HAKES: Scalable Vector Database for Embedding Search Service

Guoyu Hu
National University of Singapore
guoyu.hu@u.nus.edu

Shaofeng Cai
National University of Singapore
shaofeng@comp.nus.edu.sg

Tien Tuan Anh Dinh
Deakin University
anh.dinh@deakin.edu.au

Zhongle Xie*
Zhejiang University
xiezl@zju.edu.cn

Cong Yue
National University of Singapore
yuecong@comp.nus.edu.sg

Gang Chen
Zhejiang University
cg@zju.edu.cn

Beng Chin Ooi
Zhejiang University
National University of Singapore
ooibc@comp.nus.edu.sg

ABSTRACT

Modern deep learning models capture the semantics of complex data by transforming them into high-dimensional embedding vectors. Emerging applications, such as retrieval-augmented generation, use approximate nearest neighbor (ANN) search in the embedding vector space to find similar data. Existing vector databases provide indexes for efficient ANN searches, with graph-based indexes being the most popular due to their low latency and high recall in real-world high-dimensional datasets. However, these indexes are costly to build, suffer from significant contention under concurrent read-write workloads, and scale poorly to multiple servers.

Our goal is to build a vector database that achieves high throughput and high recall under concurrent read-write workloads. To this end, we first propose an ANN index with an explicit two-stage design combining a fast filter stage with highly compressed vectors and a refine stage to ensure recall, and we devise a novel lightweight machine learning technique to fine-tune the index parameters. We introduce an early termination check to dynamically adapt the search process for each query. Next, we add support for writes while maintaining search performance by decoupling the management of the learned parameters. Finally, we design HAKES, a distributed vector database that serves the new index in a disaggregated architecture. We evaluate our index and system against 12 state-of-the-art indexes and three distributed vector databases, using high-dimensional embedding datasets generated by deep learning models. The experimental results show that our index outperforms index baselines in the high recall region and under concurrent read-write workloads. Furthermore, *HAKES* is scalable and achieves up to 16× higher throughputs than the baselines.

PVLDB Reference Format:
Guoyu Hu, Shaofeng Cai, Tien Tuan Anh Dinh, Zhongle Xie, Cong Yue, Gang Chen, and Beng Chin Ooi. *HAKES*: Scalable Vector Database for Embedding Search Service. PVLDB, 18(9): 3049 - 3062, 2025.
doi:10.14778/3746405.3746427

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746427

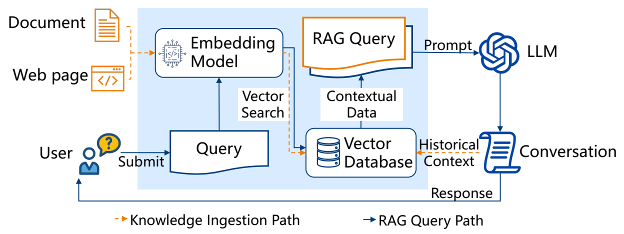


Figure 1: Vector database in retrieval augmented generation.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/nusdbsystem/HAKES-Search.

1 INTRODUCTION

High-dimensional embedding vectors generated by deep learning models are becoming an important form of data representation for complex, unstructured data such as images [40, 46], audios [7], and texts [32, 52]. The models convert input data to vectors in an embedding space and capture the data semantics relevance by their relative positions in the high-dimensional space. Typical embedding vectors nowadays have hundreds to thousands of dimensions.

Vector databases are designed to support efficient nearest neighbor search in the vector space. They underlie many modern applications, ranging from search engines [9, 40] , recommendation systems [43] to retrieval-augmented generation (RAG) [35]. These applications require efficient, high quality search as well as support for database updates. Figure 1 shows an example of using a vector database in RAG applications. RAG turns a user-submitted query into a vector, performs nearest neighbor search to find similar data stored in the vector database, augments the query with the data found, and then sends the augmented query to a large language model (LLM). RAG frequently updates the vector database with new knowledge, such as new documents, web pages, and past user interactions, enabling timely integration of relevant context, compared to offline training or fine-tuning LLMs with new information [39].

The design of vector databases centers on indexes optimized for efficient nearest neighbor search, distinct them from other general-purpose databases for deep learning [44]. Since searching for exact nearest neighbors is too costly due to the curse of dimensionality [27], existing works on vector indexes focus on approximate nearest neighbor (ANN) search. The ANN indexes can be classified as graph-based or partitioning-based indexes [1, 10, 17, 22, 38, 42]. They are mostly evaluated using read-only workloads. Many of the datasets used for evaluation, such as Deep, Sift, and Glove, have lower dimensions than the deep embedding vectors used in emerging applications [3, 22, 34, 42]. We identify three limitations of vector databases built around the existing ANN indexes to support modern applications, for which graph-based indexes are the recommended choice.

The first limitation is the computation overhead under high-dimensional spaces. In particular, comparing a vector against its neighbors becomes more expensive with higher dimensions. Graph-based indexes [15, 29, 38, 42] are very costly to build because they require connecting each data point to its near neighbors and optimizing the graph structure to enable efficient traversal. The second limitation is the search performance under concurrent read-write workloads. Updating an existing index can be done in-place [42, 57, 64], or out-of-place using a separate data structure and performing periodic consolidation [13, 51, 55]. Graph indexes perform in-place updates, and require fine-grained locking over the neighborhood of the nodes on its traversal path [42, 51]. This results in significant read-write contention. Out-of-place updates, on the other hand, require a separate search on the newly inserted data, while only postponing the update cost to a later time. The third limitation is scalability. Existing vector databases treat their indexes as black boxes [9, 21, 55]. They shard the data and build an independent graph index for each shard. However, to achieve high recall in high-dimensional spaces, nearly all data shards are searched. The large number of searches per query leads to low throughput.

We present *HAKES*, a scalable vector database that achieves high recall and throughput under concurrent read-write workloads. The database adopts a filter-and-refine design that consists of two stages. The filter stage narrows down the search candidates using compressed vectors for efficiency. The refine stage ranks the candidates based on the full-precision vectors. The system addresses the first limitation by employing dimensionality reduction, coarse-grained partitioning, and quantization techniques. Furthermore, it proposes a novel light-weight machine learning technique to optimize the index parameters such that the filter stage is efficient and returns a set of high-quality candidates. *HAKES* also includes an early termination check at the filter stage to avoid unnecessary processing. The compressed vectors are grouped by IVF index in contiguous buffers, and decoupling the index parameters used for compressing the vectors and those used during search enables seamless integration of new vectors at minimal overhead and contention, addressing the second limitation. *HAKES* addresses the scalability limitation by exploiting the decoupling of the filter and refine stage to deploy them in a disaggregated architecture. It distributes the memory and computation cost over multiple nodes, thereby achieving high throughput at scale.

