Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

A Comparative Study of Vector Indexing Strategies Using Facebook AI Similarity Search as a Case Study

Dukhik Hovsepyan

dukhik.hovsepyan@gmail.com

European University of Armenia

Hamlet Harutyunyan

European University of Armenia

Research Article

Keywords: Facebook AI Similarity Search, Approximate Nearest Neighbor, high-dimensional vector indexing, inverted file indexing, product quantization, Hierarchical Navigable Small World, vector similarity search.

Posted Date: September 29th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9962353/v2

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

A Comparative Study of Vector Indexing Strategies Using Facebook AI Similarity Search as a Case Study

Dukhik Hovsepyan[1*] and Hamlet Harutyunyan[1]

[1*]Chair of Information Technologies and Applied Mathematics, European University of Armenia, 10 Davit Anhaght Str., Yerevan, 0037, Armenia.

*Corresponding author(s). E-mail(s): dukhik.hovsepyan@gmail.com;
Contributing authors: hamhar1945@gmail.com;

Abstract

Efficient similarity search in high-dimensional vector spaces has become a core component of many modern AI applications, such as semantic search, recommendation systems, fraud detection, and embedding-based large language models. As data volumes grow, selecting an appropriate indexing strategy becomes critical for balancing accuracy, query latency, and memory usage. Facebook AI Similarity Search offers a diverse set of index structures designed to address these trade-offs under different workload conditions. This study provides a structured experimental evaluation of several widely used Facebook AI Similarity Search index types: IndexFlatL2, IndexFlatIP, IndexIVFFlat, IndexIVFPQ, IndexIVFSQ, and IndexHNSW. The analysis investigates their computational behavior, recall performance, and sensitivity to various distance metrics, including L1, L2, inner product, and cosine similarity. It also explores the effects of quantization techniques, clustering strategies, and graph-based connectivity parameters on overall efficiency. The results highlight clear differences in performance patterns across index families and offer practical guidance for selecting appropriate configurations in real-world, large-scale deployment scenarios.

Keywords: Facebook AI Similarity Search, Approximate Nearest Neighbor, high-dimensional vector indexing, inverted file indexing, product quantization, Hierarchical Navigable Small World, vector similarity search.

1 Introduction

Vector databases have become a cornerstone of modern machine learning infrastructure, enabling efficient similarity search across high-dimensional data. Unlike traditional databases that query based on exact matches, vector databases allow for semantic search, finding items that are "conceptually similar" rather than identical. The core challenge in vector similarity search is speed and scalability. Searching for the nearest neighbors in a dataset of millions or billions of vectors is computationally expensive. To address this, various indexing strategies have been developed to balance the trade-offs between search speed, accuracy (recall), and memory consumption. FAISS (Facebook AI Similarity Search) is a library developed by Facebook AI Research for efficient similarity search and clustering of dense vectors [1]. It contains algorithms that search in sets of vectors of any size, up to ones that possibly do not fit in RAM. It also contains supporting code for evaluation and parameter tuning. Similarity search in high-dimensional spaces underpins numerous AI-driven applications. As embedding models produce vectors of dimension 128-1536 and beyond, brute-force nearest neighbor search becomes computationally prohibitive at scale. Modern dense retrieval systems use learned text representations for open-domain question answering and semantic passage search [2, 3]. Retrieval-augmented generation systems likewise depend on a retrieval stage whose latency, memory footprint, and ranking quality affect the downstream application [4]. Recent embedding evaluations show that no single encoder or retrieval configuration dominates across all tasks [5]; this motivates evaluating index choices under more than one vector distribution. FAISS provides scalable solutions via:

- Exact search (Flat indexes)
- Clustering-based search (IVF)
- Quantization-based compression (PQ, SQ)
- Graph-based navigation (HNSW)

This paper evaluates these techniques experimentally.

2 Distance Metrics For Vector Search

In this study, several distance metrics are examined for vector search, including Euclidean (L2), Manhattan (L1), Inner Product, and Cosine similarity. Since FAISS does not directly support cosine similarity, it is computed by first normalizing vectors and then using the inner product. Comparing these metrics helps highlight their effect on search accuracy, recall, and overall performance. FAISS natively supports Euclidean distance (L2) and Inner Product (IP). However, it also supports L1 (Manhattan) distance via specific configuration.

- **Euclidean (L2)**: measures the straight-line distance between two points.

$$d_{L2}(x, y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2} \tag{1}$$

- **Manhattan (L1)**: sum of absolute differences between vectors.

$$d_{L1}(x, y) = \sum_{i=1}^{d} |x_i - y_i| \tag{2}$$

- **Inner Product**:

$$x \cdot y = \sum_{i=1}^{d} x_i y_i \tag{3}$$

- **Cosine Similarity**: FAISS does not implement cosine similarity directly. It is achieved via L2 normalization and then applies Inner Product: To compute cosine similarity in FAISS, vectors must be normalized before indexing and querying. This works because the inner product of two normalized vectors is mathematically identical to their cosine similarity.
  Since:

$$\cos(x, y) = \frac{x \cdot y}{\|x\| \, \|y\|} \tag{4}$$

  If $\|x\| = 1$ and $\|y\| = 1$, then

$$\cos(\theta) = x \cdot y \tag{5}$$

  Normalization makes inner product equivalent to cosine similarity. This equivalence is well established in vector space retrieval models [6].

3 FAISS Index Types

The main index types supported by Facebook AI Similarity Search (FAISS) are:

Table 1 FAISS index types

Index Type	Description
IndexFlatL1 / IndexFlatL2	Exact search with Manhattan (L1) or Euclidean (L2) distance
IndexFlatIP	Exact search with inner product (or cosine similarity if vectors are normalized)
IndexIVFFlat	Inverted file (clustering-based) index for faster approximate search
IndexIVFPQ	IVF combined with Product Quantization for memory-efficient approximate search
IndexIVFSQ	IVF combined with Scalar Quantization
IndexHNSW	Hierarchical Navigable Small World graph for fast approximate search (L2 or IP)

1. **IndexFlatL2**: stores the vectors exactly as they are without compression or clustering. It performs exact brute-force search with time complexity $O(Nd)$ and memory usage $4Nd$ bytes.

   For Manhattan (L1) distance, FAISS supports `IndexFlat` with `METRIC_L1`:

$$L_1(x, y) = \sum_{i=1}^{d} |x_i - y_i| \tag{6}$$

Lower values indicate higher similarity between vectors; the smaller the distance, the more similar the items.

2. **IndexFlatIP**: computes similarity using the inner product.

   FAISS supports both normalized and non-normalized vectors, depending on the similarity definition.

   For non-normalized vectors, the inner product is defined as:

$$x \cdot y = \sum_{i=1}^{d} x_i y_i \tag{7}$$

   In this case, similarity depends not only on the angular relationship between vectors but also on their magnitude.

   For L2-normalized vectors, the inner product becomes equivalent to cosine similarity:

$$\cos(x, y) = x \cdot y \tag{8}$$

Table 2 Inner Product similarity values

Dimension	Expected value
64	~ 21.3
128	~ 42.7
256	~ 85.3
768	~ 256
1024	~ 341
1536	~ 512

3. **IndexIVFFlat**: speeds up search by narrowing down the scope. It is a clustering-based index where vectors are stored exactly inside clusters. Method is clustering-based inverted file index. This requires training. Here are the steps for applying IVF index:

   - **Training / Clustering** FAISS runs k-means clustering on the training vectors to define *nlist* clusters (Voronoi cells). Each cluster has a centroid.
   - **Adding vectors** Each database vector is assigned to the nearest cluster centroid and stored in that bucket.
   - **Searching** The query is compared to all centroids to find the closest clusters. The search is then performed exactly (brute-force) only within those selected clusters.

