Exhaustive Analysis of Vector Data Storage and Approximate Nearest Neighbor (ANN) Search Algorithms: Complexity, Performance, and Implementation

I. Theoretical Foundations of High-Dimensional Search

A. The Vector Space Model and Embedding Generation

The recent paradigm shift from traditional keyword matching to semantic search relies fundamentally on the Vector Space Model (VSM).[1] In this model, all forms of unstructured data—including text, images, and audio—are transformed into dense, uniform numerical representations known as vector embeddings.[2] These embeddings capture the semantic meaning or intrinsic features of the data, allowing machine intelligence to understand relationships based on conceptual similarity rather than just lexical matches.[3]

Similarity search, therefore, becomes a problem of calculating vector proximity within this high-dimensional embedding space. The "closeness" of two vectors dictates the relevance of the corresponding data points. This proximity is measured using specific distance metrics. The most common metrics utilized in vector databases include L_2 (Euclidean) distance, which measures the shortest path between two points; Dot Product similarity; and Cosine similarity, which is often used for normalized vectors as it measures the angle between them.[4]

B. The Dimensionality Crisis: The Curse of Dimensionality (CoD)

The computational challenge intrinsic to vector similarity search is formally encapsulated by the Curse of Dimensionality (CoD).[5] This phenomenon describes the challenges that arise when analyzing data in spaces with a large number of dimensions (D). As D increases, the volume of the space expands exponentially.[2] Consequently, data points become extremely sparse, often resulting in all points appearing nearly equidistant from one another.[2] This sparsity obscures meaningful patterns and relationships, diminishing the efficacy of distance-based measurements.

In this high-dimensional environment, the traditional Brute Force k-Nearest Neighbor (k-NN) approach becomes computationally prohibitive.[6] Exact k-NN requires calculating the distance from the query vector to every single vector in the dataset (N). This results in a search time complexity of $O(N \cdot D)$, which scales linearly with the size of the dataset (N) and the

dimensionality (D).[7] While feasible for small datasets (e.g., under 10,000 vectors) or scenarios demanding 100% accuracy, such as validation of medical diagnoses, this linear scaling makes it intractable for modern large-scale applications like real-time recommendation systems or large language model (LLM) Retrieval-Augmented Generation (RAG) applications that often involve millions or billions of vectors.[6]

To address the limitations imposed by the CoD and the computational inefficiency of $O(N)$ searches, Approximate Nearest Neighbor (ANN) algorithms were engineered. ANN deliberately sacrifices the guarantee of 100% precision in results for a dramatic improvement in search speed, often achieving sublinear time complexity, such as $O(\log N)$.[6] This trade-off is the central engineering decision in high-dimensional search systems, making billion-scale vector processing practical on standard server infrastructure.[9]

C. The ANN Trilemma: Speed, Accuracy (Recall), and Resource Cost

The necessity of ANN algorithms establishes an inherent tension in the design of vector search systems. The architectural choice of index is not purely technical but rather a strategic decision regarding how to balance the trilemma between performance (query latency), accuracy (recall), and cost (memory/compute).[7]

ANN achieves its high efficiency by pre-processing the data into intelligent indexing structures—such as graphs, clusters, or hash tables—designed to "skip irrelevant comparisons".[7] For instance, a music streaming service with millions of tracks might prioritize finding 95% accurate results in milliseconds over 100% accuracy in seconds to deliver a satisfactory user experience.[6] This deliberate trade-off means that the marginal cost of pursuing perfect results rises exponentially. The process of selecting an ANN algorithm determines the acceptable boundaries for recall (@k) versus query latency (e.g., p99 latency) and the required hardware footprint.

II. Algorithmic Categories and Search Principles

ANN indexing techniques broadly fall into several categories based on how they structure the high-dimensional space: Graph-based, Cluster-based, and Quantization/Hashing methods.

