Research Q&A Bot by Benchmarking Vector Database Internals

An Empirical Study of ANNOY, HNSW, and FAISS-Flat for RAG Systems

Neehanth Reddy Maramreddy†
M.S. Data Science
University of Memphis
Memphis, TN, USA
neehanthreddym@gmail.com

ABSTRACT

Retrieval-Augmented Generation (RAG) systems rely on vector databases to supply relevant context to large language models (LLMs). While many approximate nearest neighbor (ANN) algorithms exist, their trade-offs in realistic RAG workloads are often poorly quantified. This paper presents a research Q&A bot over arXiv AI/ML papers and a reusable benchmarking harness that compares three ANN backends – ANNOY, HNSW, and FAISS-Flat – under a common interface. The system ingests multi-column PDFs, reconstructs reading order, computes sentence embeddings with `all-MiniLM-L6-v2`, and evaluates indexes on build time, memory usage, query throughput, and Recall@K using a brute-force ground-truth index. Experiments on multiple corpora ranging from 4,066 to 8,874 document chunks, with a highlighted dataset of 7,641 vectors and 76 queries visualized in Figure 3 and Figure 4, show that FAISS-Flat offers the fastest build time and smallest memory footprint, HNSW delivers near-perfect recall with moderate throughput, and ANNOY achieves very high throughput but near-zero recall due to a configuration bug. We discuss the implications of these results for small-to-medium-scale RAG systems and outline directions for extending the framework to additional ANN families and end-to-end RAG quality metrics.

KEYWORDS

Retrieval-Augmented Generation; Vector Search; Approximate Nearest Neighbors; HNSW; FAISS; ANNOY; Benchmarking; Question Answering

1 Introduction

Large language models are increasingly deployed with Retrieval-Augmented Generation (RAG) pipelines to answer questions grounded in external knowledge. In such systems, user queries are encoded as dense vectors, matched against a corpus in a vector database, and the retrieved passages are fed back to the model as context. Retrieval quality and latency directly shape user experience: slow retrieval makes the system feel unresponsive, while inaccurate retrieval leads to hallucinations even when the underlying documents contain the correct answer.

The literature on RAG typically focuses on model architectures and prompting, while treating the vector index as an opaque component. In practice, engineers building RAG applications must choose among approximate nearest neighbor (ANN) algorithms with different performance envelopes – graph-based structures such as Hierarchical Navigable Small Worlds (HNSW) [4], tree-based methods such as ANNOY, and flat or quantization-based schemes implemented in FAISS [5]. Each exposes its own hyperparameters and APIs, making controlled, apples-to-apples comparisons difficult.

This project addresses that gap by building:

- a research Q&A bot over arXiv AI/ML papers, and
- a benchmarking harness that isolates the vector search layer and evaluates three ANN backends – ANNOY, HNSW, and FAISS-Flat – under a shared interface.

The Q&A bot implements a standard RAG pipeline. Multi-column PDFs are parsed into coherent text chunks, sentence embeddings are computed with a pre-trained model, and a vector store powers retrieval for a FastAPI front-end. The harness factors out the vector store into a common interface and evaluates each ANN implementation on four metrics:

1. index build time,
2. memory usage,
3. query throughput (Queries Per Second, QPS), and
4. search accuracy (Recall@K) against a brute-force baseline.

The plots (Figures Figure 3 and Figure 4) visualize these metrics for a corpus of 7,641 vectors (384 dimensions) and 76 evaluation queries.

The main objectives are:

- Provide a modular, strategy-pattern abstraction that can host multiple ANN implementations without changing application code.
- Benchmark ANN algorithms under workloads derived from real research papers and questions.
- Make metric computations explicit and reproducible through clear formulas.
- Feed the findings back into the design of the Q&A bot.

The main contributions are:

- A RAG Q&A bot over arXiv AI/ML papers, with a robust PDF ingestion pipeline tailored to multi-column scientific layouts.

- A reusable ANN benchmarking harness exposing a unified `VectorStoreInterface` and four core metrics defined mathematically.
- An empirical comparison of ANNOY, HNSW, and FAISS-Flat, visualized in Figures Figure 3–Figure 4, highlighting their strengths and failure modes on realistic RAG workloads.
- Practical design guidance for choosing a vector index in small-to-medium-scale RAG systems.
- **Open-source implementation.** All code and experiment scripts are available at: https://github.com/neehanthreddym/doc_query_rag

2 Related Work

2.1 RAG and Question Answering

Retrieval-Augmented Generation (RAG) augments a parametric sequence-to-sequence model with a non-parametric document store to answer knowledge-intensive questions [1]. Follow-up systems explore different retrieval and fusion strategies, such as fusion-in-decoder and multi-vector retrieval [2][3]. These works usually report retrieval accuracy at the level of models and datasets, but they rarely expose low-level index metrics like build time or memory usage, which are critical for system design.