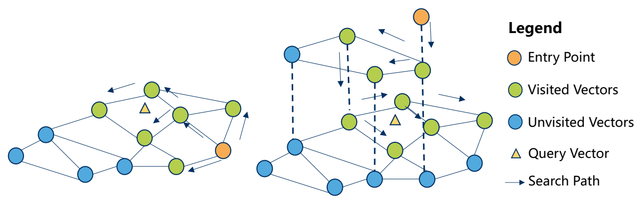
HAKES combines and adapts known techniques in a novel way to achieve its goal. In particular, existing works on quantization aim to improve the quality of similarity score approximation over the compressed vectors, minimizing the need for reranking the full-precision vectors [1, 18, 20, 22, 48]. *HAKES* aims to achieve good throughput-recall tradeoffs overall. By having a separate refine stage that reranks the original vectors, the dimensionality reduction and quantization aim to compress the vectors aggressively to reduce the computation cost at the filter stage. The compression parameters are learned in an end-to-end manner, in which the objective is to minimize the similarity score distribution distortion locally for vectors close to each other. The learning approach in *HAKES* does not assume access to external information, such as the embedding generation models, ground truth neighbors, or semantic labels, which is a different problem setting compared to other works that employ learning to improve the retrieval quality [56, 58]. Moreover, our system allows for applying the newly learned parameters during search directly without re-indexing vectors in the database. In other words, learning can be done asynchronously while the vector database serves queries. Finally, the early termination check in *HAKES-Index* is more lightweight than that in [36, 60], and more effective in our context than those in [28, 61, 63], since it does not rely on accurate similarity scores under compression.

In summary, we make the following contributions:

- We propose a novel index, *HAKES-Index*, that combines a compressed partitioning-based index with dimensionality reduction and quantization. The index leverages a lightweight machine learning technique to generate high-quality candidate vectors, which are then refined by exact similarity computation. It allows terminating the search early based on the intermediate results.
- We propose a technique that decouples index parameters for compressing vectors during updates from those used for similarity computation. This ensures high performance under concurrent read-write workloads.
- We design a distributed vector database, called *HAKES*, employing the new index in a disaggregated architecture. The system achieves scalability by spreading out the memory and computation overhead over multiple nodes.
- We compare *HAKES-Index* and *HAKES* against 12 state-of-the-art indexes and three popular commercial distributed vector databases. We evaluate the indexes and systems using high-dimensional embedding vector datasets generated by deep learning models. The results demonstrate that *HAKES-Index* outperforms both partitioning-based and graph-based index baselines. Furthermore, *HAKES* is scalable, and achieves up to 16× higher throughputs at high recall than the three other baselines do.

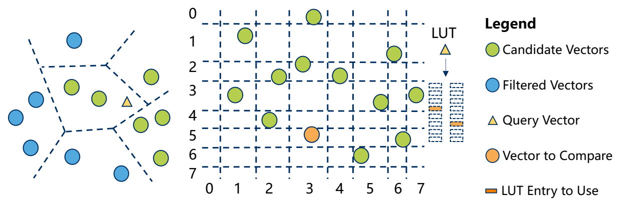
2 PRELIMINARIES

Approximate nearest neighbor Search. Let $\mathcal{D}$ denote a dataset containing N vectors in a d-dimensional vector space $\mathbb{R}^d$. For a query vector $\mathbf{x}$, the similarity between $\mathbf{x}$ and a vector $\mathbf{v} \in D$ is defined by a metric $d(\mathbf{x}, \mathbf{v})$. Common metrics include the Euclidean distance, inner product, and cosine similarity. A vector $\mathbf{v_i}$ is considered closer to $\mathbf{x}$ than $\mathbf{v_j}$ if $d(\mathbf{x}, \mathbf{v_i}) < d(\mathbf{x}, \mathbf{v_j})$. The k nearest neighbors of $\mathbf{x}$ are vectors in $\mathcal{R} \subseteq \mathcal{D}$, where $|\mathcal{R}| = k$ and $\forall \mathbf{v} \in \mathcal{R}, \forall \mathbf{u} \in \mathcal{D} \backslash \mathcal{R}, d(\mathbf{x}, \mathbf{v}) \leq d(\mathbf{x}, \mathbf{u})$. Finding the exact set $\mathcal{R}$ in a high-dimensional space is expensive due to the curse of dimensionality [27]. Instead, existing works on vector databases focus on approximate nearest neighbor (ANN) search, which use ANN indexes to quickly find a set $\mathcal{R}'$ of vectors that are close to, but not necessarily nearest to x. The quality of $\mathcal{R}'$ is measured by its *recall* relative to the exact nearest neighbor set, computed as $\frac{|\mathcal{R} \cap \mathcal{R}'|}{|\mathcal{R}|}$. We discuss two major classes of ANN indexes below.



(a) Graph index (b) HNSW

Figure 2: Graph-based ANN index.



(a) Index partitioning (b) PQ

Figure 3: Partitioning-based ANN index.

Graph-based indexes. They build a proximity graph in which the vertices are the vectors, and an edge between two vertices means the two corresponding vectors are similar [15, 38, 42]. An ANN query involves a greedy beam search that starts from an entry point to locate close neighbors. The query maintains a fixed-size set of candidates and visited nodes during the traversal. At each step, the nearest unvisited vector from the candidate set is selected, and its unvisited neighbors are new potential candidates. These new candidate vectors are evaluated for their similarity scores against the query vector and added to the candidate set accordingly. The process repeats until the candidate set contains only visited nodes, as illustrated in Figure 2a. When building or adding new vectors to the graph, a similar search is conducted to find the nodes to be connected based on a condition that allows future queries to reach their nearest neighbors and in a small number of steps [14, 29, 42]. Since the search efficiency and recall depend on the graph, most existing works on graph indexes focus on building and maintaining a high-quality graph [14, 15, 42, 64].

The Hierarchical Navigable Small World graph (HNSW) is the most popular graph index. It supports incremental updates and efficient search by introducing a hierarchical structure with an exponentially decreasing number of vertices from the bottom to the top level, as shown in Figure 2b. A search starts from an entry point at the top level. At each level, it finds the nearest neighbor and starts the search in the next level with that vertex. Finally, at the bottom level, it performs beam search to find nearest neighbors. During an update (i.e. adding a new vector), the new vertex's neighbors are first located at each level, and then the edges are updated. The update condition restricts the number of neighbors and only adds an edge if the similarity between the searched candidate and the new vector is larger than that of the new vector and its existing neighbors. This update process is costly, and it creates significant contention under concurrent read-write workloads.