A. Graph-Based Methods: Hierarchical Navigable Small World (HNSW)

The Hierarchical Navigable Small World (HNSW) algorithm is widely regarded as a state-of-the-art method for ANN search, known for its exceptional speed and high recall.[7] HNSW is a fully graph-based solution, where the index is a complex network of nodes (vectors) connected by edges (links to neighbors).[7] The core principle involves building a multi-layer structure of proximity graphs.[7] This hierarchical design leverages the Navigable Small World (NSW) principle, ensuring short path lengths between any two nodes. The sparse upper layers contain long-range connections that facilitate rapid global navigation, while the denser lower layers focus on local neighborhood refinement.[14]

B. Cluster-Based Methods: Inverted File Index (IVF)

Inverted File Index (IVF) algorithms reduce the search space by partitioning the dataset into distinct clusters using algorithms like k-means.[12] During the indexing process, vectors are grouped by their nearest cluster centroid. When a query is initiated, the system first identifies the cluster centroids closest to the query vector. It then limits the exhaustive search only to the relevant subset of vectors residing in those few clusters, drastically reducing the required computation.[12] IVF is particularly suitable for datasets that naturally segment well and is often optimized by combining it with quantization methods (IVF-PQ) to save memory.[9]

C. Quantization Methods: Product Quantization (PQ)

Product Quantization (PQ) is not a standalone index but a powerful compression technique often used in conjunction with cluster or graph-based indices to improve efficiency. PQ works by splitting high-dimensional vectors into smaller subvectors, encoding each subvector using a separate low-bit codebook.[2] This compression significantly reduces the memory footprint, often achieving memory savings up to 75%.[9] The reduced size allows a larger index to fit into server RAM, which directly reduces query latency. While fast, PQ is a lossy compression method, meaning it can reduce precision if not carefully tuned.[16]

D. Hashing Methods: Locality-Sensitive Hashing (LSH)

Locality-Sensitive Hashing (LSH) utilizes randomized hash functions designed to maximize collisions—specifically, aiming to place similar vectors into the same hash buckets with high probability.[17] This approach relies on the mathematical foundation that a family of hash functions is locality sensitive if similar items hash to the same bucket with high probability, and dissimilar items hash to different buckets with high probability.[18] LSH offers extremely fast index construction and query speed, but it typically suffers from reduced precision compared to graph-based methods.[6]

III. In-Depth Analysis of Dominant ANN Indexing Techniques

A. Hierarchical Navigable Small World (HNSW)

The architecture of HNSW is predicated on the concept of a hierarchical probability skip list.[13] The index is composed of multiple layers, where the top layers are sparse and contain long-range edges for rapid traversal across the vector space. As the search descends through the layers, the graph becomes progressively denser, allowing for finer-grained navigation.[12] A single entry point is maintained at the top layer.[14] HNSW employs an internal pruning strategy, often inspired by Random Neighbor Graphs (RNG), to eliminate redundant edges, ensuring the graph remains sparse yet robustly connected for efficient traversal.[14]
The search mechanism begins at the single entry point in the highest layer. At each layer, the algorithm greedily navigates to the nearest neighbors of the current node, guiding the search path.[19] This process is repeated, refining the candidate set as it descends through the

hierarchy until the search reaches the lowest, densest layer where the final, precise nearest neighbors are selected.[20] This layered approach allows HNSW to bypass comparing the query vector against the vast majority of the dataset.

HNSW Complexity and Operational Parameters

HNSW is computationally expensive during index construction but highly efficient during query time. The theoretical query complexity is logarithmic, achieving $\sim O(\log N)$, making it suitable for million-to-billion scale datasets.[8] However, the initial construction of the hierarchical graph requires substantial computational effort, estimated to be $\sim O(N \cdot D \log N)$.[8] Building an HNSW index on a dataset comprising tens of millions of vectors can take approximately 10 hours, and construction times can extend to five days for billion-scale datasets, even when using relaxed parameters.[22]

Modern systems like Weaviate manage this construction cost by incrementally building the index with each incoming object, which helps the system remain highly responsive to dynamic data updates.[15] The search quality and memory cost are controlled by several parameters: M (the maximum number of neighbors linked to a node), $ef_{construction}$ (which controls the graph quality during indexing), and ef_{search} (which determines the number of candidate neighbors checked during the query).[21] Increasing ef_{search} improves recall by expanding the search space but directly increases query latency and memory usage.[6] The high initial index construction cost and resulting high memory usage (due to graph storage overhead) must be amortized over frequent, rapid search operations.[16]

B. Inverted File Index (IVF)

The performance of an IVF index is highly dependent on how well the dataset naturally clusters and the quality of the initial pre-training phase, typically using K-Means, to define the cluster centroids.[12] If the centroids do not accurately represent the data distribution, the accuracy may suffer, risking relevant vectors being missed if they fall into unprobed clusters.[12] The primary parameter controlling the search trade-off in IVF is n_{probe}, which specifies the number of cluster centroids searched during a query.[21] A small n_{probe} yields faster results but increases the probability of missing the true nearest neighbors.