   Search complexity is given by this formula:

$$O\left(nprobe \cdot \frac{N}{nlist} \cdot d\right) \tag{9}$$

Several parameters are critical when setting up FAISS indexes, as they directly impact search speed and accuracy:

(a) **nlist**
(b) **nprobe**
(c) **metric**
(d) **quantizer**

Here are the brief descriptions of each parameter.

(a) nlist - Refers to the number of clusters (Voronoi Cells), in an IVF index. It defines the number of centroids created during the training (k-means clustering) phase. Mathematical Role: Given a dataset of size N, the space is partitioned into nlist clusters. Each vector is assigned to its nearest centroid. On average, each cluster contains $\frac{N}{nlist}$ vectors. A common empirical rule is:

$$nlist \approx \sqrt{N} \tag{10}$$

In an IVF index, the dataset is divided into a number of clusters, each represented by a centroid. Every vector is assigned to the cluster with the nearest centroid. On average, each cluster will contain a certain number of vectors depending on the total dataset size and the number of clusters.

A practical rule of thumb for choosing *nlist* is as follows:

- **Larger** *nlist*
  - Produces smaller clusters
  - Speeds up searches within each cluster
  - Increases indexing and training time
  - Can improve recall if enough clusters are probed
- **Smaller** *nlist*
  - Creates larger clusters
  - Queries take longer to process
  - Reduces memory requirements for storing centroids

Careful selection of *nlist* is important to balance search efficiency, memory usage, and retrieval accuracy in large-scale similarity search tasks.

(b) nprobe - Specifies how many clusters are searched during a query. Examining more clusters usually boosts search accuracy, but it can also make queries slower.

During search, the query vector is compared only against vectors inside the nearest *nprobe* clusters.

$$1 \leq nprobe \leq nlist \tag{11}$$

Search complexity becomes:

$$O\left(nprobe \cdot \frac{N}{nlist} \cdot d\right) \tag{12}$$

- **Higher** *nprobe*

  - Improves recall
  - Increases query time
- **Lower** *nprobe*
  - Speeds up search
  - Reduces recall

When $nprobe = nlist$, the index effectively performs an exact search. As such, *nprobe* serves as the main parameter for controlling the trade-off between accuracy and query latency.

(c) metric - Determines how the similarity between vectors is calculated. Typical options include L2 (Euclidean distance), inner product, or cosine similarity (if vectors are normalized). Choosing the right metric ensures that the results reflect meaningful similarity for the task at hand.

Supported metrics in FAISS are:

- `METRIC_L2` (Euclidean distance)
- `METRIC_INNER_PRODUCT`
- `METRIC_L1` (Manhattan distance)

For cosine similarity, normalized vectors are used with inner product:

$$\text{normalized vectors} \Rightarrow \text{use Inner Product}$$

The metric affects:

- Cluster formation
- Vector assignment
- Query ranking

The choice of distance metric should align with the training objective of the embedding model. For example:

- Sentence Transformers typically use cosine similarity
- Embeddings trained with Euclidean distance use L2 distance
- Applications requiring maximum similarity retrieval rely on inner product

Selecting the appropriate metric ensures that similarity scores accurately reflect the relationships learned by the embedding model.

(d) quantizer - Defines the method used to assign vectors to clusters in IVF-based indexes. The type of quantizer influences both the efficiency of search and the accuracy of the retrieved results.

The quantizer is itself a FAISS index used to perform k-means clustering and centroid search. It is very important, because it performs the following functions:

- Computes nearest centroid during training
- Assigns database vectors to clusters

The quantizer determines:

- Clustering quality
- Partition boundaries
- Assignment precision

A poor quantizer results in:

- Unbalanced clusters
- Recall degradation

In most CPU use cases, IndexFlatL2 is sufficient. It finds the closest centroids at query time. Table 3 shows the affect of different parameters:

Table 3 Key FAISS Index Parameters and Their Effects

Parameter	Controls	Affects	Primary Trade-off
nlist	Number of clusters	Cluster size	Speed vs training cost
nprobe	Clusters searched	Recall	Accuracy vs latency
metric	Distance function	Ranking behavior	Correctness of similarity
quantizer	Centroid computation	Cluster quality	Recall stability

In this study L2 metric and IndexFlatL2 quantizer are taken for further calculations and analysis.

4. **IndexIVFPQ**:Inverted File with Product Quantization for Memory-Efficient Approximate Search.

   IndexIVFPQ is often the preferred choice when working with very large datasets and limited memory resources. By combining inverted file indexing with product quantization, it compresses vectors into a compact representation, which greatly reduces storage requirements. Although this compression introduces a small loss in accuracy, it allows the system to scale efficiently and handle millions of vectors without excessive memory consumption.

   IndexIVFPQ is exactly what you use when memory efficiency and large scale really matters. Adds Product Quantization. Vector is split into subvectors as originally proposed in [7]:

$$d = m \cdot d' \tag{13}$$

   Each subvector is encoded with $nbits$.

   The following key parameters are used:

   - nq - query vectors
   - nlist - IVF clusters
   - nprobe - clusters searched
   - m - number of PQ sub-vectors (rule: $d \bmod m = 0$)
   - nbits: bits per sub-vector (code size), with typical values 4, 6, or 8

   Memory:

$$\frac{N \cdot m \cdot nbits}{8} \tag{14}$$

   Typically compression is produced from 4 to 16 times. This compression principle follows the PQ framework described in [7].

5. **IndexIVFSQ**: this can be viewed as a middle-ground solution between IVFFlat and IVFPQ. It is less complex than Product Quantization, yet more memory-efficient than the standard flat inverted index. This balance makes it easier to implement and analyze, while still offering meaningful storage savings compared to fully uncompressed approaches. scalar Quantization works the following way: for each dimension:

- Computes min/max or statistical range
- Quantizes values into fixed bins (e.g., 8-bit → 256 bins)

So, instead of storing float32 (4 bytes) × dim memory, you store uint8 (1 byte) × dim, which is approximately 4× memory reduction.

It is simpler than PQ but less compressive.

This approach is grounded in classical scalar quantization theory [8]. The following key parameters are used:

- nq: number of query vectors
- $nlist$: number of IVF clusters
- $nprobe$: number of clusters searched
- $qtype$: type of scalar quantizer

Supported quantizers in FAISS are:

- 4, 6, 8, 4u, 8u, 8d, 8ds, fp16, bf16

6. **IndexHNSW**: This is an approximate nearest neighbor method that relies on a hierarchical graph structure. The method follows the algorithm introduced in [Y. A. Malkov and D. A. Yashunin][9]. Instead of searching through all data points, it organizes them into several layers, where the upper layers contain fewer points and provide a broader overview of the dataset. The search begins at the top layer and moves downward, using a greedy strategy to gradually focus on the most promising neighbors. By limiting the number of comparisons at each step, HNSW achieves both high accuracy and fast query times, making it especially suitable for large, high-dimensional datasets.

   Unlike IVF (clustering-based), HNSW organizes vectors into a multi-layer proximity graph, enabling efficient greedy navigation toward nearest neighbors.

   The following key parameters are used:

   - M: graph connectivity
   - $efConstruction$: construction-time search width
   - $efSearch$: query-time search width
   - Metric: distance function used for similarity computation

   (a) M - Graph Connectivity (Maximum Number of Neighbors per Node) M defines the maximum number of bi-directional connections each node (vector) maintains at each layer of the graph.

   $$M = \text{max neighbors per node} \tag{15}$$

   Structural role of M includes:
   - determining graph density
   - controlling navigability
   - affecting memory usage

   The average memory complexity becomes $O(N \cdot M)$, since each vector stores approximately M connections.
   - **Larger M results in:**
     – Denser graph

  - Higher recall
  - Increased memory consumption
  - Slower construction
- **Smaller M results in:**
  - Sparse graph
  - Lower memory usage
  - Faster indexing
  - Reduced recall

Higher M improves the probability that greedy search avoids local minima in high-dimensional space.