2.2 Approximate Nearest Neighbor Algorithms

Approximate nearest neighbor search has been studied extensively across tree-based, hashing-based, and graph-based methods. HNSW builds a multi-layer small-world graph and performs greedy search from upper layers to achieve high recall at logarithmic complexity [4]. FAISS provides a broad collection of indexes, including flat, inverted file (IVF), and product-quantized (PQ) structures, with both CPU and GPU backends [5]. ANNOY, developed at Spotify, uses a forest of random projection trees and is widely used in recommendation systems for its simplicity and practical performance.

Survey papers and system evaluations further analyze ANN algorithms. Wang et al. compare graph-based methods and show the advantages of HNSW-like structures across a range of workloads [6]. Azizi et al. study vector search in a database context, highlighting trade-offs such as graph connectivity vs memory footprint [7]. These works, however, target generic embedding workloads rather than RAG-specific document retrieval.

2.3 RAG-Oriented Vector Databases

Vector databases such as Milvus, Chroma, and others expose convenient APIs around ANN libraries, adding persistence, metadata filtering, and horizontal scalability. Higher-level frameworks like LangChain wrap those stores behind abstractions, but the choice of ANN algorithm is often exposed only as a configuration option. Benchmarks shipped with those systems are usually synthetic or vendor driven.

2.4 Positioning of This Work

This project differs from prior work in three ways:

1. It couples an ANN evaluation harness directly with a concrete RAG application over research literature, ensuring that embedding distributions, document lengths, and query styles reflect realistic usage.
2. It provides a transparent interface and metric suite, allowing new ANN algorithms to be plugged in and compared without rewriting application logic.
3. It grounds the discussion in visual evidence and explicit formulas: Figures 1–4 show the RAG pipeline, benchmarking workflow, and metric summaries, while Section 3.5 and Section 4.1 formalize the computations implemented in the code.

3 Solution

3.1 System Overview

The system consists of two coupled subsystems:

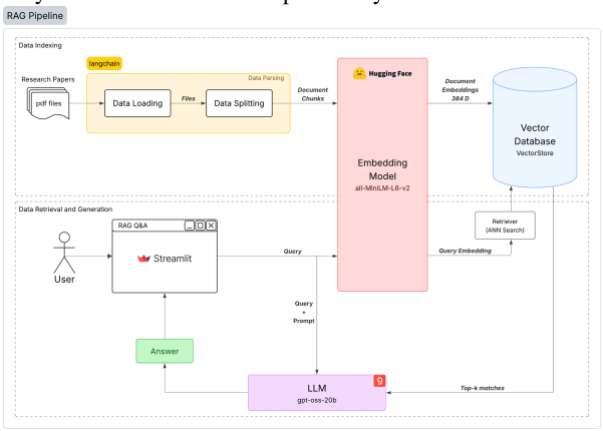

Figure 1: Overall architecture of the research Q&A bot, including PDF ingestion, chunking, embedding, vector search, and LLM answer generation.

1. **RAG Q&A Application**. A FastAPI service exposes an endpoint where users ask natural-language questions about AI/ML papers. The service retrieves relevant chunks from a vector store and calls a hosted LLM (gpt-oss-20b via the Groq API) to generate answers conditioned on retrieved context.
2. **ANN Benchmarking Harness**. A standalone module constructs embeddings for the same corpus and benchmarks ANN indexes implementing a shared `VectorStoreInterface`. It measures build time, memory usage, QPS, and Recall@K, and prints structured summaries that are visualized as bar charts in Figures Figure 3 and Figure 4.

Both subsystems operate on the same embeddings and differ primarily in how they use the vector store: the application serves end users, while the harness probes performance.

3.2 Data Ingestion and Chunking

The corpus is composed of PDF versions of AI/ML research papers downloaded from arXiv. Scientific PDFs frequently use two-column layouts, figures, and equations, which can scramble reading order if extracted naively.