A significant operational advantage of IVF arises in scenarios involving hybrid search, where vector similarity is combined with traditional scalar metadata filtering (e.g., filtering results by a date range or category). HNSW performance can become unstable under high filtering ratios (e.g., 90%+), as the removal of nodes fractures the continuous graph structure, potentially degrading performance to near brute-force traversal.[21] In contrast, IVF exhibits stable query speed because it performs a "coarse filtering at the centroid level" before commencing the vector comparisons within the candidate clusters. This architectural difference allows IVF to provide more predictable performance for searches that involve stringent structured query conditions.[21]

C. Product Quantization (PQ) and Compression

Product Quantization is a critical technique for managing cost and scalability in high-dimensional vector search. By splitting a 128-dimensional vector into smaller segments (subvectors) and applying compression, PQ can achieve a substantial reduction in the memory required to store the index.[16] This compression is essential for deploying large-scale indexes, such as those used with the BIGANN dataset, where memory constraints are often the limiting factor for fitting the index entirely into RAM.[9]

The efficiency gains of PQ enable compressed-domain search, accelerating distance calculations. However, this method is lossy. Consequently, the reduction in precision introduced by PQ must be counterbalanced by careful tuning of parameters, such as the number of subvectors (m), to ensure that semantic meaning is retained while maximizing compression.[2] When combined with clustering (e.g., IVFPQ), the index offers low memory usage and fast response times, making it optimal for environments with limited resources.[16]

D. Locality-Sensitive Hashing (LSH)

LSH is a robust method when extreme search speed is required and a greater tolerance for approximation error is acceptable.[6] It uses multiple hash tables and hash functions to amplify the probability that similar items are placed into the same buckets.[18]

LSH excels particularly in index construction speed. Experimental data demonstrates that LSH can build an index for approximately one million vectors in just 0.9 seconds, compared to 40 seconds for IVF-PQ and 290 seconds for HNSW-PQ on the same dataset.[24] This rapid construction time makes LSH highly attractive for systems that prioritize indexing speed and where query accuracy is a secondary concern.

IV. Quantitative Comparison of Speed and Complexity

A. Theoretical Complexity Analysis

The fundamental difference between search algorithms is codified in their Big O notation for time complexity. The shift from linear to logarithmic search time is the primary technical justification for adopting ANN in large-scale systems.

Theoretical Complexity Comparison of Key ANN Algorithms

Algorithm	Approach Type	Theoretical Query Complexity	Index Construction Complexity	Memory Footprint
k-NN (Brute Force)	Exhaustive Search	$O(N \cdot D)$ (Linear) [7]	$O(1)$ (None)	$O(N \cdot D)$ (Raw Vectors)
HNSW	Graph-Based	$\sim O(\log N)$ (Logarithmic) [6]	$\sim O(N \cdot D \log N)$ (High) [8]	High (Graph Structure Overhead) [16]
IVF (Flat)	Cluster-Based	Sublinear, $\sim O(N/k \cdot	Moderate, $O(k \cdot D)$	$O(N \cdot D)$

		n_{probe})$	(K-Means) [12]	
IVF-PQ	Cluster + Compression	Sublinear (Compressed Distance)	Moderate + Quantization Overhead	Very Low (Compressed) [9]
LSH	Hashing/Projection	Sublinear, dependent on hash tables	Very Fast, $O(N)$ dependent on hash function complexity [24]	Low/Moderate