Partitioning-based indexes. They divide vectors into multiple partitions using one or multiple hashing schemes, such that similar vectors are in the same partition. The similarity of a query vector to all the vectors in a partition can be approximated by its proximity to the partition itself. The partition assignments can be encoded for efficient search. Examples of hashing schemes include locality sensitive hashing (LSH) [16, 34, 45], clustering [10, 28], quantization [19, 30, 31], or neural networks [8, 23, 37]. New vectors are added to the corresponding partition by computing its partition assignment. A search for vector $\mathbf{x}$ starts by identifying close partitions, then retrieving the vectors belonging to the selected partitions, as shown in Figure 3a. Finally, the k closest vectors are selected by evaluating the similarity scores.

Inverted-file (IVF) and product quantization are the most popular partitioning-based indexes. IVF [10, 30] uses k-means and a sample set of vectors to determine the cluster centroids, and then vectors having the same closest centroids are stored together in respective buckets. During a search, all partitions are ranked based on the similarity between their centroids and the query vector $\mathbf{x}$. The top *nprobe* partitions are scanned to produce k nearest neighbors. The number of centroids N_c for k-means and the *nprobe* determine the cost of ranking partitions and the number of candidate vectors. These parameters also affect recalls. For example, million-scale datasets typically require N_c in 1000s and *nprobe* in 10s to 100s to achieve high recall.

Product quantization (PQ) splits the original d-dimensional space into m orthogonal subspaces of the same dimension $d' = d/m$. Each subspace is further partitioned, e.g., using k-means with N_c centroids, resulting in $(N_c)^m$ partitions. A codebook $\mathbf{C^{PQ}} \in \mathbb{R}^{N_c \times d}$ is the concatenation of subspace centroids $\mathbf{C^{PQ}}_j \in \mathbb{R}^{N_c \times d'}$, i.e., $\mathbf{C^{PQ}} = [\mathbf{C^{PQ}}_1, \mathbf{C^{PQ}}_2, ..., \mathbf{C^{PQ}}_m]$. A vector can be quantized into a concatenation of indexes of the centroids in the codebook at each subspace, $p(v) = [p_1(\mathbf{v}), p_2(\mathbf{v}), ..., p_m(\mathbf{v})]$, where $p_j(\mathbf{v}) = \arg\min_i ||\mathbf{C^{PQ}}_j[i] - \mathbf{v}_j||$ denotes the index of the closest centroid in the j^{th} subspace centroids $\mathbf{C^{PQ}_j}$. Let $q_j(\mathbf{v}) = \mathbf{C^{PQ}}_j[p_j(\mathbf{v})]$ be the closest centroid of $\mathbf{v}$. The concatenation of centroids closest to $\mathbf{v}$ in respective subspaces forms its approximation: $\mathbf{v} \approx q(\mathbf{v}) = [q_1(\mathbf{v}), q_2(\mathbf{v}), ..., q_m(\mathbf{v})]$. Then, the similarity between a vector $\mathbf{x}$ and a vector $\mathbf{v}$ can be approximated as $d(\mathbf{x}, q(\mathbf{v}))$. For the commonly used Euclidean distance (normally without taking the square root) and inner product, we have:

$$d(\mathbf{x}, \mathbf{v}) \approx d(\mathbf{x}, q(\mathbf{v})) = \sum_{j=1..m} d(\mathbf{x}_j, q_j(\mathbf{v})). \quad (1)$$

PQ enables efficient comparison of $\mathbf{x}$ against the candidate vectors. A query vector $\mathbf{x}$ is split into m subvectors, each of which is compared against all the centroids in its corresponding subspace, $\mathbf{C^{PQ}}_j$. The resulting similarity scores are stored in a lookup table, $\text{LUT} \in \mathbb{R}^{N_c \times m}$. Given Equation 1, the similarity between $\mathbf{x}$ and any vector $\mathbf{v}$ can be approximated using the quantized vector $q(\mathbf{v})$ via m lookups into the LUT, followed by a summation, as shown in Figure 3b. In practice, PQ generates compact vector representations. Typically, N_c is 16 and 256, such that only 4 or 8 bits can encode the vector in each subspace. Recent indexes using 4-bit PQ, which yields a LUT small enough to fit in CPU caches, achieved significantly higher throughputs with SIMD [2, 10, 22]. In practice, quantization is used together with other indexes to avoid evaluating all vectors for a query, for example, IVF [30] and graph [1, 29].

Quantization enables efficient but lossy approximation of the similarity scores between vectors. Reranking the candidates can be performed to improve recall, using additional information [1, 31] or the original vectors [10, 18, 22]. Some existing works on quantization, namely [1, 18, 20, 22], focus on reducing approximation errors to minimize reranking. Others aim to transform the vectors to be more suitable for quantization [19, 48], or leverage information about the downstream task and upstream embedding model to improve end-to-end retrieval quality [56, 58].

3 HAKES-INDEX

In this section, we present a novel index, called *HAKES-Index*, that supports efficient search and index update.

3.1 Overview

Figure 4a shows the components in *HAKES-Index*. It consists of three parts, the two respective sets of index parameters to process the vectors for search and insert, the partitions that contain compressed vectors, and the full vectors. Each set of index parameters is composed of a dimensionality reduction module, IVF centroids, and a PQ codebook. The compressed vectors are partitioned by the IVF centroids and the compression involves dimensionality reduction followed by quantization guided by the codebook. The dimensionality reduction module uses a transformation matrix $\mathbf{A} \in \mathbb{R}^{d \times d_r}$ and a bias vector $\mathbf{b} \in \mathbb{R}^{d_r}$ to compress vectors from the original d-dimensional space to that of d_r-dimensional spaces, where $d_r < d$. The IVF centroids, $\mathbf{C^{IVF}}$, determine the partition a new vector is attached to during the insert and rank the partitions for a query vector during the search. The quantization codebooks, $\mathbf{C^{PQ}}$, are used to generate the quantized vector stored in the partitions and compute the Lookup table for search. Note that dimensionality reduction is placed at the front, which speeds up all subsequent computations. We use $\mathbf{A}, \mathbf{b}, \mathbf{C^{IVF}}, \mathbf{C^{PQ}}$ to refer to the insert index parameters and $\mathbf{A'}, \mathbf{b'}, \mathbf{C^{IVF'}}, \mathbf{C^{PQ'}}$ to the search index parameters.