The ingestion pipeline therefore:

1. Uses PyMuPDF to extract text blocks with bounding boxes.
2. Sorts blocks by page, vertical position, and then horizontal position to reconstruct the human reading order.
3. Cleans the text (e.g., fixing hyphenated line breaks and collapsing excessive whitespace).
4. Wraps cleaned passages into Document objects and splits them using a recursive character splitter with configurable chunk size and overlap.

For the main evaluation condition, the pipeline splits 80 PDFs into **7,641 text chunks**, each later embedded into a 384-dimensional vector. Other runs use different chunk sizes and subsets of the corpus, producing between 4,066 and 8,874 chunks.

3.3 Embedded Generation

The system uses the `all-MiniLM-L6-v2` model from Sentence-Transformers [8] to encode text chunks and queries into 384-dimensional vectors. This model is fast enough to embed several thousand chunks on a CPU-only machine in under a minute while maintaining reasonable semantic quality.

Given a text string t, the embedding manager applies the sentence-transformer $f(\cdot)$ and returns

$$x = f(t) \in \mathbb{R}^{384} \quad (1)$$

Embeddings are cached on disk to avoid re-computation. The same encoder is used for both document chunks and evaluation queries, ensuring consistency between index building and querying.

3.4 Vector Store Abstraction

At the heart of the harness is an abstract `VectorStoreInterface` that defines two core methods:

- `build(embeddings, documents)`
- `query(query_embedding, top_k)`

Concrete implementations include:

- **ANNOYVectorStore**. Wraps ANNOY with a configurable number of trees and a distance metric (e.g., "angular" or "euclidean").
- **HNSWVectorStore**. Wraps hnswlib with parameters such as max_connections and ef_construction. HNSW builds a layered proximity graph and supports fast approximate search [4].
- **FAISSVectorStore (Flat)**. Uses a FAISS Flat index as both the exact ground-truth baseline and an approximate index. The project originally attempted IVF+PQ and IVF-Flat, but both failed on the available hardware due to memory and threading errors; the final configuration uses Flat only.

A factory function instantiates the desired implementation given a name, allowing the rest of the code (including the RAG application) to depend only on the interface rather than any specific ANN library. This follows the Strategy pattern and makes it easy to plug in additional vector stores.

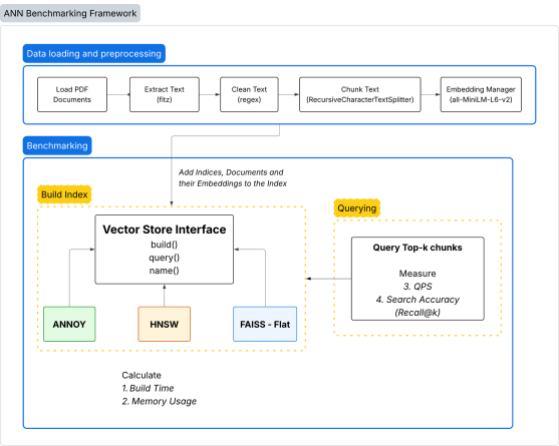

Figure 2: Architecture of the benchmarking framework. The pipeline prepares embeddings, builds multiple ANN indexes via a shared interface, and computes build time, memory, QPS, and Recall@K against a ground-truth index.

3.5 Ground-Truth Distance, Similarity, and Metric Computation

The metric computations in the codebase are simple but important, so it is useful to state them explicitly.

Let $\mathbf{q} \in \mathbb{R}^d$ be a query vector and $\mathbf{x}_i \in \mathbb{R}^d$ be the embedding of document i.

Cosine similarity. For stores that conceptually use cosine similarity (e.g., ANNOY in "angular" mode), the code normalizes vectors and computes

$$\cos(q, x_i) = \frac{q \cdot x_i}{\|q\|_2 \, \|x_i\|_2} \quad (2)$$

Ground-truth neighbors are obtained by sorting documents in descending order of this value.

Euclidean (L2) distance. For FAISS Flat in L2 mode, ground-truth distances are

$$d_2(q, x_i) = \|q - x_i\|_2 = \sqrt{\sum_{j=1}^{d} (q_j - x_{ij})^2} \quad (3)$$

and neighbors are sorted in ascending order of d_2.