The complexity analysis underscores the architectural trade-offs. HNSW requires a significant initial time investment in index construction ($O(N \cdot D \log N)$) to achieve its superior, logarithmic query speed ($O(\log N)$).[8] This high initial cost is necessary because HNSW must build a highly connected, multi-layered graph to guarantee efficient traversal. Conversely, k-NN has virtually no index construction cost but fails catastrophically at scale due to its linear query time complexity. Algorithms incorporating PQ maintain a very low memory footprint, offsetting the high memory consumption associated with graph-based structures like vanilla HNSW.[23]

B. Empirical Performance Benchmarks: The Billion-Vector Scale

To evaluate the practical implications of these algorithmic differences, it is necessary to examine their performance at production scale. Experiments conducted on the BIGANN dataset, which contains 1 billion 128-dimensional vectors, provide clear evidence of the trade-offs between speed, recall, and resource usage.[25] The following table synthesizes the performance of HNSW and IVFPQ configurations, focusing on p99 query latency (tail latency) and recall at $k=10$.

Empirical Comparison of HNSW vs. IVFPQ on BIGANN (1 Billion Vectors)

Algorithm Configuration	**Optimization Priority**	**Recall@10**	**p99 Query Latency (ms)**	**Native Memory Consumption (GB)**
HNSW (hnsw1)	Speed/Low Latency	0.84	16.9	High
HNSW (hnsw3)	High Recall	0.99	32.2	Highest
IVFPQ (ivfpq4)	Balanced (Rec. > 0.6)	0.61	151.8	Low
IVFPQ (ivfpq1)	Memory/Cost Optimized	0.17	106.4	Lowest

Data synthesized from the BIGANN experiment detailed in [25] and.[25]

The empirical data demonstrates a clear correlation between resource expenditure, latency, and recall. HNSW consistently yields superior performance, exhibiting much lower tail latency (as low as 16.9 ms) and significantly higher recall (up to 99%) compared to IVFPQ. However, this superior performance is only achieved at the cost of "High" to "Highest" native memory

consumption.[25]
A closer analysis of the HNSW configurations reveals the non-linear cost of accuracy. Improving the recall rate from 0.84 (hnsw1) to 0.99 (hnsw3) necessitates an approximately twofold increase in the p99 query latency, from 16.9 ms to 32.2 ms. This explicit quantification shows that the marginal gain in accuracy (the final few percentiles of recall) requires a disproportionately large sacrifice in search speed and increased memory resources.
Conversely, IVFPQ provides a path to search 1 billion vectors with the "Lowest" memory footprint, making it highly cost-effective for large-scale storage. This efficiency comes with substantial compromises: even the best IVFPQ configurations demonstrated tail latencies significantly slower than HNSW (e.g., 151.8 ms) and achieved only moderate recall (e.g., 0.61). For applications that can tolerate this increase in latency and reduced accuracy—such as those prioritizing infrastructure cost control—IVFPQ serves as the optimal algorithm.[25]

V. Source Code Availability and Operational Deployment

The availability of vector search capabilities spans dedicated low-level libraries, general-purpose databases with extensions, and specialized vector database management systems.

A. Core Libraries and Open-Source Frameworks

Low-level libraries provide maximal control over indexing parameters and are essential tools for experimentation and benchmarking:

1. **Facebook AI Similarity Search (Faiss):** Faiss is a seminal library for similarity search, supporting a vast array of index types. It includes implementations for all major ANN indices, such as HNSW, IVF, LSH, and composite indices like IVF-PQ.[4] Faiss is widely noted for its high performance and ability to leverage GPUs and parallel processing, accelerating vector operations 10 to 100 times faster than CPU-only systems for large datasets.[9]
2. **Annoy:** Annoy (Approximate Nearest Neighbors Oh Yeah) is distinguished by its tree-based approach, utilizing Random Projection Forests.[26] A key feature of Annoy is its ability to store indices on disk, which provides an excellent balance of speed and simplicity for medium-to-large datasets (e.g., 1M 50-dimensional embeddings).[26] Its low memory footprint and disk-based indexing make it a highly cost-effective choice for large datasets that exceed available RAM capacity, though it often sacrifices some search accuracy compared to HNSW.[23]
3. **Scikit-learn:** This library offers straightforward implementations of traditional k-NN for small-scale applications or validating diagnostic searches.[6]

B. Production-Ready Vector Database Implementations

The adoption of specialized vector databases or integrated solutions is increasingly common for production deployments, largely because they address the inherent operational complexities of data management.