Search query involves four steps, shown in Figure 4b. Step 1 reduces the dimensionality of the query vector from d to d_r with $\mathbf{A'}, \mathbf{b'}$. Next, the output of the dimensionality reduction is used to compute the lookup table (LUT) with the quantization codebook $\mathbf{C^{PQ'}}$ in step 2. Step 3 evaluates the d_r-dimension query vector with $\mathbf{C^{IVF'}}$ to select the closest partitions for scanning using the LUT. $k' > k$ candidates are selected, and step 4 obtains the top k of them by comparing the query vector to their full vectors. The four steps in the search workflow can be mapped into two stages. The filter stage spans steps 1-3, where the majority of vectors are filtered out, leaving k' candidate vectors. The last step is the refine stage, when k' candidates are refined to the top k nearest vectors.

To add vectors to *HAKES-Index*, the insert index parameters are utilized, as shown in Figure 4c. Each new vector is transformed using the dimensionality reduction parameters (step 1) and quantized using the codebook (step 2). It is then appended to both the corresponding partition determined by the IVF centroids and the buffer holding full vectors. For deletion, *HAKES-Index* uses tombstones to mark the deleted vectors. During the filter stage, the tombstones are checked before adding the vectors to the candidate set. The deleted vectors and their corresponding compressed vectors are removed by a compaction step that rewrites the partitions. This step happens when the index is being checkpointed or rebuilt. The latter is triggered by an update in the embedding model, or when the data size grows beyond certain sizes. This approach reduces the interference of deletion on the search and insert operations.

3.2 Index Construction

We construct *HAKES-Index* following the procedure illustrated in Figure 5. Figure 5a shows the first step of building the base index. The insert index parameters are initialized with existing processes and then the dataset is inserted into the index. Particularly, Optimal Product Quantization (OPQ) is employed to initialize $\mathbf{A}$ and $\mathbf{C^{PQ}}$, which iteratively finds a transformation matrix that minimizes the reconstruction error of a PQ codebook, and K-means is employed to initialize the IVF centroids, $\mathbf{C^{IVF}}$. The bias vector $\mathbf{b}$ is zero. Next, the training set is prepared by sampling a set of vectors and obtaining their neighbors with the base index, as in Figure 5b. Note that another set of sampled pairs is used for validation. Then, we use a self-supervised training method to learn the search parameters, $\mathbf{A'}$, $\mathbf{b'}$ and $\mathbf{C^{PQ'}}$, illustrated in Figure 5c, which is the key to *HAKES-Index*'s high performance at high recall, and the technical details will be revealed in the Section 3.3. After training, the IVF centroids $\mathbf{C^{IVF'}}$ are computed by partitioning the sample data with $\mathbf{A}, \mathbf{C^{IVF}}$, then recomputing the centroid for each partition after applying the learned $\mathbf{A'}$ and $\mathbf{b'}$ to vectors in it (Figure 5d). Finally, the newly learned $\mathbf{A'}, \mathbf{b'}, \mathbf{C^{PQ'}}$, and $\mathbf{C^{IVF'}}$, are installed in the index, as shown completed in Figure 5e, serving subsequent search queries.

The training process can run independently from the serving system. In practice, the index is first built and uses $\mathbf{A}, \mathbf{b}, \mathbf{C^{IVF}}, \mathbf{C^{PQ}}$ for both insert and search. As it serves requests, the system records samples, and the training process runs in the background. Once the training is finished, the new parameters $\mathbf{A'}, \mathbf{b'}, \mathbf{C^{IVF'}}, \mathbf{C^{PQ'}}$ can be used immediately to serve queries. In other words, *HAKES-Index* can be updated incrementally. Moreover, the construction of *HAKES-Index* is efficient. That reduces the time to rebuild the index for serving at an updated throughput-recall frontier, when the database sizes and distributions are significantly changed by insertion and deletion.

3.3 Learning Compression Parameters

Since the search recall depends on the quality of the candidate vectors returned by the filter stage, *HAKES-Index* achieves high recall by ensuring that the set of k' candidate vectors includes many true nearest neighbors. The *compression parameters* in *HAKES-Index*, which include $\mathbf{A'}$ and $\mathbf{b'}$ for dimensionality reduction, and $\mathbf{C^{PQ'}}$ for the PQ codebooks, are fine-tuned to capture the similarity relationship between the query vector and indexed vectors.

At the beginning of training process, $\mathbf{A'}$ and $\mathbf{C^{PQ'}}$ are initialized with $\mathbf{A}$ and $\mathbf{C^{PQ}}$ produced by OPQ. The bias vector $\mathbf{b'}$ is initialized with zero. We then jointly optimize $\mathbf{A'}, \mathbf{b'}$, and $\mathbf{C^{PQ'}}$ to minimize the mismatch between the similarity score distribution after quantization and that of the original d-dimensional space. We only focus on the mismatch in a local region that the training objective is defined based on the similarity score distributions of a sampled query vector $\mathbf{x}$ and its close neighbors ANN_x, because distant vectors are filtered

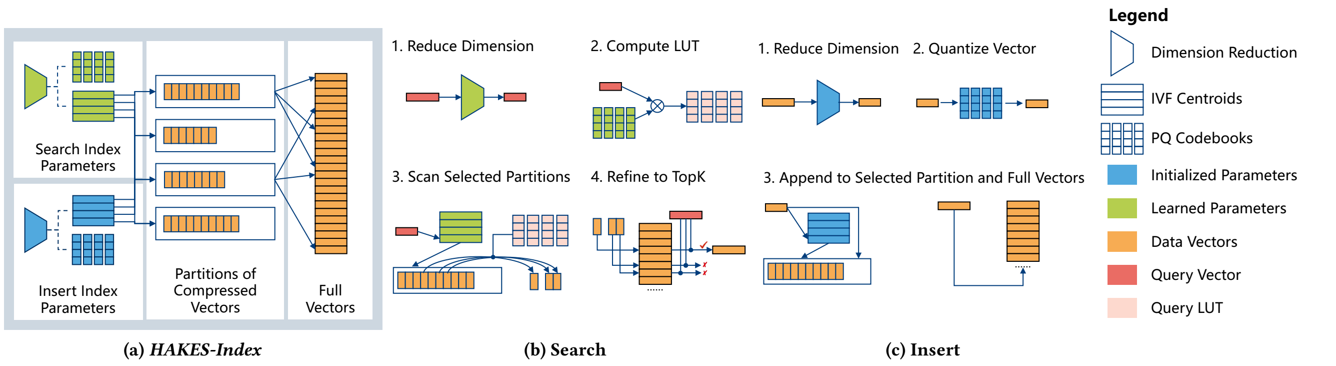


Figure 4: *HAKES-Index* overview.