(b) $efConstruction$ - Construction-Time Candidate List Size $efConstruction$ controls the size of the dynamic candidate list during graph building.

$$efConstruction \geq M \tag{16}$$

It determines how thoroughly each new vector explores the graph before establishing connections.

- **Larger $efConstruction$ results in:**
  - Higher-quality graph
  - Better long-range links
  - Higher recall
  - Slower indexing time
- **Smaller $efConstruction$ results in:**
  - Faster build time
  - Lower-quality connectivity
  - Potential recall degradation

Empirical observations show that the best results are obtained when:

$$efConstruction \approx 2 \times M \quad \text{(minimum)} \tag{17}$$

And for serious benchmarks:

$$efConstruction = 100 \text{ to } 400 \tag{18}$$

Indexing complexity roughly scales as:

$$O(N \cdot efConstruction \cdot \log N) \tag{19}$$

(c) $efSearch$ - Search-Time Candidate List Size.

$efSearch$ controls how many candidate neighbors are explored during query time.

$$efSearch \geq k \tag{20}$$

where k is the number of nearest neighbors requested.

It defines how deeply the search explores the graph beyond greedy navigation.

- **Larger $efSearch$ results in:**
  - Higher recall
  - Slower query time
  - More stable ranking

- **Smaller $efSearch$ results in:**
  - Faster queries
  - Lower recall

The evaluated $efSearch$ values, 16, 64, 128, and 256, represent increasing candidate-list budgets. Their effect on Recall@k is dataset-dependent and is therefore reported from exact-ground-truth measurements rather than qualitative labels. Search complexity approximates:

$$O(efSearch \cdot \log N) \tag{21}$$

This logarithmic behavior is consistent with the theoretical analysis in [9].

(d) *metric* - Similarity Measure For FAISS, the HNSW index supports the following metrics:

- METRIC_L2
- METRIC_INNER_PRODUCT
- Cosine similarity (via vector normalization)

The metric determines:

- Graph connectivity structure
- Neighbor selection
- Ranking consistency

As in IVF, metric choice must align with the embedding training objective.

Table 5 shows the parameter interaction summary. Search complexity is

Table 4 Parameter Interaction Summary

Parameter	Controls	Affects	Trade-off
M	Graph connectivity	Recall and memory	Memory vs accuracy
$efConstruction$	Graph quality	Final recall	Build time vs quality
$efSearch$	Search depth	Query recall	Latency vs accuracy
Metric	Similarity measure	Ranking behavior	Correctness

$O(\log N)$. Empirically, it achieves very high recall.

4 Experimental Implementation

Backend: Python + FAISS

Frontend: C# performance visualization tool

Dataset:

- Random vectors
- Dimension: 64–1536
- Database size: up to 10 million
- Query size: up to 1000

Hardware:

- CPU: Intel i7-12700H, base frequency 2.3 GHz, 14 cores

- RAM: 32 GB
- OS: Windows 11

4.1 Reproducible evaluation protocol

The benchmark stores returned identifiers and constructs an exact-search ground truth for every query. Nearest-neighbor distances are reported separately from retrieval recall. One fixed database and one fixed query set are used for every index configuration, generated from a specified distribution with a fixed random seed.

For each query q_i, let G_i^k be the set of the k identifiers returned by an exact flat index and let A_i^k be the identifiers returned by an approximate index. Recall is defined as

$$\text{Recall@}k = \frac{1}{Q} \sum_{i=1}^{Q} \frac{|A_i^k \cap G_i^k|}{k}, \tag{22}$$

where Q is the number of queries. Recall@1, Recall@10, and Recall@100 are reported when the requested neighborhood contains at least 100 neighbors. The exact index is the accuracy baseline and has Recall@k equal to one, apart from possible tie-ordering effects.

Search latency is measured with batch search after one warm-up pass and is reported as total query time, milliseconds per query, and throughput. Index construction time is measured through training and adding the database vectors. Serialized index size is reported in bytes and megabytes. The revised benchmark records the FAISS and NumPy versions, operating system, distribution, seed, IVF/PQ/SQ settings, and HNSW settings; each synthetic configuration is repeated ten times and summarized by its median, standard deviation, and 95% confidence interval for latency.

The synthetic component is a controlled index benchmark. Claims about semantic retrieval, recommendation, healthcare search, and retrieval-augmented generation are application motivation rather than direct validation. To add a real-text validation, we also encode a bounded subset of the 20 Newsgroups training corpus using Sentence-BERT-style sentence embeddings [5, 10]. The text is cleaned by removing headers, footers, and quotations; documents shorter than 80 characters are discarded. We use 3,000 database documents and 300 query documents, encode them with `all-MiniLM-L6-v2`, normalize the 384-dimensional embeddings, and use inner product, which is equivalent to cosine similarity after normalization. The same exact-ground-truth protocol is used, with $nlist = 32$, $nprobe = 8$, $M = 16$, $m = 32$, and 8-bit PQ. This is an embedding-space retrieval benchmark rather than a supervised classification or question-answer evaluation: the query documents have no relevance labels, so Recall@k measures agreement with the exact nearest-neighbor ranking induced by the encoder.

The real-text results are shown in Table 6. They demonstrate that the conclusions depend on the embedding distribution. In particular, IVFFlat, IVFSQ, and HNSW retain high agreement with the exact baseline, whereas IVFPQ has a larger accuracy loss under the selected compression settings.

5 Experimental Results

5.1 Real-text embedding validation

The real-text experiment provides an embedding-level validation using a public text corpus and a contemporary sentence encoder. It should not be interpreted as a complete semantic-retrieval benchmark because the query documents do not have human relevance judgments. It instead tests whether the index trade-offs observed with synthetic vectors persist for normalized text embeddings. The results in Table 6 show that HNSW and IVFFlat preserve high agreement with the exact ranking, while IVFPQ provides the strongest size reduction with a larger Recall@10 loss.

5.2 Exact Search Baseline

As expected from an exhaustive search method, the recall remained 100% in all test scenarios. Since this index performs a full comparison between the query vector and every vector in the dataset, it guarantees exact results and therefore serves as a reliable performance baseline for further comparisons. The latency measurements reveal a consistent pattern across all experiments. When the dataset size increases from 1,000 to 10,000 vectors, the query time also increases for both embedding dimensions (128 and 512). This trend is clearly visible in Figure 1. The growth in latency follows the expected behavior of a brute-force search approach: because every additional vector must be compared during query execution, the computational cost rises proportionally with dataset size. In other words, scalability is directly tied to the number of stored vectors.

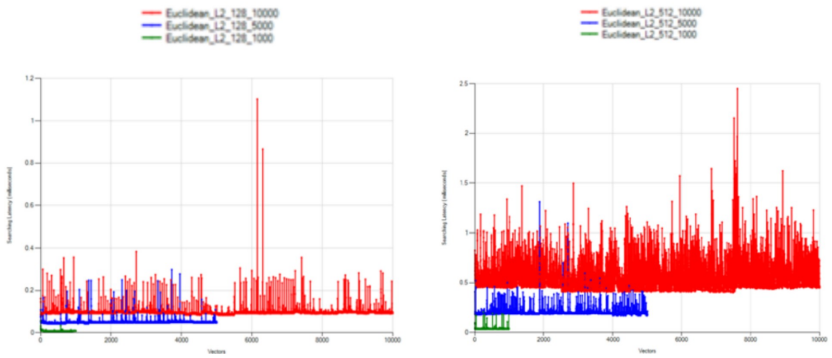


Fig. 1 Query Latency for IndexFlatL2, dataset sizes: 1000, 5,000, 10,000,dimension: 128,512

When comparing embedding dimensions more closely (Figure 2), an additional pattern emerges. For each fixed dataset size, the 512-dimensional embeddings consistently require more time than the 128-dimensional ones. This difference is not surprising, as higher-dimensional vectors involve a greater number of arithmetic operations during L2 distance calculation. The gap between the two dimensions becomes more noticeable

as the dataset grows, suggesting that dimensionality amplifies the impact of dataset size on performance.