Dedicated Vector Databases

Dedicated vector databases optimize their architecture specifically for high-dimensional operations, metadata handling, and distributed index scaling.

1. **Milvus:** An open-source vector database designed for embedding similarity search. Milvus supports a comprehensive set of index types, including HNSW, IVF, FLAT (brute-force), SCANN, and DiskANN.[4] This flexibility allows users to select the index type best suited for their specific performance and cost objectives.
2. **Qdrant:** Qdrant currently focuses its dense vector indexing exclusively on HNSW.[27] It significantly enhances the HNSW algorithm by implementing **Filterable HNSW**. This innovation allows the system to integrate vector search with structured filtering conditions efficiently, avoiding the performance degradation seen when traditional HNSW graphs are heavily filtered by metadata.[20]
3. **Weaviate:** Weaviate uses HNSW as its core production-ready indexing algorithm.[15] Its architecture is optimized for dynamic environments, supporting the incremental building of the HNSW index as new vectors arrive.[15]

Integrated Vector Solutions

General-purpose databases have begun integrating vector capabilities, offering a unified platform for mixed workloads:

1. **PostgreSQL:** Through the pgvector extension, PostgreSQL supports the storage and querying of vector embeddings. It supports indexing methods such as ivfflat to optimize similarity search performance, allowing complex queries that combine vector searches with traditional SQL operations.[4]
2. **OpenSearch:** OpenSearch integrates vector search by supporting k-NN algorithms like HNSW and IVFPQ, bringing the power of vector search together with traditional search and analytics capabilities.[4]

C. Scalability and Index Maintenance

Operational deployment of vector search at scale introduces a crucial architectural decision regarding data silos. Using standalone libraries like Faiss requires engineering teams to build complex Extract, Transform, Load (ETL) pipelines to synchronize the vector embeddings with the associated metadata (e.g., text, author, timestamp) stored in a separate operational or analytical database.[1] This synchronization adds latency, increases overhead, and risks consistency issues.

Dedicated vector databases and integrated solutions eliminate this problem by storing and managing both vector and scalar metadata natively within the same system. This streamlined architecture significantly reduces the operational complexity and the total cost of ownership (TCO) for production AI stacks.[1]

For scalability, large vector databases employ distributed architectures relying on sharding and replication.[29] Sharding divides the massive dataset into smaller, manageable chunks distributed across multiple nodes, ensuring efficient query execution. Replication ensures data redundancy by storing copies of shards on different nodes, guaranteeing resilience and high availability.[29]

Memory management remains a paramount concern. HNSW, optimized for low latency, generally performs best when the entire graph structure resides in fast RAM, leading to high infrastructure costs, particularly for billion-scale deployments.[23] Conversely, algorithms optimized for cost, such as Annoy, manage index size by storing the structure on disk, making them efficient even in memory-constrained environments.[23] The selection of an architecture depends on whether the system is primarily constrained by query latency or hardware budget.

VI. Conclusion and Strategic Recommendations

The selection of a vector data storage and search algorithm requires a rigorous analysis of the application's tolerance for approximation error, its latency requirements, and the available infrastructure budget. The fundamental engineering challenge is the deliberate trade-off between guaranteed precision and scalable sublinear search performance.

A. Algorithmic Decision Matrix by Application Need

Application Priority	Optimal Algorithm Choice	Performance Characteristics	Resource Constraints
Highest Accuracy & Real-time Latency (e.g., Conversational RAG, High-Value Recommendations)	HNSW (Optimized for high ef_{search})	Extremely Fast (low $\text{p}99$ latency), Recall > 0.95 (up to 0.99 at high cost) [25]	Highest memory consumption; Long initial construction time [22]
High Scalability & Low Cost (e.g., Massive Data Deduplication, Ad Retrieval)	IVF-PQ or LSH	Very Low Memory Footprint, Moderate Latency ($\text{p}99 > 100$ ms for IVF-PQ) [25]; Fast indexing (LSH) [24]	Tolerates reduced recall and higher tail latency [16]
Filtered Search & Predictable Performance (e.g., Hybrid Search with Metadata Constraints)	Filterable HNSW (Qdrant) or IVF	Stable Query Speed under high filtering ratio [20]	Requires indexing stability under dynamic data segmentation