Recall@K. For a given query, let R_K be the set of indices returned by the ANN store and G_K be the ground-truth top–K indices. The helper function `calculate_recall_at_k` implements

$$\text{Recall@K} = \frac{|R_K \cap G_K|}{|G_K|}, 0 \le Recall@K \le 1 \quad (4)$$

The harness computes averaged Recall@10 and Recall@100 across queries.

Queries Per Second (QPS). For each store, the function `measure_query_speed` records the time (in milliseconds) for each query, averages them to obtain $\bar{t}_{\text{ms}}$, and then computes

$$\text{QPS} = \frac{1000}{\bar{t}_{\text{ms}}} \quad (5)$$

This is equivalent to N/T, where $N$is the number of queries and $T$is the total time in seconds.

Memory overhead (MB). `measure_memory_usage` samples the process's resident set size (RSS) in bytes, converts it to megabytes, and the benchmark takes the difference before and after index construction:

$$M_{index} = \frac{RSS_{after} - RSS_{before}}{1024^2} \quad (6)$$

3.6 Integration into the RAG Bot

The RAG application uses ChromaDB as its production vector store, configured with an HNSW backend. The live query path is:

1. Receive a question via FastAPI.
2. Encode the question using the same sentence-transformer to obtain $\mathbf{q}$.
3. Query ChromaDB/HNSW for the top–K chunks, which returns distances $d_i$for each retrieved document.
4. Convert distances to similarity scores via
$$s_i = 1 - d_i \quad (7)$$
assuming a cosine-distance metric.
5. Select a **confidence score** for the answer as
$$conf = \max_i s_i \quad (8)$$
and pass the top documents, question, and confidence score through the `RAGPipeline`.
6. Concatenate the question and retrieved chunks into a prompt and send it to the LLM, streaming the answer back to the client.

Figure 1 shows the RAG Q&A pipeline from PDF ingestion to LLM response, and Figure 2 shows the benchmarking pipeline from corpus preparation through metric reporting.

4 Evaluation

4.1 Experiment Setting

Datasets and Workloads

The harness is run on several corpus variants: **Corpus sizes**. 4,066; 4,827; 7,641; 8,278; 8,785; and 8,874 vectors (all 384-dimensional), **Query sets**. Depending on the run, between 6 and 76 natural-language queries are used. Queries are drawn from a `test_queries.txt` file with research-style questions.

The **primary condition visualized in Figures Figure** 3**–Figure** 4 uses: **7,641 vectors**, 384 dimensions, **76 queries**, and `K=10` and `K=100` for Recall@K as defined in Section 3.5.

For each query $\mathbf{q}$, the ground-truth neighbors $G_{10}$and $G_{100}$are computed using either cosine similarity or L2 distance depending on the store type, following the formulas above. The ANN store returns retrieved indices R_{10}(and a larger set for recall@100), and the code applies the Recall@K formula to obtain per-query scores before averaging.

Metrics and Baselines

For each corpus and query set, the following are recorded: build time in seconds, memory usage in MB, query throughput (QPS), Recall@10 and Recall@100.

The FAISS Flat index serves as the **exact ground truth** for Recall@K and is also evaluated as an approximate index to quantify its cost relative to HNSW and ANNOY.

Hardware and Software

The artifacts do not record detailed hardware specifications. Repeated failures of IVF+PQ and IVF-Flat due to memory allocation and OpenMP errors suggest a single CPU machine with modest RAM (likely ≤16 GB) and no GPU acceleration. Python 3, Sentence-Transformers, FAISS CPU, hnswlib, and Annoy are used. Because the hardware is under-specified, absolute timings should be interpreted as indicative rather than definitive; relative comparisons under the same environment are more reliable.

4.2 Results

Build Time and Memory

Figure 3 summarizes build time and memory usage for the 7,641-vector corpus and 76-query workload:

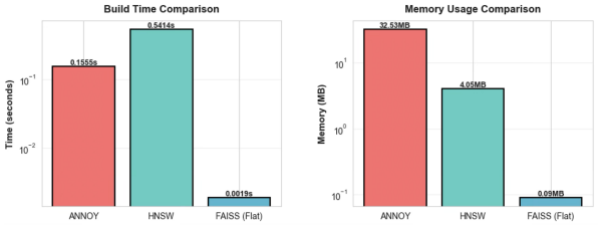

Figure 3: Build time and memory usage for ANNOY, HNSW, and FAISS-Flat on a corpus of 7,641 vectors (384-dimensional) and 76 queries.