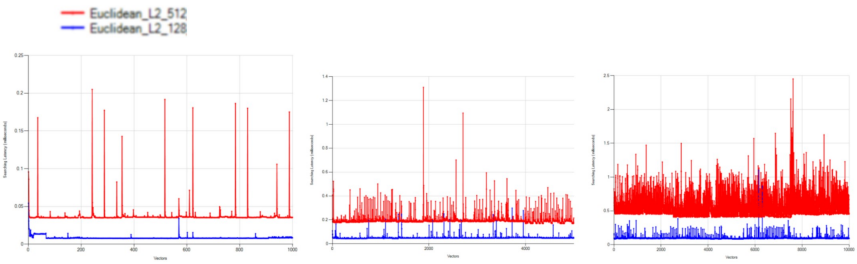


Fig. 2 Query Latency comparison of dimensions 128 and 512 for IndexFlatL2,dataset size: 1,000; 5,000; 10,000

Several important conclusions can be drawn from these results:

- Exact search ensures perfect recall but does not scale efficiently.
- Query latency increases steadily as the dataset becomes larger.
- Higher-dimensional embeddings introduce additional computational overhead.
- The combined effect of large datasets and high dimensionality leads to more pronounced performance degradation.

Overall, while IndexFlatL2 provides maximum retrieval accuracy, the observed latency behavior highlights its practical limitations for large-scale deployments. These findings reinforce the motivation for exploring approximate indexing techniques when working with growing data volumes.

5.3 Inner Product

The experiments were conducted using the IndexFlatIP index from FAISS under varying dataset sizes and vector dimensionalities.

The evaluation focused on:

- Recall performance
- Query latency
- Impact of dataset size
- Impact of vector dimensionality

IndexFlatIP does not expose an *nprobe* search parameter and is therefore evaluated only as an exact inner-product baseline. Its Recall@k is one when evaluated against itself.

Figure 3 compares nearest-neighbor distances across vector dimensions: 128, 256, 512, and 1024. These values are not Recall@k.

Key Observations:

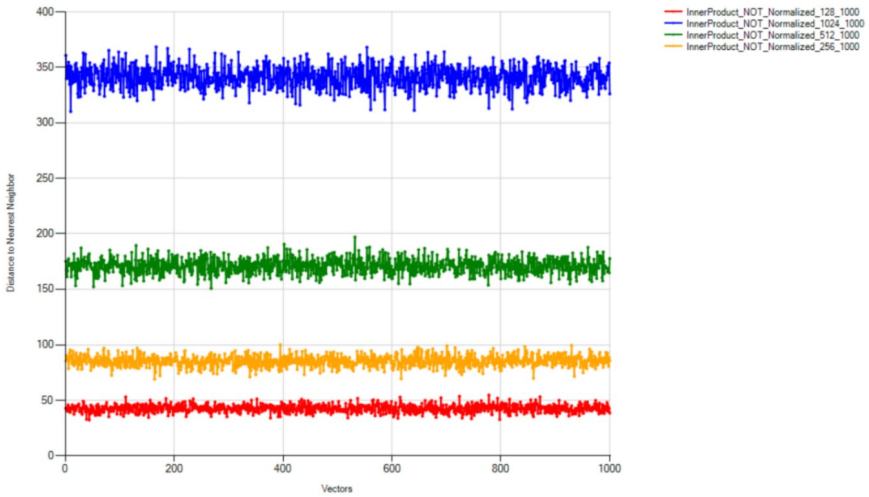


Fig. 3 Nearest-neighbor distances for unnormalized IndexFlatIP vectors across dimensions 128, 256, 512, and 1024; dataset size 1,000.

- Dimensionality Does Not Reduce Recall remains consistently high across all dimensions.
- Exact Search Robustness Recall Because no compression or partitioning is used, increasing dimensionality does not introduce approximation error.

IndexFlatIP maintains retrieval correctness regardless of embedding dimensionality. However, dimensionality is expected to affect computational cost rather than recall. Figure 4 measures query latency for dataset sizes 1,000, 5,000, and 10,000.

Key Observations:

- Linear Growth Pattern: query latency increases approximately linearly with dataset size.
- Scalability Limitation: because every query compares against all vectors, computational complexity is $O(Nd)$, where N is the dataset size and d is the vector dimension.
- Memory Access Cost Becomes Dominant: for larger datasets, memory bandwidth begins to influence performance more significantly. While recall is optimal, IndexFlatIP does not scale efficiently for large datasets due to linear search complexity.

Figure 5 evaluates query latency for dimensions 128, 256, 512, and 1024 at dataset size 10,000.

Key Observations:

- Latency Increases with Dimensionality Higher dimensional vectors require more multiplications per comparison.
- Computation Dominates Cost Time complexity grows proportionally with dimension

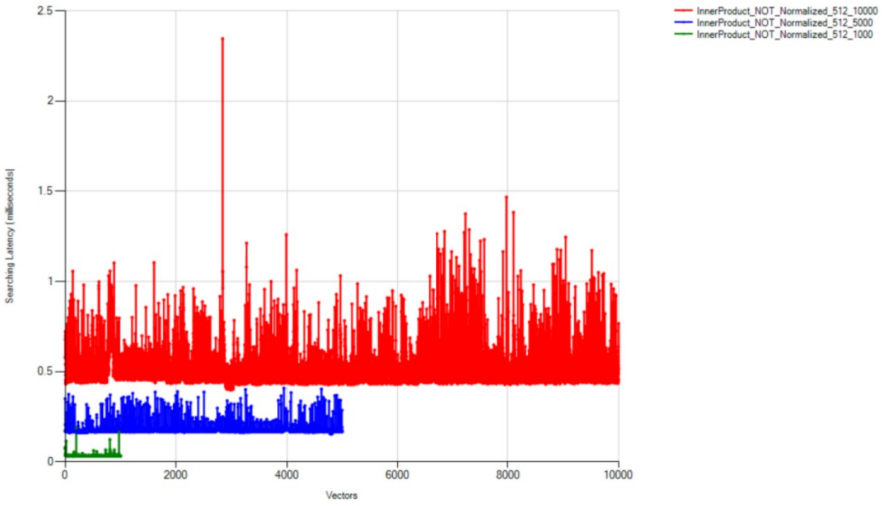


Fig. 4 Query Latency Comparison of different size of not normalized vectors for IndexFlatIP, dimension: 128, dataset sizes: 1000; 5,000; 10,000

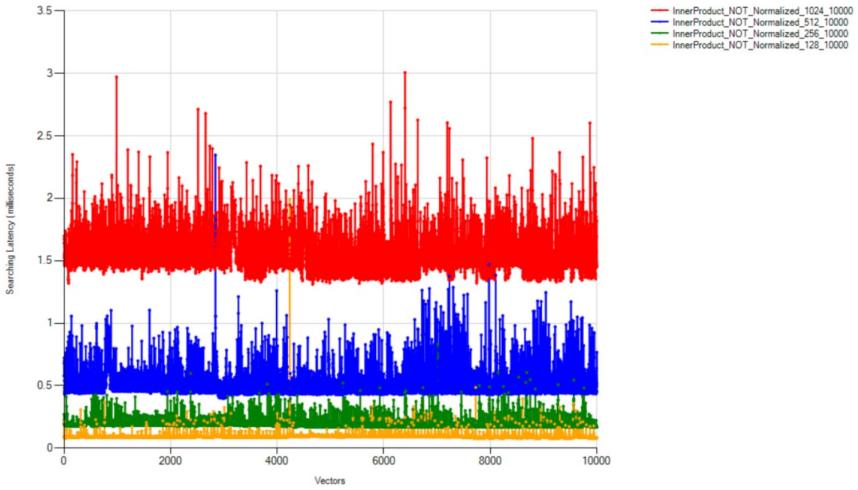


Fig. 5 Query Latency Comparison of different dimensions of not normalized vectors for IndexFlatIP, dimensions: 128,256,512,1024, dataset size: 10,000

- No Accuracy Trade-Off Unlike compressed indices (e.g., PQ), increased dimensionality does not reduce recall.