away by coarse grained IVF partition selection during search in *HAKES-Index*. Specifically, the similarity score distributions before and after the dimensionality reduction are:

$$S_{\mathbf{o,x}} = \text{softmax}([d(\mathbf{x}, \mathbf{v_1}), \dots, d(\mathbf{x}, \mathbf{v_K})]) \quad (2)$$

$$S_{\mathbf{r,x}} = \text{softmax}([d(R'(\mathbf{x}), R(\mathbf{v_1})), \dots, d(R'(\mathbf{x}), R(\mathbf{v_K}))]) \quad (3)$$

where $K = |ANN_x|$ is the number of retrieved close neighbors, and the softmax function converts the similarity scores to a distribution. $R'(\mathbf{x}) = \mathbf{A'x} + \mathbf{b'}$ and $R(\mathbf{v}) = \mathbf{Av} + \mathbf{b}$ represent dimensionality reduction. The similarity scores distribution after quantization is:

$$S_{q,\mathbf{x}} = \text{softmax}([d(R'(\mathbf{x}), q'(R(\mathbf{v_1}))), \dots, d(R'(\mathbf{x}), q'(R(\mathbf{v_K})))]) \quad (4)$$

where the vector approximation $q'(\mathbf{v}) = [q'_1(\mathbf{v}), q'_2(\mathbf{v}), \dots, q'_m(\mathbf{v})]$ from PQ is modified to use both C^{PQ} and $C^{PQ'}$. Specifically, $q_j(\mathbf{v}) = \mathbf{C^{PQ'}}_j[\arg\min_i ||\mathbf{C^{PQ}}_j[i] - \mathbf{v}_j||]$. It means that the indexes of the centroids of the codebook are produced by C^{PQ} and the fine-tuned centroids of $C^{PQ'}$ at the corresponding position are used to approximate the vector.

With the distributions of similarity scores, we can then reduce the mismatch by minimizing the Kullback-Leibler (KL) divergence defined over two pairs of distributions. One pair is defined between the distribution in the original vector space (Equation 2) and that in the vector space after dimensionality reduction (Equation 3). The other pair is between (Equation 2) and the distribution of similarity scores calculated between a query vector after dimensionality reduction and its quantized close neighbors (Equation 4). The overall training objective is as follows:

$$L = -\sum_{\mathbf{x} \in D_{sample}} S_o \log \frac{S_{r,\mathbf{x}}}{S_{o,\mathbf{x}}} - \lambda \sum_{\mathbf{x} \in D_{sample}} S_{o,\mathbf{x}} \log \frac{S_{q,\mathbf{x}}}{S_{o,\mathbf{x}}} \quad (5)$$

where D_{sample} is the sampled query vectors for training, and λ is a hyperparameter to control the strength of the regularization.

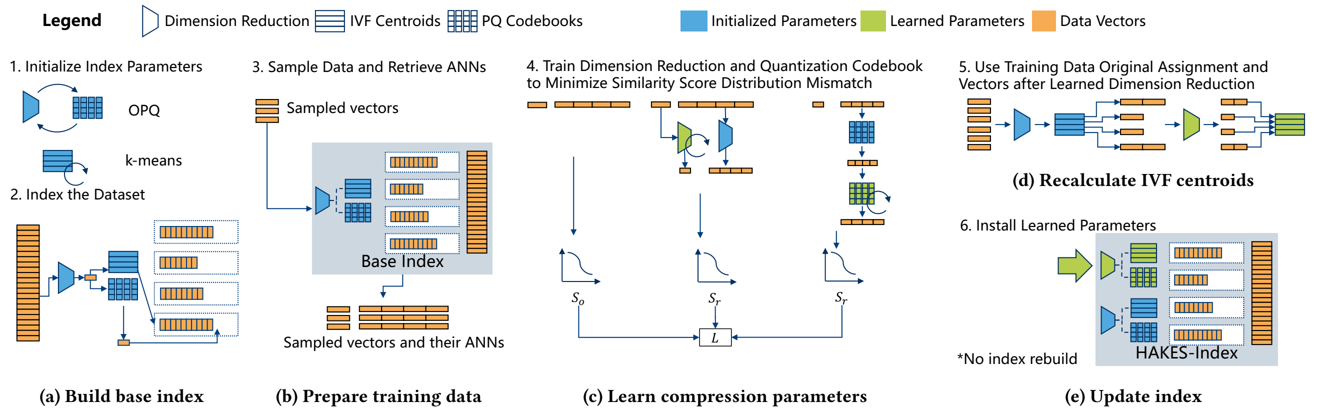
The training process iteratively updates $\mathbf{A'}, \mathbf{b'}, \mathbf{C^{PQ'}}$ to minimize the loss defined in Equation 5 that is to minimize the mismatch among three similarity distributions for close vectors as illustrated in Figure 5c. It stops when the loss reduction computed on the validation set is smaller than a threshold (e.g., 0.1).

3.4 Search Optimizations

HAKES-Index contains two additional optimizations that improve search efficiency. The first is INT8 scalar quantization at each dimension of the IVF centroids. This allows using SIMD to evaluate 4× more dimensions in a single instruction. Although quantization can be lossy, such representation errors are tolerable in practice since the centroids are only used for partition assignment and a large number of partitions are selected for high recall. The second optimization is to adapt the cost of the filter stage based on the query. Fixing the value of *nprobe* means that the computation cost is roughly the same for every query. We note that in extreme cases, all the true nearest neighbors are in the same partition, where only that partition needs to be scanned. In other extreme cases, the true nearest neighbors are evenly distributed among the partitions, where all the partitions need to be scanned to achieve high recall. In high-dimensional space, it is challenging to determine the *nprobe* based solely on the centroids. *HAKES-Index* introduces a heuristic condition for early stopping the scanning of subsequent partitions based on the intermediate search results. The search process ranks the partitions by the similarity score of their centroids to the query, and scans the partitions in order. The key idea is that, as the search process moves away from the query vector, new partitions will contribute fewer vectors to the candidate set. We track the count of consecutively scanned partitions that each partition adds fewer than t vectors to the candidate set, where t is a search configuration parameter. When that count exceeds a specified threshold n_t, it indicates that the search has likely covered all partitions containing nearest neighbors, and we terminate the filter stage. *HAKES-Index* terminates the filter stage either when the heuristic condition above is met, or when *nprobes* partitions have been scanned.

