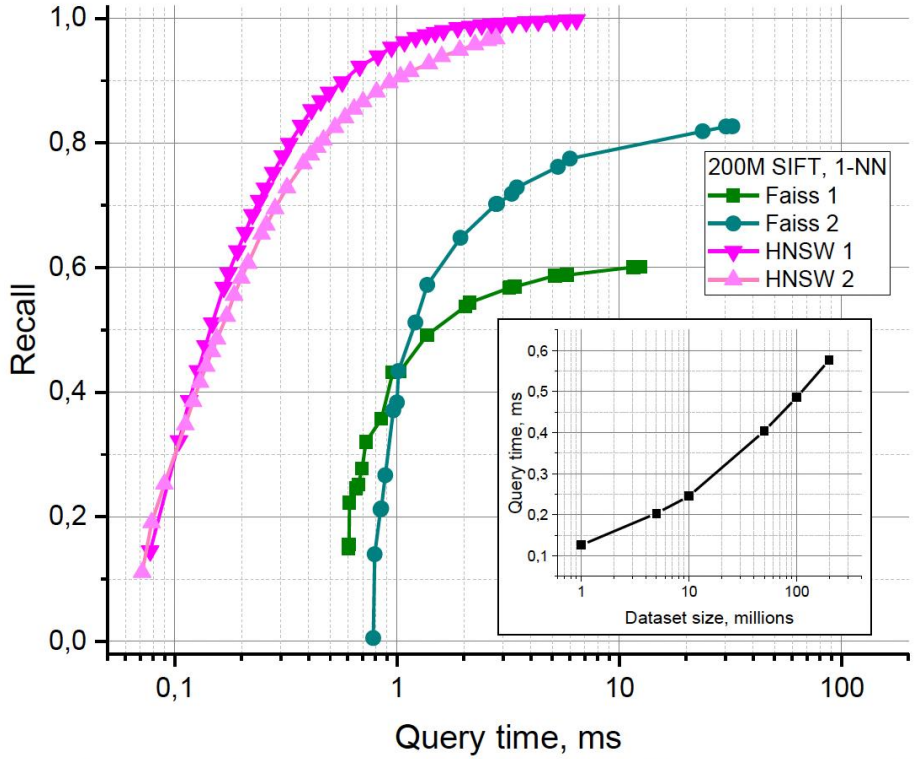


Fig. 15 Results of comparison with Faiss library on the 200M SIFT dataset from [13]. The inset shows the scaling of the query time vs the dataset size for Hierarchical NSW.

HNSW is faster, more accurate and builds faster, but uses 2–3× the RAM

Minimal code: same index, two libraries

hnswlib

```
import hnswlib
p = hnswlib.Index(space='l2', dim=128)
p.init_index(max_elements=N, M=16,
             ef_construction=200)
p.add_items(xb, num_threads=8)

p.set_ef(64)           # ef >= k
I, D = p.knn_query(xq, k=10)
```

Faiss

```
import faiss
idx = faiss.IndexHNSWFlat(128, 16)
idx.hnsw.efConstruction = 200
idx.add(xb)        # float32, contiguous

idx.hnsw.efSearch = 64
D, I = idx.search(xq, 10)
```

Lessons from practice

1 ef is the main knob

Sweep ef at query time; rebuild only for M or efConstruction. Both use max(ef, k).

2 Defaults differ

hnswlib: M 16, efC 200, ef 10.
Faiss: efC 40, efSearch 16. Set them explicitly.

3 Parallelism

Build: threads insert concurrently, one lock per neighbor list. Search: read-only graph → one query per thread, no locks.

4 RAM is the cost

Vectors dominate; use HNSW + SQ/PQ when memory is tight.

5 Deletes are soft

Marked nodes stay in the graph and in memory. No eager GC: repairing in-links is costly; smoother updates are active research (e.g. FreshDiskANN).

6 Reproducibility

Insertion order, RNG seed and thread count change the graph. Fix them when comparing.

7 Filtering is inline

Filtered-out nodes still route; only allowed ones are returned. hnswlib: filter callback; Faiss: built-in IDSelector. Pre-filter = brute force; post-filter may return < k.

8 SIMD helps less

Both vectorize the distance loop (hnswlib intrinsics; Faiss auto-vectorized, 4 at a time). Random hops make it less SIMD-friendly compared with IVF.

9 Python wrappers

Both implemented in C++, hnswlib is header-only, and Faiss is compiled C++ lib. Both have python bindings, which is a thin layer on top that passes numpy arrays.

Limitations and what came next

Limitations

- Memory: full vectors + links
- Distributed search is harder: every query enters at the top layer; but still, for both HNSW and NSW, if the search frontier even for a single query keeps go across the machine boundary again and again, the commute cost is unacceptable
- Theory assumes exact Delaunay layers; high-d is open
- Deletes and updates are left as future work
- Still many parameters to be tuned (during construction, during search, and the graph quality determined by construction will affect the effort needed during search)

What came next

- From RAM to disk, VAMANA/DiskANN
- How to do better than inline/pre-/post-filtering for filtered vector search? Predicate-agnostic or predicate-aware?
- From RAM to GPU: e.g., CAGRA
- Sparse vector + dense vector
- ……

Takeaways

1. Separate links by length scale into layers (a skip list of graphs) → O(log N) routing with bounded degree

2. The RNG-style neighbor heuristic gives diverse edges, which keeps clustered and low-d data connected

3. In practice: tune M once, then sweep ef; budget memory for vectors plus links

Questions?

SIMD: fast distances, irregular memory access

hnswlib: hand-written intrinsics

```
// hnswlib/space_l2.h  (AVX: 8 floats per register)
__m256 sum = _mm256_set1_ps(0);
while (p1 < end) {
  __m256 diff = _mm256_sub_ps(_mm256_loadu_ps(p1),
                              _mm256_loadu_ps(p2));
  sum = _mm256_add_ps(sum, _mm256_mul_ps(diff, diff));
  p1 += 8; p2 += 8;
}   // then add up the 8 lanes
```

AVX-512 / AVX / SSE chosen at runtime; fastest path needs d % 16 == 0

Faiss: auto-vectorized loops, 4 neighbors at once

```
// faiss/utils/simd_impl/distances_autovec-inl.h
FAISS_PRAGMA_IMPRECISE_LOOP      // compiler vectorizes
for (size_t i = 0; i < d; ++i) {
  const float q0 = x[i] - y0[i];  d0 += q0 * q0;
  const float q1 = x[i] - y1[i];  d1 += q1 * q1;
  ...                            // y2, y3 likewise
}
```

Built per SIMD level (AVX2, AVX-512, NEON); HNSW search uses this batch-of-4

Why HNSW is less SIMD-friendly than IVF

	HNSW	IVF
Access	≤ 2M vectors at random addresses per hop	a few contiguous lists, streamed
Dependency	next hop depends on these distances	lists chosen up front
Inner loop	small bursts + heap and visited checks	one tight loop over many codes
Kernel	1 query × 1 vector (batch of 4)	matrix ops; PQ fast-scan: LUT in SIMD registers
Bound by	memory latency (cache misses)	bandwidth and arithmetic

Mitigations: prefetch the next neighbors (both libraries); hnswlib packs each element's links + vector + label into one record.

HNSW: far fewer distances. IVF: more throughput per distance.