IndexFlatIP exhibits predictable computational growth with dimensionality, reinforcing its role as a computational baseline rather than a scalable production solution.

5.4 IVFFlat index

To evaluate the effectiveness of FAISS IndexIVFFlat, we conducted a series of experiments analyzing training behavior, recall performance, query latency, and scalability. Unlike exhaustive search methods, IVFFlat introduces a clustering-based structure that partitions vectors into inverted lists, reducing search space during queries. The IVFFlat evaluation reports Recall@k against the exact flat-index ground truth. Distance and latency measurements are reported separately from recall.

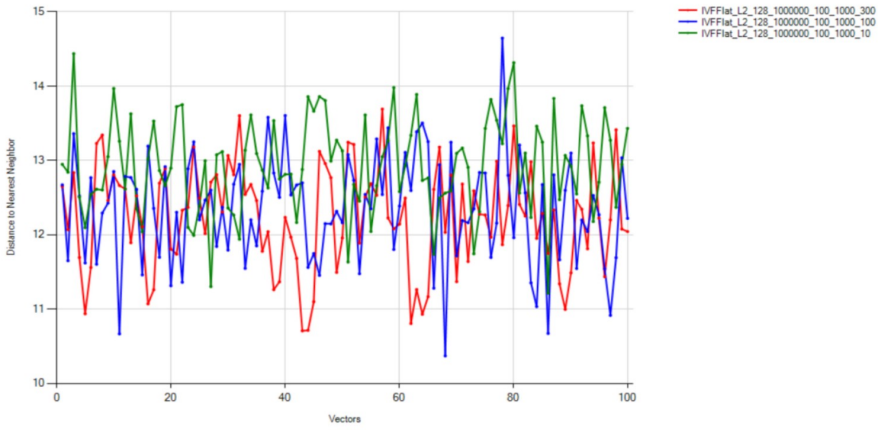


Fig. 6 Nearest-neighbor distances for different *nprobe* values for IVFFlat. Distance is reported separately from Recall@k.

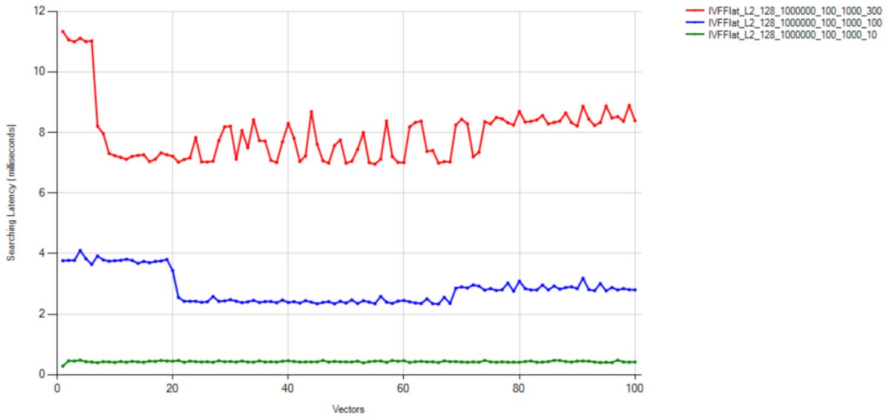


Fig. 7 Queary Latency for different nprobe values for IVFFlat index, Dimension: 128, Dataset size 1mln., Query vectors 100, clusters 1000, probes: 10,100,300

The results clearly show:

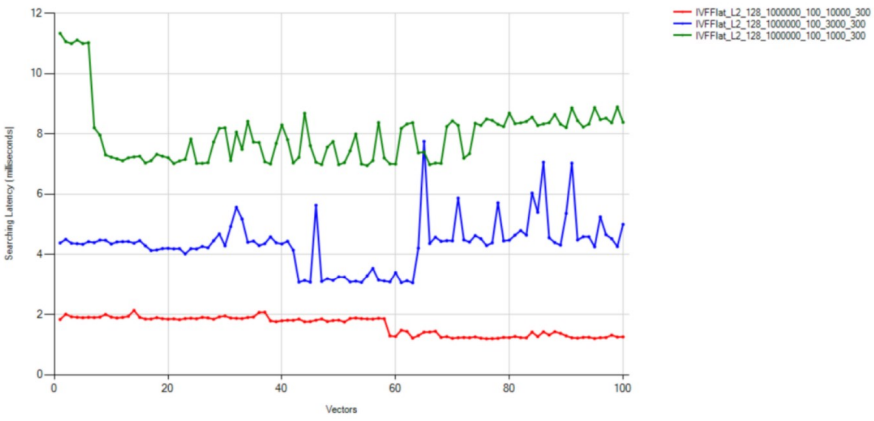


Fig. 8 Queary Latency for different number of clusters for IVFFlat index, Dimension: 128, Dataset size 1mln., Query vectors 100, clusters 1000;3,000;10,000; probe: 300

- Lower nprobe values reduce search time but may miss true nearest neighbors.
- Increasing nprobe improves recall.
- Recall approaches 100% when probing more clusters. This confirms the fundamental trade-off in IVFFlat: Higher $nprobe \rightarrow$ higher recall, higher latency.

Key conclusions:

- IVFFlat introduces controllable approximation.
- Recall is tunable through nprobe.
- Full recall is achievable but at increased cost.

Nearest-neighbor distances across different dataset sizes are presented in Figure 9; latency is evaluated separately in Figure 10.

The distance plot does not measure latency. The complexity analysis predicts that IVFFlat avoids scanning all vectors and instead searches within selected clusters; the corresponding latency comparison is reported separately.

The results indicate:

- The matched latency benchmark quantifies the cost of searching selected clusters.
- The effect of fixed *nprobe* is evaluated separately from nearest-neighbor distance.
- The system scales efficiently with increasing dataset size.

Performance comparison across embedding dimensions is illustrated in Figure 10.

The results show:

- Higher-dimensional embeddings require more computation.
- 512 dimension vectors produce consistently higher query times than 128 dimension.

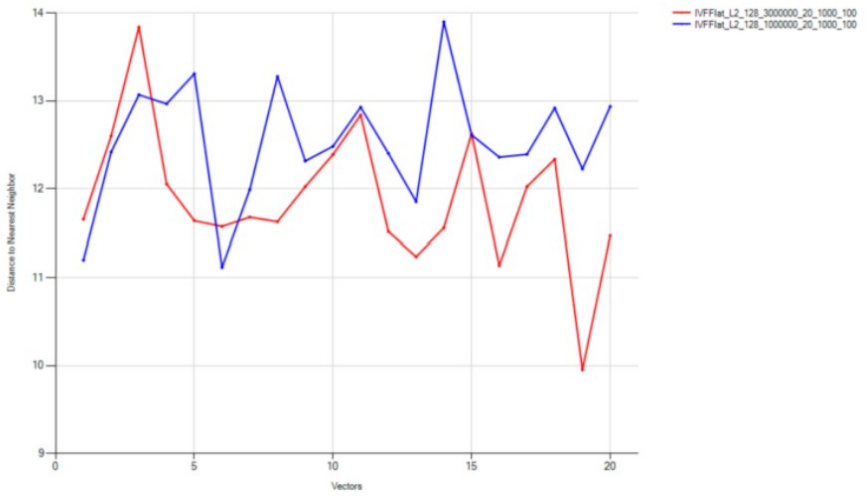


Fig. 9 Nearest-neighbor distances for IVFFlat at dataset sizes 1 and 3 million, dimension 128, 20 queries, 1,000 clusters, and $nprobe = 100$.