3.5 Discussion

HAKES-Index's two-stage design allows the filter stage to trade accuracy of similarity score evaluation for lower computation overhead. This stage performs aggressive compression, combining dimensionality reduction at the beginning and then 4-bit product quantization. The index parameters are optimized to achieve high compression ratios while preserving only the distribution and not the exact values of similarity scores. The optimization focuses on the local regions instead of globally, since distant vectors in IVF are already filtered out and never evaluated. Our experimental results demonstrate that



Legend
Dimension Reduction
IVF Centroids
PQ Codebooks
Initialized Parameters
Learned Parameters
Data Vectors
1. Initialize Index Parameters
OPQ
k-means
2. Index the Dataset
3. Sample Data and Retrieve ANNs
Sampled vectors
Base Index
Sampled vectors and their ANNs
4. Train Dimension Reduction and Quantization Codebook to Minimize Similarity Score Distribution Mismatch
S_o
S_r
S_r
L
5. Use Training Data Original Assignment and Vectors after Learned Dimension Reduction
6. Install Learned Parameters
*No index rebuild
HAKES-Index

(a) Build base index **(b) Prepare training data** **(c) Learn compression parameters** **(d) Recalculate IVF centroids** **(e) Update index**

Figure 5: End-to-end index construction.

deep embedding vectors can be aggressively compressed to achieve superior throughput-recall tradeoff overall for *HAKES-Index*, with d_r as small as $1/4$ or $1/8$ of the original dimension d, and with 4-bit PQ with $m = 2$. The early termination checking is designed to operate in the filter stage with aggressive compression. It does not rely on accurate similarity score calculation, unlike existing works [28, 61, 63]. The statistics tracking and check incurs minimal overhead, compared to other works on early termination [36, 60].

The compression techniques in *HAKES-Index* differs from those of existing works on quantization, which either focuses on minimizing the reconstruction error [1, 5, 30], i.e., $d(\mathbf{v}, q(\mathbf{v}))$, or the error of similarity score approximation [20, 22, 49], i.e., $d(\mathbf{x}, q(\mathbf{v}))$. *HAKES-Index* learns both dimensionality reduction and quantization together to reduce the distortion of the similarity distribution. Some learned data transformations for quantization [19, 41] aim to transform the original vector to reduce the quantization error. [48] introduces complex data transformation, increases serving complexity, and some other works tune even the embedding models [56, 58], which differ from our ultimate goal of achieving superior throughput-recall trade-off for the ANN search with given embedding vectors. Moreover, we only use the approximate nearest neighbor for training, which can be efficiently obtained compared to ground truth neighbors required by other works [48, 58].

A key design in *HAKES-Index* is that it decouples the management of parameters used for search and insert, enabling its high-recall search while supporting the incorporation of new data. Specifically, it maintains two sets of compression parameters: the learned parameters obtained through training as the search index parameters, and the original parameters established upon initialization as the insert index parameters, as shown in Figure 4a. It is closely related to the lightweight self-supervised training process. As discussed in Section 3.3, we use the prebuilt base index and fix PQ code assignment for training, where all the data vectors are processed only once using the original set of parameters. Consequently, new vectors can follow the same process of being indexed by the initialized parameters and searched by the learned parameters. Empirical observations also confirm that using the learned parameters for inserting new vectors leads to recall degradation in Section 5. Furthermore, as a consequence of the decoupling, the learned search index parameters can be directly applied without re-indexing the vectors. Existing works on learned compression use the updated codebook for assignment during every training iteration [56, 58]. They would require expensive re-indexing of the vectors when applying the trained parameters in vector databases to serve queries.

The aggressive compression employed by *HAKES-Index* not only significantly speeds up the filter stage, but also reduces the memory consumption in this stage. We now analyze the memory cost of *HAKES-Index* for a vector dataset of $(N \cdot 4 \cdot d)$ bytes. The dimensionality reduction matrices and the bias vector take $(2 \cdot 4 \cdot d \cdot d_r + 4 \cdot d_r)$ bytes. IVF centroids and the 4-bit quantization codebooks consume $(N_c \cdot 4 \cdot d_r + N_c \cdot d_r)$ bytes and $(2 \cdot 2^4 \cdot 4 \cdot d_r)$ bytes respectively. The compressed vectors take $(N \cdot (1/2) \cdot (d_r/m))$ bytes. The filter stage index is significantly smaller than the vector dataset.

As the dataset grows considerably, the index should be rebuilt with a larger number of IVF partitions. In practice, even the embedding models that generate the vectors are frequently retrained, for example, on a daily basis in recommendation systems [62]). After model training, rebuilding the index is necessary.

4 THE *HAKES* DISTRIBUTED VECTORDB

4.1 Overview

HAKES-Index processes a search query in two stages, namely the filter and refine stage. These stages do not share data, and they have distinct resource requirements due to the types and amount of vectors being evaluated. Specifically, the memory consumption of the filter stage, accessing compressed vectors, is significantly lower than that of the refine stage, which accesses the original vectors. In addition, the filter stage has a much higher computation cost because it performs computation over a large number of vectors.

We design an architecture that exploits the filter-and-refine design to disaggregate the two stages. In particular, we separate the management of the filter-stage index from the full-precision, original vectors used only in the refine stage, and employ different scaling policies for them in a server cluster. There are two sets of workers, the IndexWorkers and the RefineWorkers, each performs one stage of the index using the local data. The former are responsible for the filter stage, managing the replicated compressed vectors. The latter performs the refine stage, storing shards of the original vectors. Figure 6 shows an example in which a physical server runs both an IndexWorker and a RefineWorker. However, we stress that

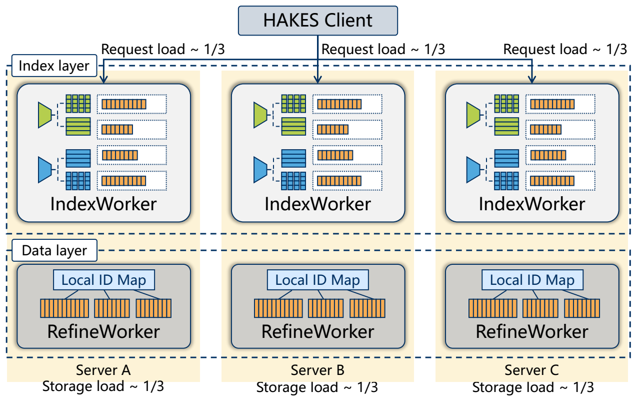


Figure 6: *HAKES* architecture.