- **FAISS-Flat** builds the index in **0.002 s** and uses **0.09 MB** of additional memory, corresponding to a very small M_{index}.
- **HNSW** takes about **0.54 s** to build and uses around **4.19 MB**.
- **ANNOY** takes about **0.15 s** to build but uses roughly **29.47 MB**.

Across other corpus sizes, the same pattern appears in the logs: FAISS-Flat is consistently the fastest to build and has the smallest memory footprint; HNSW is moderately slower with modest memory usage; ANNOY's memory footprint grows fastest with corpus size.

In practical terms, this means that FAISS-Flat is ideal for rapid iteration and debugging, while HNSW and ANNOY introduce overhead that must be justified by either better accuracy or higher QPS.

Query Throughput and Accuracy

Figure 4 presents query throughput and Recall@K for the same 7,641-vector, 76-query setting:

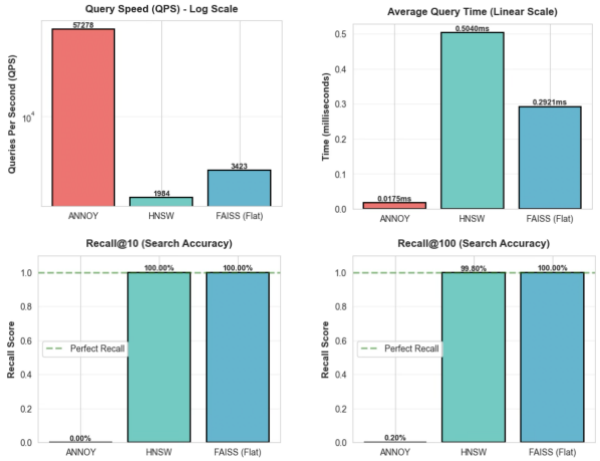

Figure 4: Query throughput (QPS) and Recall@K for ANNOY, HNSW, and FAISS-Flat on the 7,641-vector, 76-query workload.

ANNOY achieves the highest throughput at approximately **56,309 QPS**, which corresponds to an average query time of $\bar{t}_{\text{ms}} \approx 0.018$ms via the QPS formula. However, it has **Recall@10 = 0.0** and **Recall@100 ≈ 0.002**, indicating that $|R_K \cap G_K|$ is almost always zero in the Recall@K formula.

HNSW achieves around **1,439 QPS** (average query time ≈ 0.69ms) with **Recall@10 = 1.0** and **Recall@100 = 1.0** in the highlighted run (and ≥0.998 across other runs). Here, the set intersection $R_K \cap G_K$ equals G_K for almost all queries.

FAISS-Flat achieves about **3,214 QPS** (average query time ≈ 0.31ms) with **Recall@10 = 1.0** and **Recall@100 = 1.0**, matching the ground truth by definition.

The plots make this trade-off visually clear: ANNOY's bar towers over others in QPS but collapses in the recall charts, whereas HNSW and FAISS show lower but still ample QPS with full-height recall bars.

ANNOY's recall remains near zero, HNSW's recall is effectively perfect, and FAISS-Flat maintains perfect recall with somewhat lower QPS than HNSW but still in a regime suitable for interactive use.

Additional Observations

IVF+PQ and IVF-Flat failures. Attempts to benchmark FAISS IVF+PQ and IVF-Flat resulted in segmentation faults or OpenMP errors during index construction. This suggests that these more complex indexes exceed available memory or thread resources on the test machine. As a result, they are excluded from Figures Figure 3 and Figure 4.

Best-performer summaries. The harness prints textual "best performer" summaries after each run, derived directly from the metric values defined in Section 3.5. For build time and memory, FAISS-Flat is almost always the winner. For QPS, ANNOY wins in every case, but its recall figures make those wins meaningless for retrieval quality.

4.3 Discussion / Threats to Validity

HNSW as a robust default. The bar charts in Figures Figure 3–Figure 4, combined with the Recall@K and QPS definitions, justify using HNSW (via ChromaDB) as the production vector store for the research Q&A bot. It offers near-perfect recall with sub-millisecond latency and modest memory usage on the tested corpora.

FAISS-Flat as a strong baseline. FAISS-Flat's extremely fast build time and negligible memory overhead make it ideal for experimentation, small deployments, and debugging. Thanks to its exact L2 search, Recall@K is trivially 1.0, and the observed QPS is more than sufficient for user-facing workloads.