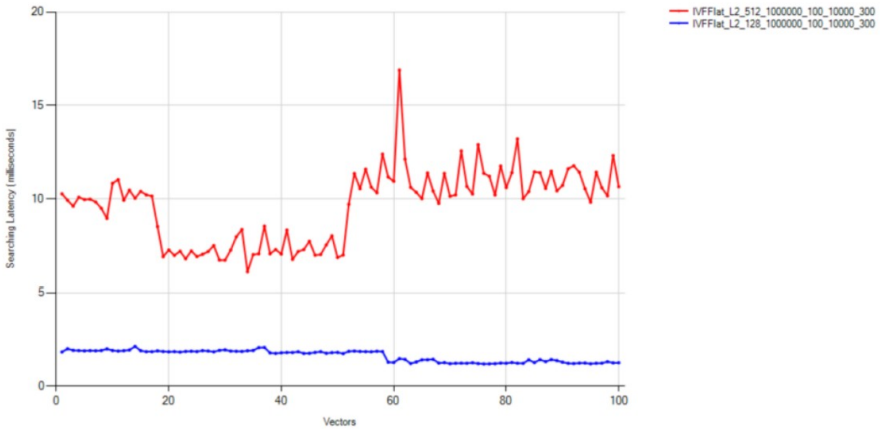


Fig. 10 Query Latency for different dataset size for IVFFlat index, Dimension: 128,512, Dataset size 1mln., Query vectors 100, clusters 10,000; probe: 300

- The dimensionality effect becomes more noticeable as nprobe increases.

However, because IVFFlat reduces candidate vectors through clustering, the dimensionality impact is less severe compared to exhaustive search.

Huge range of analysis reveals that:

- IVFFlat significantly reduces query latency compared to exhaustive search.
- Recall can be tuned through nprobe.

- Memory usage remains linear with slight centroid overhead.
- The method offers a practical balance between speed and accuracy.

The experimental results demonstrate that IndexIVFFlat effectively balances computational efficiency and retrieval accuracy. While it does not inherently guarantee perfect recall at low nprobe values, it provides a flexible mechanism to approach exact search accuracy when needed.

Compared to exhaustive search, IVFFlat offers:

- Improved scalability
- Adjustable recall-performance trade-off.
- Stable and predictable behavior across dataset sizes.

As dataset size increases, the advantage of clustering-based partitioning becomes increasingly significant. Therefore, IVFFlat is well suited for medium- to large-scale similarity search applications where brute-force methods become computationally prohibitive.

5.5 IVFSQ Index

The experimental evaluation of IVFSQ was conducted using one million vectors with 1,000 query vectors under controlled indexing parameters. Unless otherwise stated, the number of clusters was fixed at 100 and 10 clusters were probed during search. The analysis focuses on three main aspects: the effect of scalar quantizer type, the impact of dimensionality, and comparison with alternative index structures.

Figure 11 presents nearest-neighbor distances for different scalar quantizer configurations. These distance values are diagnostics and do not measure Recall@k; retrieval accuracy is reported separately in the matched ground-truth benchmark. Figure 12

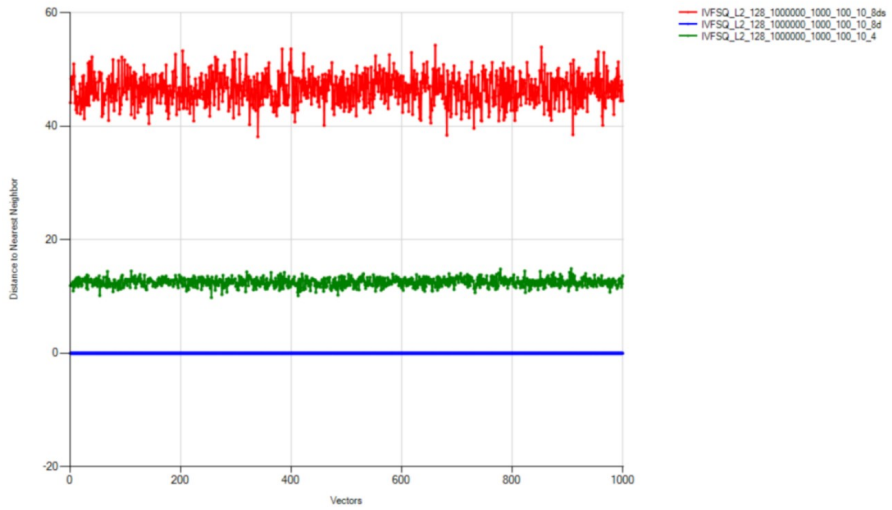


Fig. 11 Nearest-neighbor distances for IVFSQ with scalar quantizers 4, 8d, and 8ds; dimension 128, one million vectors, 1,000 queries, 100 clusters, and $nprobe = 10$.

examines nearest-neighbor distances at dimensions 128, 256, and 512 with a fixed 4-bit scalar quantizer. The figure is descriptive and does not establish Recall@k or retrieval accuracy.

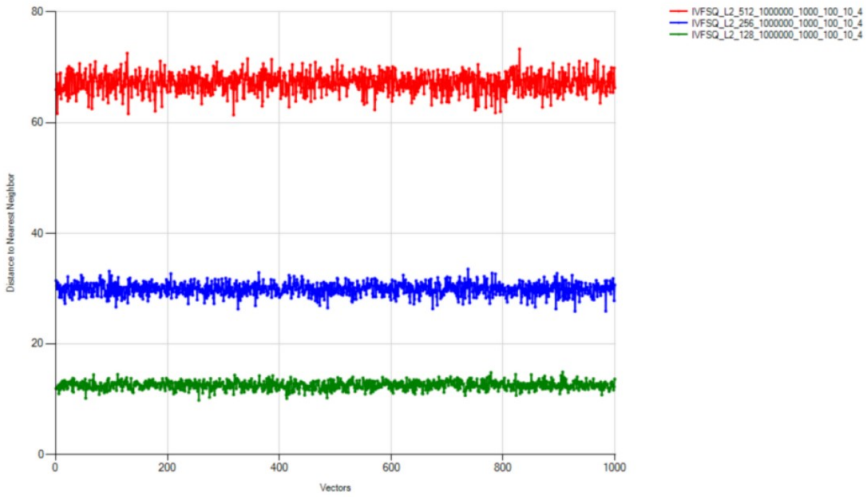


Fig. 12 Nearest-neighbor distances for IVFSQ at dimensions 128, 256, and 512 with a 4-bit scalar quantizer; one million vectors, 1,000 queries, 100 clusters, and $nprobe = 10$.

Figure 13 compares nearest-neighbor distances for IVFSQ, IVFFlat, and IVFPQ under the displayed settings. It is not a Recall@k comparison; accuracy and memory trade-offs are quantified in Table 5.

The distance diagnostics alone do not establish an accuracy–memory trade-off. The measured IVFSQ size and Recall@k values in Table 5 provide the controlled evidence for that comparison.

5.6 HNSW Index

The behavior of the HNSW index was analyzed using one million 128-dimensional vectors with 1,000 queries. The experiments focus on the influence of efSearch and M-connectivity parameters on recall and latency. Figure 14 reports nearest-neighbor distances for the HNSW parameter sweep. Recall@k is evaluated separately using returned identifiers and exact-search ground truth.

