Approximate Nearest Neighbor Search: IVF, HNSW, and DiskANN

CS 7800/4810 — Advanced Algorithms

1 Inverted File Index (IVF)

1.1 Core Idea

The Inverted File Index (IVF) partitions the vector space into C clusters (cells) via k-means or a similar algorithm. Each cluster is represented by its centroid $\mu_1, \ldots, \mu_C$. Vectors are assigned to the cluster with the nearest centroid.

Index construction:

1. Run k-means on $\mathcal{D}$ to obtain centroids $\mu_1, \ldots, \mu_C$.
2. For each $v \in \mathcal{D}$, assign v to the cluster $\arg\min_j \|v - \mu_j\|$.

Insertion of a new vector v:

1. Compute distances to all C centroids; assign v to the nearest.

Query for q:

1. Compute distances from q to all C centroids.
2. Select the m nearest centroids (called the *probe count*).
3. Compute exact distances from q to every vector in those m clusters.
4. Return the top-k results.

1.2 Analysis

Let $n = N/C$ denote the average cluster size (assuming roughly balanced clusters). A query with probe count m examines approximately mn vectors rather than N, giving a speedup factor of roughly C/m.

The key parameter tradeoff is:

- **Number of clusters C:** Higher C means smaller clusters (faster search per probe) but more centroid comparisons and greater chance of boundary errors.
- **Probe count m:** Higher m improves recall (since true neighbors near cluster boundaries are more likely to be found) but increases query time proportionally.

Remark 1.1. *The boundary problemboundary A query q near a Voronoi boundary between two clusters may have its true nearest neighbor in the adjacent cluster. Probing only $m = 1$ cluster will miss this neighbor. Increasing m or listing vectors near boundaries in multiple clusters mitigates the issue at the cost of more computation or storage.*

1.3 Index Degradation Over Time

The k-means centroids are computed on the *initial* data distribution. As new vectors are inserted, the distribution may drift: clusters become unbalanced, centroids no longer represent their members well, and boundary errors increase. This necessitates *periodic rebuilding* of the cluster structure.

Incremental approaches exist (e.g., Xu et al., SOSP 2023) that update cluster assignments without a full rebuild, but the standard IVF implementation requires occasional retraining.

1.4 Properties Summary

Advantage	Disadvantage
Low memory overhead (centroids only)	Not the fastest index type
Fast inserts (new vectors immediately visible)	Needs periodic cluster rebuilding
Easy to scale (increase C)	Accuracy degrades with distribution drift
Simple: one main parameter (m)	

Table 1: IVF properties summary.

On the ANN-Benchmarks suite (SIFT-1M, Xeon), IVF (FAISS) achieves roughly 900 QPS at 96% recall — about 50× faster than brute-force.

2 HNSW: Hierarchical Navigable Small Worlds

HNSW (Malkov and Yashunin, TPAMI 2020) is widely regarded as the "crown jewel" of ANNS graph indexes. It offers near state-of-the-art speed–accuracy tradeoffs and is the backbone of several production vector databases (Qdrant, Weaviate, etc.). HNSW combines two classical ideas: *navigable small world* (NSW) graphs and *skip list* hierarchies.

2.1 Navigable Small World Graphs

Definition 2.1. *Navigable Small World graphnsw A* navigable small world *(NSW) graph is a class of graphs containing both long-range and short-range edges such that the characteristic path length is $O(\log N)$. Greedy search, i.e., starting at an entry point and always moving to the neighbor closest to the query, finds approximate nearest neighbors in $O(\log N)$ hops.*

However, in a plain NSW graph, the average out-degree grows as $O(\log N)$, making each hop more expensive. The total search cost becomes $O(\log^c N)$ for some $c > 1$ (polylogarithmic), which is suboptimal.

2.2 Skip Lists

A skip list is a probabilistic data structure for ordered sequences. Elements are organized in multiple levels: level 0 contains all elements, and each higher level contains a geometrically decreasing random subset. Search begins at the top level and descends when it can no longer make progress, achieving $O(\log N)$ expected search time.

2.3 HNSW = NSW + Skip List

HNSW layers a hierarchy on top of the NSW idea. Vectors are randomly assigned to layers $0, 1, \ldots, L_{\max}$, where the probability of appearing at layer ℓ decays geometrically (sampled from a log-uniform distribution). Every vector appears at layer 0; only a few appear at the top.

Within each layer, vectors are connected to nearby neighbors, forming an NSW-like proximity graph. Higher layers have fewer nodes with longer-range connections; lower layers have more nodes with shorter-range, finer-grained connections.