ANNOY's configuration bug. ANNOY's high QPS but near-zero recall indicate a serious misconfiguration (e.g., mismatch in vector dimensionality, distance metric, or ID handling). Plugging the ANN outputs into the Recall@K formula makes this failure obvious; without explicit accuracy metrics, one might mistakenly choose ANNOY based on throughput alone.

There are, however, several threats to validity:

Hardware ambiguity. Because CPU model, core count, and memory size are not recorded, absolute QPS and build times cannot be generalized. GPU-accelerated FAISS would likely alter relative performance dramatically.

Scale limitations. The largest corpora considered (≈8–9k vectors) are small compared to production RAG systems, which may contain millions of vectors. Graph-based and quantized indexes behave differently at larger scales; Figure 3 and Figure 4 therefore capture only the small-to-medium regime.

Single encoder and domain. All experiments use a single sentence-transformer and a corpus of AI/ML research papers. Different embedding models or domains might shift the distribution of distances and, in turn, ANN performance.

Incomplete ANNOY evaluation. Until the configuration bug is fixed, this work cannot claim to have meaningfully compared ANNOY to HNSW and FAISS. The current results are best interpreted as a cautionary example of how ANN indexes can silently fail.

Lack of end-to-end quality metrics. The evaluation focuses on index-level metrics, not user-visible answer quality. There is good reason to believe that high recall correlates with better answers, but the relationship is not quantified here.

The original plan included ScaNN and FAISS IVF+PQ, but resource constraints prevented a stable integration; these remain open items for future iterations of the harness.

5 Conclusion and Future Work

This paper presented a research Q&A bot over arXiv AI/ML papers and a benchmarking harness for evaluating vector database internals in RAG systems. By defining a shared `VectorStoreInterface`, using a FAISS-Flat ground-truth

index, and formalizing metrics for build time, memory usage, QPS, and Recall@K, the project delivers both a usable application and a reusable evaluation framework. Experiments show that FAISS-Flat combines the fastest build time and smallest memory footprint with perfect recall, while HNSW provides near-perfect recall (≥0.998) with QPS in the low thousands and moderate memory usage. ANNOY, despite extreme throughput, yields near-zero recall in its current configuration, and FAISS IVF+PQ and IVF-Flat could not be evaluated due to hardware-related errors. For small-to-medium-scale RAG deployments on CPU-only machines, these results support using FAISS-Flat for experimentation and HNSW for production, and they highlight high QPS with low recall as a clear warning signal.

Future work includes debugging the ANNOY configuration and re-running the benchmarks, extending the framework to additional ANN families (e.g., IVF+PQ when feasible, ScaNN, disk-backed indexes), and scaling evaluations to corpora with millions of vectors and more diverse query workloads. A natural next step is to add end-to-end assessment of answer quality – such as faithfulness, factual accuracy, and user satisfaction – and to automate hyperparameter search for indexes like HNSW and ANNOY to better balance recall, latency, and memory. Treating the vector database as an empirical object rather than a black box, and grounding its behavior in explicit metrics, moves RAG system design toward more transparent and data-driven choices.

For the research Q&A bot, these findings concretely justified deploying an HNSW-backed ChromaDB index in production and using a small FAISS-Flat index as a debugging tool during development.

REFERENCES

[1] Patrick Lewis, Ethan Perez, Aleksandra Piktus, et al. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*. DOI:https://doi.org/10.48550/arXiv.2005.11401

[2] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. In *Findings of EMNLP*. DOI:https://doi.org/10.48550/arXiv.2004.05150

[3] Sebastian Hofstätter, Sophia Althammer, Michael Schröder, et al. 2021. Efficiently Teaching an Effective Dense Retriever with Balanced Topic Aware Sampling. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. DOI:10.48550/arXiv.2104.06967

[4] Yury A. Malkov and Dmitry A. Yashunin. 2018. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. DOI:https://doi.org/10.48550/arXiv.1603.09320

[5] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*. DOI:https://doi.org/10.48550/arXiv.1702.08734

[6] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. Proceedings of the VLDB Endowment 14, 11, 1964–1978. DOI:https://doi.org/10.48550/arXiv.2101.12631

[7] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2025. Graph-Based Vector Search: An Experimental Evaluation of the State of the Art. *Proceedings of the ACM on Management of Data* 3, 1, Article 43. DOI:https://doi.org/10.48550/arXiv.2502.05575

[8] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*. DOI:https://arxiv.org/pdf/1908.10084