Figure 15 analyzes latency as a function of $efSearch$ (16, 32, 64, 128). The results show a clear upward trend: latency increases progressively as $efSearch$ grows. This behavior is expected because a larger $efSearch$ requires visiting more nodes and performing additional distance computations. While higher $efSearch$ enhances recall, it directly impacts query response time. The latency growth appears gradual rather than abrupt, suggesting predictable performance scaling when adjusting $efSearch$.

Figure 16 evaluates latency against different M-connectivity values with $efSearch = 16$. Increasing M-connectivity results in higher latency. A denser graph

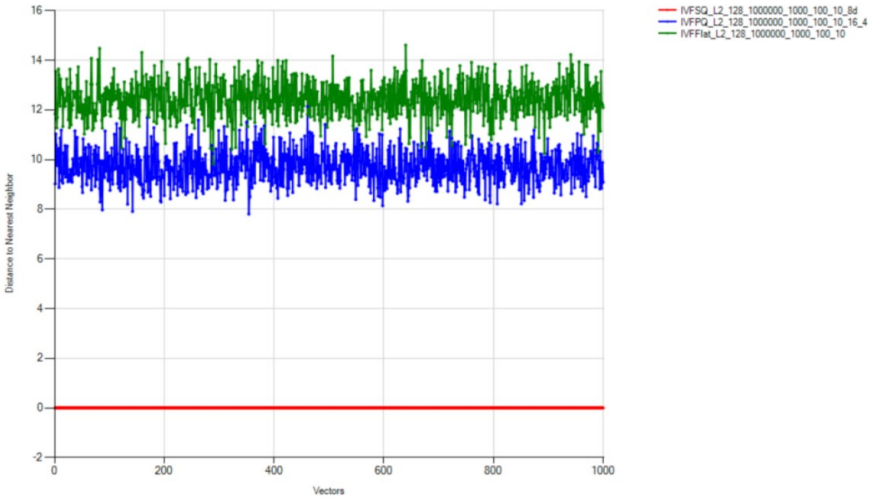


Fig. 13 Nearest-neighbor distances for IVFFlat, IVFPQ, and IVFSQ; dimension 128, one million vectors, 1,000 queries, 100 clusters, and $nprobe = 10$.

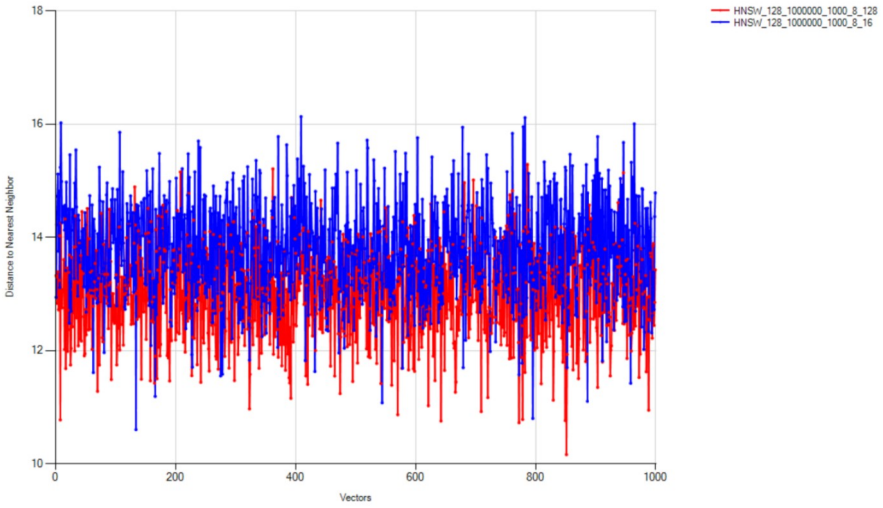


Fig. 14 Nearest-neighbor distances for different $efSearch$ values for HNSW. Distance is reported separately from Recall@k.

structure increases the number of neighbor connections maintained per node, which leads to additional computational overhead during search.

The combined observations indicate that efSearch strongly impacts recall, while both efSearch and M-connectivity influence latency. High efSearch values approximate the accuracy of more exhaustive search methods but at the expense of longer query time. Conversely, lower efSearch settings reduce latency but compromise recall.

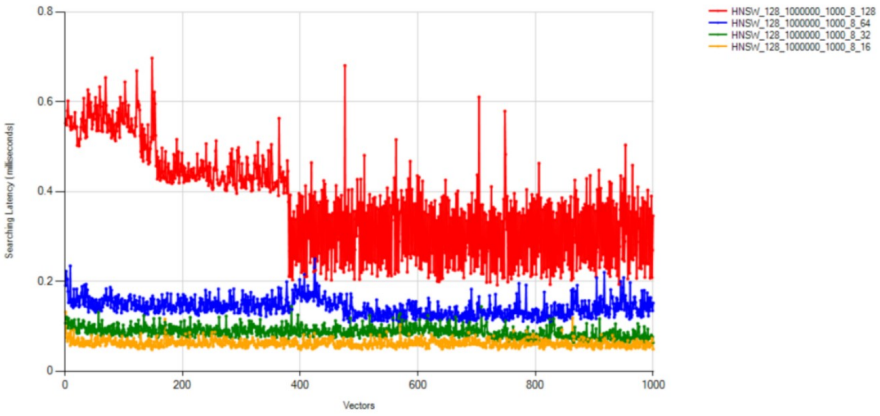


Fig. 15 Latency vs efSearch, Dimension is 128, vectors 1mln, Queried vectors 1000, M-Connectivity 8, efSearch 16,32,64,128

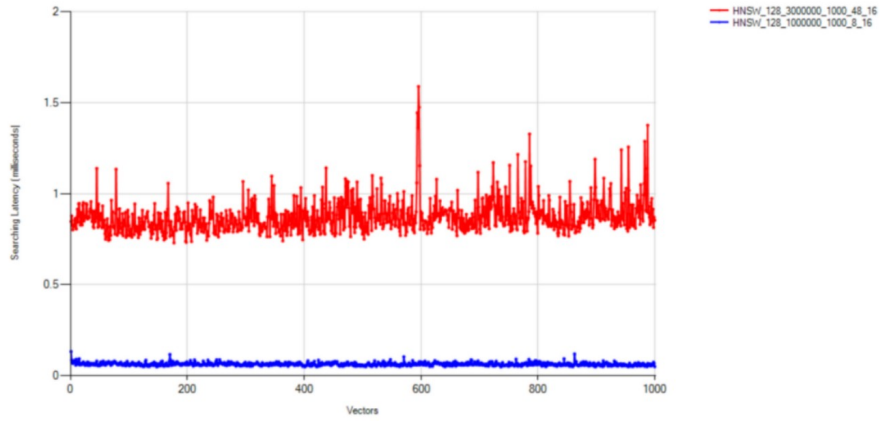


Fig. 16 HNSW latency for connectivity settings $M = 8$ and $M = 48$ with $efSearch = 16$.

Similarly, increasing M-connectivity enhances structural richness of the graph yet contributes to higher computational expense.

In summary, HNSW demonstrates a controllable trade-off between accuracy and efficiency. The experimental results confirm that parameter tuning plays a decisive role in achieving the desired operational balance. For large-scale vector collections, selecting appropriate efSearch and M values allows practitioners to align the index behavior with application-specific requirements, whether prioritizing recall or minimizing response time.

6 Comparative Evaluation Of Vector Indexing Methods

Table 6 reports measured values from the matched benchmark: 100,000 database vectors, 1,000 queries, dimension 128, $k \in \{1, 10, 100\}$, $nlist = 100$, $nprobe = 10$, $M = 16$, and ten repetitions. Times and memory values are medians across repetitions; resident set size is process-level RSS and should not be interpreted as index-only memory. Recall is computed against the exact flat ground truth.