B. Architectural Recommendations for Production Systems

The analysis confirms that the computational efficiency of ANN, particularly the logarithmic scaling of HNSW, is the primary enabler of billion-scale semantic search. However, the path to

production mandates careful consideration of operational complexity:

1. **For maximum performance and control (Prototyping/Optimization):** Low-level libraries like Faiss should be employed. These allow for precise parameter tuning (e.g., ef_{search} in HNSW) and direct leveraging of GPU acceleration to maximize search throughput.[9]
2. **For distributed, production-critical applications (Minimizing TCO):** Dedicated vector databases (e.g., Milvus, Qdrant, Weaviate) are recommended. These systems abstract away the complexities of distributed indexing, sharding, replication, and, crucially, eliminate the high engineering overhead associated with maintaining separate vector and scalar metadata silos.[1]
3. **For unified data management (Mixed Workloads):** Integrated solutions like PostgreSQL with pgvector or OpenSearch allow organizations to leverage existing database infrastructure, providing the flexibility to combine high-fidelity vector searches with traditional SQL operations on a single platform.[4]

Ultimately, the choice of algorithm—whether prioritizing the high-recall, high-cost HNSW or the memory-efficient, lower-recall IVFPQ—is determined by the application's sensitivity to milliseconds of latency versus gigabytes of infrastructure memory. The continuous evolution of vector databases focuses on mitigating the index construction costs and memory overhead of state-of-the-art algorithms while maintaining their superior $O(\log N)$ query speed.

Referenzen

1. A complete guide to vector search - Redis, Zugriff am Dezember 7, 2025, https://redis.io/blog/vector-search-guide/
2. What is the curse of dimensionality and how does it affect vector search? - Milvus, Zugriff am Dezember 7, 2025, https://milvus.io/ai-quick-reference/what-is-the-curse-of-dimensionality-and-how-does-it-affect-vector-search
3. What Is Milvus? A Distributed Vector Database - Oracle, Zugriff am Dezember 7, 2025, https://www.oracle.com/database/vector-database/milvus/
4. What Is a Vector Database? Top 10 Open Source Options - Instaclustr, Zugriff am Dezember 7, 2025, https://www.instaclustr.com/education/vector-database/top-10-open-source-vector-databases/
5. Zugriff am Dezember 7, 2025, https://milvus.io/ai-quick-reference/what-is-the-curse-of-dimensionality-and-how-does-it-affect-vector-search#:~:text=affect%20vector%20search%3F-,What%20is%20the%20curse%20of%20dimensionality%20and%20how%20does%20it,to%20become%20sparse%20and%20dissimilar.
6. What is the difference between k-NN and ANN in vector search? - Milvus, Zugriff am Dezember 7, 2025, https://milvus.io/ai-quick-reference/what-is-the-difference-between-knn-and-ann-in-vector-search
7. A Comprehensive Technical Analysis of Vector Database Architecture and