Query(q, k):

1. Enter the graph at the single entry point in the top layer $L_{\max}$.
2. **Greedy descent (layers $L_{\max}$ down to 1):** At each layer, greedily traverse the graph toward q by always moving to the connected neighbor closest to q. When no closer neighbor exists, descend to the next layer, using the current node as the entry point.
3. **Expanded search (layer 0):** At layer 0, expand the search to maintain a dynamic candidate list of size `efSearch`, exploring neighbors of candidates in a best-first fashion.
4. Return the k closest candidates found.

The bounded out-degree at each layer (parameter $M_{\max}$) ensures each hop takes $O(M_{\max} \cdot d)$ time (computing $M_{\max}$ distances in d dimensions). With $O(\log N)$ layers, each traversed in $O(1)$ to O(small constant) hops, the overall query complexity is $O(\log N)$ distance computations (ignoring the layer-0 expansion).

2.4 HNSW Insertion

Insert(x):

1. Sample the highest layer L for x from $\lfloor -\ln(\text{uniform}(0,1)) \cdot m_L \rfloor$, where m_L is a normalization factor.
2. **Find entry point:** Starting from the global entry point at layer $L_{\max}$, perform greedy search for x down to layer $L+1$.
3. **Insert at each layer L down to 0:**
   - Find $W =$ `efConstruction` closest neighbors of x in the current layer.
   - Create the node x and connect it to the M best neighbors from W.
   - For each neighbor in W, if its out-degree now exceeds $M_{\max}$, *trim* its edge list by removing the least useful connections.
   - Descend to the next layer using W as the set of candidate entry points.

The trimming step is critical: without it, repeated insertions would cause out-degree to grow unboundedly, destroying the $O(\log N)$ search guarantee.

2.5 HNSW Parameters

Parameter	Description	Query time	Insert time	Memory	Recall
M / $M_{\max}$	Neighbors added per vertex; max out-degree	✓	✓	✓	✓
`efSearch`	Candidate list size during query (layer 0)	✓			✓
`efConstruction`	Candidate list size during index build	(if M high)	✓		(if M low)

Table 2: HNSW parameters and their effects.

The parameter M is the most consequential: it simultaneously affects all four performance dimensions. Increasing M improves recall and creates a better-connected graph, but increases memory (each node stores M edge pointers per layer) and slows both queries and inserts. The

`efSearch` parameter provides a query-time knob: higher `efSearch` improves recall at the cost of more distance computations. Notably, `efConstruction` has diminishing returns on recall beyond a moderate value, except when M is very small.

2.6 Properties of HNSW

Strengths:

- Excellent speed–accuracy tradeoff (near state-of-the-art).
- On ANN-Benchmarks (SIFT-1M), HNSW (FAISS) achieves approximately 3000 QPS at 95% recall — roughly 100×–500× faster than brute-force.
- Incremental insertion is supported (in principle).
- Lower memory footprint than LSH.

Weaknesses:

- **Memory overhead:** higher than IVF or PQ, since the graph structure (edge lists at multiple layers) must be stored. Addressed by combining HNSW with quantization.
- **Insertion is slower than querying**, due to the need to find neighbors, create edges, and trim.
- **Deletion is problematic:** removing a node can disconnect the graph. Standard practice uses tombstones (marking nodes as deleted without removing edges), which causes memory bloating and recall degradation over time.
- **Needs periodic rebuilding** when the fraction of deleted/updated vectors grows large.

Remark 2.2. *Update degradationhnsw-degrade Singh et al. (arXiv, 2021) showed experimentally that as batches of updates (5% of dataset size each) are applied, HNSW's 5-recall@5 drops from ∼98% to ∼88% after 20 batches, while DiskANN's Vamana graph degrades more gracefully (from ∼98% to ∼93%).*

3 DiskANN and the Vamana Graph

DiskANN (Subramanya et al., NeurIPS 2019) addresses a key limitation of HNSW: the requirement that the entire index fits in memory. DiskANN stores the graph index on SSD and uses a small in-memory navigation structure to guide disk access, enabling billion-scale ANNS with limited RAM.

3.1 The Vamana Graph Index

At the heart of DiskANN is the *Vamana* graph, a single-layer proximity graph (no hierarchy) with bounded out-degree.

Build$(\mathcal{D}, R, L, \alpha)$:

1. Initialize a random graph G where each node has R random neighbors.
2. Select a *medoid* s (the vector closest to the centroid of $\mathcal{D}$) as the single entry point.
3. For each vector $v \in \mathcal{D}$ (in random order):
   (a) Run greedy search from s for v with candidate list size L, collecting all visited nodes into a set $\mathcal{V}$.
   (b) Use a *robust pruning* procedure to select at most R neighbors for v from $\mathcal{V}$, preferring neighbors that are diverse in direction (not all clustered in one region).
   (c) For each new neighbor u of v, add the reverse edge $v \to u$; if u's out-degree exceeds R, prune u's edge list.

4. Optionally, repeat the process for a second pass to improve graph quality.

Parameters: R = max out-degree, L = search list size during construction, $\alpha \geq 1$ = pruning parameter controlling long-range vs. short-range edge preference.