Table 5 Comparative Analysis

Index	Recall@1	Recall@10	Recall@100	Query ms	Index MB	Build s
Flat	1.000	1.000	1.000	0.167	48.828	0.015
IVFFlat	0.358	0.317	0.273	0.054	49.641	0.090
IVFPQ	0.072	0.102	0.134	0.020	2.464	0.976
IVFSQ	0.108	0.164	0.195	0.027	13.020	0.119
HNSW	0.514	0.453	0.333	0.011	62.593	2.379

Across the ten repetitions, mean query latency was 0.167 ± 0.003 ms/query for Flat, 0.054 ± 0.003 for IVFFlat, 0.020 ± 0.001 for IVFPQ, 0.027 ± 0.004 for IVFSQ, and 0.011 ± 0.001 for HNSW, where the second term is the sample standard deviation. Approximate 95% confidence-interval half-widths for these means were 0.002, 0.002, 0.001, 0.002, and 0.001 ms/query, respectively. Recall@10 was identical across repetitions for Flat (1.000), IVFFlat (0.317), IVFPQ (0.102), and IVFSQ (0.164); HNSW ranged from 0.446 to 0.456. These repeated measurements quantify run-to-run variation but do not constitute a claim of statistical significance across hardware or datasets.

Table 6 Real-text embedding validation on 20 Newsgroups documents encoded with `all-MiniLM-L6-v2`.

Index	Recall@1	Recall@5	Recall@10	Query ms	Index MB
Flat	1.000	1.000	1.000	0.0116	4.395
IVFFlat	0.973	0.961	0.963	0.0183	4.465
IVFPQ	0.567	0.633	0.645	0.0079	0.537
IVFSQ	0.970	0.957	0.959	0.0078	1.172
HNSW	1.000	0.997	0.996	0.0109	4.806

Figure 17 shows Recall@k curves and the measured latency–recall and memory–recall frontiers for the matched benchmark. Error bars show the half-range across the ten repetitions.

Under this configuration, HNSW and IVFPQ have the lowest median query times, but HNSW uses the largest serialized index and IVFPQ has substantially lower recall.

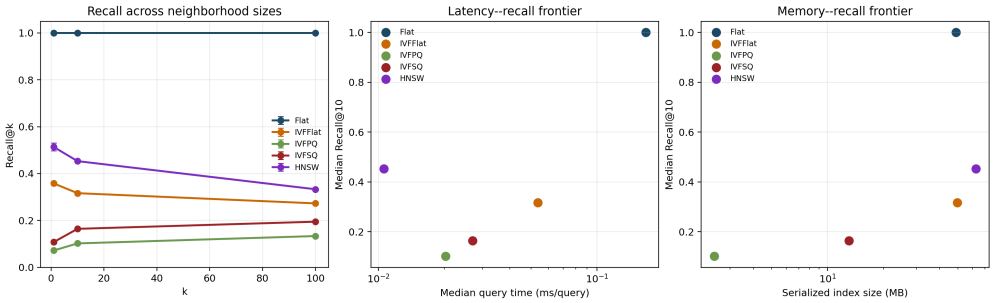


Fig. 17 Recall@k, latency versus Recall@10, and serialized index size versus Recall@10. All methods use the same 100,000-vector database, 1,000 queries, dimension 128, and exact ground truth.

IVFFlat retains more recall than the quantized indexes at the cost of higher latency, while the Flat index provides the exact reference. These measurements describe this configuration only and do not establish a universal ranking across datasets or parameter settings.

IVFPQ provides the smallest serialized index in this benchmark, but its Recall@10 is lower than the other tested approximate indexes. The present experiments do not include billion-scale data, so no billion-scale performance claim is made.

IVFSQ reduces serialized size relative to Flat and HNSW, with intermediate latency and recall in this matched setting.

HNSW provides the lowest median query time in this run and higher recall than the two quantized indexes, but its Recall@100 remains below the exact baseline. Index selection should therefore be based on a measured latency–recall–memory frontier for the target workload rather than on a universal ranking.

7 Practical Deployment Considerations

The selection of an indexing strategy largely depends on the operational requirements and data characteristics of the target application.

The benchmark does not use banking, healthcare, or production RAG data, so it does not establish domain-specific superiority. In practice, those applications require a workload-specific evaluation using the target embedding distribution and recall threshold.

For semantic search, cosine similarity may be appropriate when the embedding model is trained for angular similarity. This is a metric-selection hypothesis to be tested with real embeddings, not a conclusion established by the current synthetic-vector experiments. Citation recommendation is another example of embedding-based retrieval over scientific content, including graph-enhanced language-model representations [11].

For language-model embeddings, IVFPQ can reduce the serialized index size, as shown in Table 6, but the acceptable recall loss depends on the application. The present study does not benchmark production-scale or billion-vector collections.

8 Conclusion

This work provides a controlled comparison of FAISS indexing strategies on synthetic vectors. Results show that no single index is universally optimal; selection depends on dataset size, memory constraints, metric, and recall requirements.

In the matched experiment, HNSW achieved low query latency and higher recall than the quantized indexes, while IVFPQ achieved the smallest serialized index. These observations are configuration-specific and should be validated on real embeddings before being generalized. Normalization makes inner product retrieval equivalent to cosine similarity for normalized vectors.

The main practical contribution is a reproducible evaluation protocol that reports Recall@k, latency, construction time, index size, and process memory under a common configuration. Task-level validation using human relevance judgments remains future work.

References

[1] Johnson, J., Douze, M., Jegou, H.: Billion-scale similarity search with gpus. IEEE Transactions on Big Data **7**(3), 535–547 (2021) https://doi.org/10.1109/TBDATA.2019.2921572

[2] Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., Yih, W.-t.: Dense passage retrieval for open-domain question answering. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, pp. 6769–6781 (2020). https://doi.org/10.18653/v1/2020.emnlp-main.550

[3] Khattab, O., Zaharia, M.: Colbert: Efficient and effective passage search via contextualized late interaction over bert. In: Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 39–48 (2020). https://doi.org/10.1145/3397271.3401075

[4] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., K"uttler, H., Lewis, M., Yih, W.-t., Rockt"aschel, T., Riedel, S., Kiela, D.: Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in Neural Information Processing Systems **33**, 9459–9474 (2020)

[5] Muennighoff, N., Tazi, N., Magne, L., Reimers, N.: Mteb: Massive text embedding benchmark. In: Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, pp. 2014–2037 (2023). https://doi.org/10.18653/v1/2023.eacl-main.148

[6] Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval, p. 482. Cambridge University Press, Cambridge, UK (2008). https://doi.org/10.1017/CBO9780511809071 . https://doi.org/10.1017/CBO9780511809071

[7] Jégou, H., Douze, M., Schmid, C.: Product quantization for nearest neighbor

search. IEEE Transactions on Pattern Analysis and Machine Intelligence **33**(1), 117–128 (2011) https://doi.org/10.1109/TPAMI.2010.57

[8] Gersho, A., Gray, R.M.: Vector Quantization and Signal Compression, p. 732. Springer, New York, NY, USA (1992). https://doi.org/10.1007/978-1-4615-3626-0

[9] Malkov, Y.A., Yashunin, D.A.: Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence **42**, 824–836 (2018) https://doi.org/10.1109/TPAMI.2018.2889473

[10] Reimers, N., Gurevych, I.: 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, pp. 3982–3992 (2019). https://doi.org/10.18653/v1/D19-1410

[11] Ali, Z., Qi, G., Ullah, I., Mohammed, A.A.Q., Kefalas, P., Muhammad, K.: Glamor: Graph-based language model embedding for citation recommendation. In: Proceedings of the 18th ACM Conference on Recommender Systems, pp. 929–933 (2024). https://doi.org/10.1145/3640457.3688171