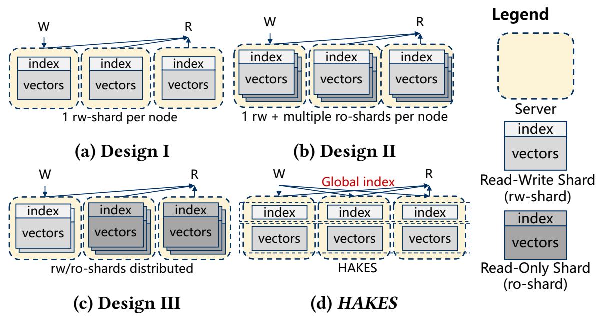


Figure 7: Distributed vector database architectures.

these components can be disaggregated and scaled independently. For example, more memory nodes running RefineWorkers can be added to handle a large volume of data, and more compute nodes running IndexWorker can be added to speed up the filter stage.

Discussion. *HAKES*'s architecture is different from that of existing distributed vector databases. Figure 7 compares four architectures with distinct shard layouts and communication for read and write. In the first architecture (Figure 7a), adopted by [9, 54], each server hosts a single read-write shard and maintains its index. A read request merges search results from every node, while a write request is routed to a single server based on a sharding policy. In the second architecture (Figure 7b), used by [13], each node maintains one read-write shard and multiple read-only shards to reduce the read-write contention. The third architecture in Figure 7c extends the first two by employing multiple read-write shards and multiple read-only shards. It is adopted by [21, 51], and supports scaling out of read or of write by adding servers for the required type of shards. We note that in these three architectures, an index is local to the shard data, i.e., the index of each shard is not constructed over the global set of vectors. However, building many small indexes over multiple shards incurs significant overhead, as we show in our evaluation later. *HAKES*'s architecture in Figure 7d, in contrast, maintains the global index at each server, since the filter stage index is small due to compression and supports efficient update.

The index in the filter stage scales with dataset size. However, *HAKES*'s high compression ratio enables a single cloud server to host TB-scale indexes. For deployments where the index exceeds individual server capacity, the index is dynamically sharded across IndexWorker groups. Searches query one replica per shard group while updates propagate atomically to all replicas in the affected group. Full-precision vectors remain managed separately by RefineWorker nodes deployed on distinct servers, ensuring physical isolation between filter and refine stages.

4.2 *HAKES* Design

The *IndexWorker* maintains a replica of the filter-stage index and the compressed vectors organized in IVF partitions. It takes a query vector as input and returns a set of candidate vectors. IndexWorker is compute-heavy. It implements dynamic batching with internal, lock-free task queues. In particular, vectors from different requests are batched into a matrix such that the dimensionality reduction and IVF assignment can be computed efficiently via matrix-matrix multiplication. Requests are batched only under high load, otherwise, they are processed immediately on separate CPU cores.

The *RefineWorker* maintains a shard of the original vectors. It handles the refine stage, which evaluates similarity scores between the query and the candidate vectors belonging to the shard. *HAKES* supports two types of sharding policies for the full vectors. One policy is sharding by vector ID, in which vectors are distributed (evenly) among the nodes by their IDs. The other is sharding by IVF assignment, in which vectors belonging to the same IVF partition are on the same RefineWorker. This policy helps reduce network communication because the refine stages only happen on a small number of nodes.

Operation workflow. Before serving queries, *HAKES* builds an index over a given dataset. It first takes a representative sample of the dataset to initialize the base index parameters. It then launches IndexWorkers that use the base index. Next, it inserts the vectors, and after that starts serving search requests. It builds training datasets for learning index parameters by collecting the results of ANN queries. Once the training process finishes, it installs the new parameters to all IndexWorkers with minimal disruption. Specifically, at every IndexWorker node, the new parameters are loaded to memory and the pointers in *HAKES-Index* are redirected to them.

During search, the client sends the query to an IndexWorker and gets back the candidate vectors. Based on the sharding configuration, the client sends these vectors to the corresponding RefineWorkers in parallel. The client reranks the vectors returned by the RefineWorkers and outputs the top *k* vectors. During insert, the client sends the new vector to the RefineWorker that manages the shard where the vector is to be inserted. The client then picks an IndexWorker to compute the new quantized vector and update the IVF structure. This update is broadcast to all the IndexWorkers. For deletion, the client broadcasts the vector IDs to be deleted to all the IndexWorkers, which then mark them as deleted in their filter-stage index.

Consistency and failure recovery *HAKES* does not guarantee strong consistency, which is acceptable because applications relying on vector search can tolerate that [51, 55]. It can support session consistency by synchronously replicating the write requests or having the client stick to an IndexWorker. *HAKES* periodically creates checkpoints of the index. During crash recovery, new vectors after the checkpoints are re-inserted into the RefineWorkers and IndexWorkers.

5 EVALUATION

In this section, we benchmark *HAKES-Index* against state-of-the-art ANN indexes and *HAKES* against state-of-the-art distributed vector databases to study the effectiveness of our design.

5.1 Implementation

We implement *HAKES-Index* by extending the FAISS library [10]. IndexWorker and RefineWorker are implemented on top of the index *HAKES-Index*, and they are accessible via an HTTP server implemented using libuv and llhttp. The index extension and serving system take ~ 7000 LoC in C++. The index training is implemented in ~ 1000 LoC in Python, using Pytorch@1.12.1. The *HAKES* client is implemented in Python in ~ 500 LoC.

5.2 Experiment Setup

Datasets and workloads. As listed in Table 1, we use six deep embedding datasets and the GIST dataset. Five of the datasets are at the 1-million scale, and we use them for index benchmarking.

- DPR-768 is generated by the Dense Passage Retrieval (DPR) context encoder model [32] on text records sampled from the Sphere web corpus dataset[1].
- OPENAI-1536 [12] is generated by OpenAI's embedding service on DBpedia text data [50].
- MBNET-1024 is generated by pretrained MobileNet [25] on one million ImageNet data [47].
- RSNET-2048 is generated by pretrained ResNet [24] on 1 million ImageNet data.
- GIST-960 is a widely in the literature for benchmarking ANN indexes [3, 17, 38]. We selected GIST for its high dimensionality.

We also use two other large datasets for in-depth analysis of our index and system.

- DPR-768-10m: use the same embedding model as DPR-768 but on 10 million Sphere text records.
- E5-1024-10m: is generated with the E5-large text model [52] on 10 million Sphere text records.

We normalize the vectors and use the inner product as the similarity metric due to its popularity in existing embedding services[2] [3]. This metric is also the default choice in all of the baseline systems [33, 51, 54]. We note that for normalized vectors, Euclidean distance, cosine similarity, and inner product are equivalent with respect to neighbor relationships. The search quality is measured by Recall10@10. The ground-truth nearest neighbors for the queries are generated by a brute-force search over the entire dataset.