Similarity Search Algorithms | Uplatz Blog, Zugriff am Dezember 7, 2025, https://uplatz.com/blog/a-comprehensive-technical-analysis-of-vector-database-architecture-and-similarity-search-algorithms/
8. Understanding Recall in HNSW Search - Marqo, Zugriff am Dezember 7, 2025, https://www.marqo.ai/blog/understanding-recall-in-hnsw-search
9. What innovations are driving vector search scalability? - Milvus, Zugriff am Dezember 7, 2025, https://milvus.io/ai-quick-reference/what-innovations-are-driving-vector-search-scalability
10. Vector Databases & RAG: How AI Finds Answers in Milliseconds | by Abinaya Subramaniam, Zugriff am Dezember 7, 2025, https://pub.towardsai.net/vector-databases-rag-how-ai-finds-answers-in-milliseconds-2194f0ef3223
11. Managing Millions of High-Dimensional Vectors in Modern Vector Database - Medium, Zugriff am Dezember 7, 2025, https://medium.com/@bhargavaparv/managing-millions-of-high-dimensional-vectors-in-modern-vector-database-cbad318068fe
12. How does indexing work in a vector DB (IVF, HNSW, PQ, etc.)? - Milvus, Zugriff am Dezember 7, 2025, https://milvus.io/ai-quick-reference/how-does-indexing-work-in-a-vector-db-ivf-hnsw-pq-etc
13. Hierarchical Navigable Small Worlds (HNSW) - Pinecone, Zugriff am Dezember 7, 2025, https://www.pinecone.io/learn/series/faiss/hnsw/
14. Attribute Filtering in Approximate Nearest Neighbor Search: An In-depth Experimental Study, Zugriff am Dezember 7, 2025, https://arxiv.org/html/2508.16263v1
15. Vamana vs. HNSW - Exploring ANN algorithms Part 1 | Weaviate, Zugriff am Dezember 7, 2025, https://weaviate.io/blog/ann-algorithms-vamana-vs-hnsw
16. Approximate Nearest Neighbor (ANN) Search Explained: IVF vs HNSW vs PQ | TiDB, Zugriff am Dezember 7, 2025, https://www.pingcap.com/article/approximate-nearest-neighbor-ann-search-explained-ivf-vs-hnsw-vs-pq/
17. Locality Sensitive Hashing (LSH): The Illustrated Guide - Pinecone, Zugriff am Dezember 7, 2025, https://www.pinecone.io/learn/series/faiss/locality-sensitive-hashing/
18. Approximate Nearest Neighbor with Locality Sensitive Hashing (LSH) - PyImageSearch, Zugriff am Dezember 7, 2025, https://pyimagesearch.com/2025/01/27/approximate-nearest-neighbor-with-locality-sensitive-hashing-lsh/
19. Understanding Vector Databases | Microsoft Learn, Zugriff am Dezember 7, 2025, https://learn.microsoft.com/en-us/data-engineering/playbook/solutions/vector-database/
20. HNSW Indexing Fundamentals - Qdrant, Zugriff am Dezember 7, 2025, https://qdrant.tech/course/essentials/day-2/what-is-hnsw/
21. Understanding IVF Vector Index: How It Works and When to Choose It Over

HNSW - Milvus, Zugriff am Dezember 7, 2025, https://milvus.io/blog/understanding-ivf-vector-index-how-It-works-and-when-to-choose-it-over-hnsw.md
22. Accelerating Graph Indexing for ANNS on Modern CPUs - arXiv, Zugriff am Dezember 7, 2025, https://arxiv.org/html/2502.18113v1
23. Annoy vs HNSWlib on Vector Search - Zilliz blog, Zugriff am Dezember 7, 2025, https://zilliz.com/blog/annoy-vs-hnswlib-choosing-the-right-tool-for-vector-search
24. Experimental comparison of graph-based approximate nearest neighbor search algorithms on edge devices - Fraunhofer-Publica, Zugriff am Dezember 7, 2025, https://publica.fraunhofer.de/bitstreams/d8d7c1ff-02c8-4a6f-8357-b9d089b995a0/download
25. Choose the k-NN algorithm for your billion-scale use case with ..., Zugriff am Dezember 7, 2025, https://aws.amazon.com/blogs/big-data/choose-the-k-nn-algorithm-for-your-billion-scale-use-case-with-opensearch/
26. In what ways do tree-based indices (such as Annoy's random projection forests) differ from graph-based indices (like HNSW) in terms of search speed and recall? - Milvus, Zugriff am Dezember 7, 2025, https://milvus.io/ai-quick-reference/in-what-ways-do-treebased-indices-such-as-annoys-random-projection-forests-differ-from-graphbased-indices-like-hnsw-in-terms-of-search-speed-and-recall
27. Indexing - Qdrant, Zugriff am Dezember 7, 2025, https://qdrant.tech/documentation/concepts/indexing/
28. HNSW - Weaviate Knowledge Cards, Zugriff am Dezember 7, 2025, https://weaviate.io/learn/knowledgecards/hnsw
29. How do distributed vector databases handle sharding and replication? - Milvus, Zugriff am Dezember 7, 2025, https://milvus.io/ai-quick-reference/how-do-distributed-vector-databases-handle-sharding-and-replication