The pruning parameter α plays a critical role. When $\alpha = 1$, the algorithm greedily selects the closest candidates (short-range edges). When $\alpha > 1$ (typically 1.2), the algorithm tolerates slightly farther candidates that point in diverse directions, creating long-range edges that reduce the graph's diameter and improve navigability — analogous to the long-range edges in HNSW's upper layers, but achieved in a single flat graph.

3.2 Disk-Resident Search

DiskANN stores the Vamana graph and full-precision vectors on SSD. A small in-memory component holds compressed (product-quantized) representations of all vectors, used for approximate distance computations during graph traversal. When the search identifies the most promising candidates using compressed vectors, it issues SSD reads to fetch the full-precision vectors for exact reranking.

Query(q, k, W):

1. Start at the medoid entry point s.
2. Maintain a candidate list of size W (the "beam width").
3. At each step, pop the closest unvisited candidate c. Issue an SSD read for c's neighbors and their full-precision vectors.
4. Compute exact distances to c's neighbors; insert promising ones into the candidate list.
5. Repeat until the candidate list converges (no unvisited candidate is closer than the k-th best found so far).
6. Return the top-k results.

The number of SSD reads per query is typically small (tens to low hundreds), making SSD latency manageable.

3.3 Why DiskANN Outperforms HNSW on SSD

HNSW's hierarchical structure requires random memory accesses across multiple layers, which translates poorly to SSD (each layer transition is a random read). Vamana's single-layer design, combined with SSD-aligned data layout and prefetching, achieves fewer total I/O operations per query.

On ANN-Benchmarks (SIFT-1M, in-memory), DiskANN/Vamana achieves approximately 7000 QPS at 95% recall — roughly 2× faster than HNSW (FAISS) and 400× faster than brute-force.

3.4 Update Resilience

The Vamana graph degrades more gracefully under updates than HNSW. Because Vamana uses robust pruning with diverse edge selection, the graph structure is more resilient to individual node additions or deletions. Empirical results show that after 20 batches of 5% updates, Vamana retains ∼93% recall@5 compared to HNSW's ∼88%.

Index	**Strengths**	**Weaknesses**	**Typical QPS (SIFT-1M, 95% recall)**
Flat	Exact; no extra memory; instant updates; no rebuild	$O(Nd)$ per query; infeasible for $N > 10^5$	16 (100% recall)
IVF	Low memory overhead; fast inserts; simple tuning (m)	Needs periodic rebuild; not the fastest	∼900 (96%)
HNSW	Excellent speed-accuracy; near SotA	Memory overhead; slow deletes; rebuild needed	∼3000
DiskANN (Vamana)	Fastest in-memory; SSD-resident; update-resilient	Build cost; complex implementation	∼7000
LSH	Bounded error; fast updates; no rebuild; data-independent	Inferior performance; high memory; slow for $d > 128$	Rarely used

Table 3: Comparison of ANNS index types. QPS numbers are approximate, from ANN-Benchmarks on a 3rd-gen Xeon.

4 Comparison of ANNS Indexes

5 Choosing an Index

The right index depends on your workload requirements. Key considerations include: the dataset size N, the vector dimensionality d, how frequently vectors are inserted or deleted, whether deleted vectors must be immediately invisible or can use tombstones, the acceptable latency and recall targets, and the available memory budget.

For small, frequently-updated collections, the flat index is hard to beat. For larger static datasets where memory is plentiful, HNSW provides excellent speed and accuracy. When the index must reside on disk or when RAM is constrained, DiskANN is the leading choice. IVF occupies a useful middle ground: simpler to implement and tune than graph indexes, with reasonable performance and low memory overhead.

6 Bibliographic Notes

The HNSW algorithm was introduced by Malkov and Yashunin (TPAMI, 2020), building on the earlier NSW work by Malkov et al. (Information Systems, 2014). DiskANN and the Vamana graph were proposed by Subramanya et al. (NeurIPS, 2019), with subsequent updates for fresh indices by Singh et al. (arXiv, 2021). The IVF index has roots in classical vector quantization and was popularized in the ANNS context by Jégou et al. (TPAMI, 2011) as part of the product quantization framework. The comprehensive survey by Pan et al. (VLDB Journal, 2024) provides an excellent taxonomy of ANNS index structures. ANN-Benchmarks (`ann-benchmarks.com`) is the standard evaluation platform for comparing ANNS implementations.

For locality-sensitive hashing, the foundational work is by Indyk and Motwani (STOC, 1998)

and Datar et al. (SCG, 2004) for the Euclidean case (E^2LSH). Notable variants include FALCONN (Andoni et al., NeurIPS 2015) for cosine similarity and SPANN (Chen et al., NeurIPS 2021) for learned, disk-resident hashing.