Training setup. Index training is conducted on an Ubuntu 18.04 server that has an Intel Xeon W-2133@3.60GHz CPU with 6 cores and an NVIDIA GeForce RTX 2080 Ti GPU. The λ parameter is searched in the set $\{0.01, 0.03, ...30\}$. The AdamW Optimizer is used with a learning rate value in the set $\{10^{-5}, 10^{-4}, 10^{-3}\}$. The batch size is set to 512. We use 100,000 samples and their 50 neighbors returned by the base index at $nprobe = 1/10$ and $k'/k = 10$.

Environment setup. We conduct all the index experiments on a Ubuntu 20.04 server equipped with an Intel Xeon W-1290P @ 3.70GHz CPU with 10 cores and 128 GiB memory, and distributed experiments in a cluster of servers with the same specification.

[1] https://weaviate.io/blog/sphere-dataset-in-weaviate
[2] https://platform.openai.com/docs/guides/embeddings
[3] https://docs.mistral.ai/capabilities/embeddings/

Table 1: High-dimensional datasets.

Dataset	N	d	nq	Size (GiB)	Type
DPR-768	1000000	768	10^4	2.86	Text
OPENAI-1536	990000	1536	10^4	5.72	Text
MBNET-1024	1103593	1024	10^3	4.21	Image
RSNET-1024	1103593	2048	10^3	8.42	Image
GIST-960	1000000	960	10^3	3.58	Image
DPR-768-10m	9000000	768	10^6	26	Text
E5-1024-10m	9000000	1024	10^6	35	Text

Index baselines. We select 12 state-of-the-art in-memory ANN index baselines.

- IVF is the classic IVF index with k-means clustering.
- IVFPQ_RF applies PQ with IVF and uses FAISS 4-bit quantization fast scan implementation [2, 10]. The RF denotes a refine stage
- OPQIVFPQ_RF uses OPQ [19] to learn a rotation matrix that minimizes the PQ reconstruction error. We use the OPQ implementation in FAISS that can generate a transformation matrix $\in \mathbb{R}^{d \times d_r}$ to reduce dimension before IVF and PQ.
- HNSW [42] is the index used in almost all vector databases.
- ELPIS [4] partitions the dataset and maintains an HNSW graph index for each partition, representative for maintaining multiple subgraph indexes.
- LSH-APG [64] leverages LSH to identify close entry points on its graph index to reduce the search path length.
- ScaNN [22], SOAR [49], and RaBitQ [18] are recent proposed quantization scheme used with partitioning-based index. They use reranking to improve recall.
- Falconn++ [45] and LCCS [34] are state-of-the-art LSH indexes.
- LVQ [1] is a state-of-the-art graph index with quantization. It supports 4-bit scalar quantization, followed by 8-bit quantization on residual for reranking.

We do not compare against recent indexes that are optimized for secondary storage [6, 53, 57], which report lower performance than in-memory indexes. The first three IVF baselines share the codebase of our extended FAISS library. For the remaining baselines, we use the implementations provided by the authors.

Distributed vector database baselines. We select three popular distributed vector databases that employ in-memory ANN indexes. They cover the three architectures described in Section 4.1. We set up the systems according to the recommendations from their respective official documentation.

- Weaviate [54] adopts an architecture in which each server maintains a single read-write shard and an HNSW graph. It implements HNSW natively in Golang with fine-grained node-level locking for concurrency. We deploy Weaviate using the official Docker image at version v1.21.2
- Cassandra [33] adds support for vector search recently [13] on its NoSQL database. It shards the data across nodes, and every node maintains a read-write shard and multiple read-only shards that are periodically merged. It uses jVector [11], a graph index that only searches quantized vectors, similar to DiskANN [29].

- Milvus [21, 51] adopts an architecture in which there is one shard that processes updates. Once reaching 1GiB, this shard becomes a read-only shard with its own index and is distributed across the servers for serving. We deploy Milvus version 2.4 using the official milvus-operator v0.9.7 on a Kubernetes (v1.23.17) cluster.

Besides the three systems above, we add two more baselines, called Sharded-HNSW and HAKES-Base. Sharded-HNSW adopts Weaviate's architecture, and uses our server implementation with hnswlib. This baseline helps isolate the performance impact of the index and system design, since the three vector databases are implemented with different languages and have different sets of features. HAKES-Base is the same as *HAKES* but employs the base index, that is, without parameter training or optimizations.

5.3 Index Benchmarking and Analysis

For each index, we explore the range of configurations recommended in the original paper and corresponding code repository, and pick the best configuration for each dataset. We then run experiments with varying search parameter values to examine the index's throughput-recall tradeoff. For the complete set of explored and selected configurations of all indexes, please refer to [26].

Sequential read workload. Figure 8 compares the throughput-recall tradeoff of the 13 indexes for the recall range above 80%. Across the different datasets, *HAKES-Index* achieves state-of-the-art throughput-recall tradeoff. At high recall, it even outperforms the recent quantized graph index, LVQ, which is heavily optimized for prefetching and SIMD acceleration. The performance difference among OPQIVFPQ_RF, IVFPQ_RF, and IVF confirms that with a refine stage, deep embeddings can be compressed significantly with quantization and dimensionality reduction for efficiency while maintaining high accuracy. Across the deep embedding datasets, OPQIVFPQ_RF and *HAKES-Index* achieve the reported tradeoff with d_r/d =1/4 or 1/8, significantly reducing computation.

Recent quantization-based indexes, namely ScaNN, SOAR, and RaBitQ, show mixed results compared to IVFPQ_RF, which uses standard PQ and fast scan implementation. ScaNN improves the quantization for inner product approximation; SOAR aims to reduce the correlation of multiple IVF partitions assignment for one vector; and RaBitQ uses LSH to generate binary code representation and decide vectors to be reranked with its error bound. ScaNN and SOAR outperform IVFPQ_RF on GIST-960, DPR-768, and MBNET-1024, but have comparable performance on RSNET-2048 and OPENAI-1536. RaBitQ only performs better than IVFPQ_RF on GIST-960. These observations highlight the importance of evaluating indexes on high-dimensional deep embedding.

The performance of Falconn++ and LCCS ranks below IVF, confirming that LSH-based indexes are less effective in filtering vectors than the data-dependent approaches in high-dimensional space [3, 38]. Among graph-based indexes, LVQ performs best as its scalar quantization avoids computation using full vectors for graph traversal. The difference between HNSW and LSH-APG indicates that the hierarchical structure of HNSW is more effective than the LSH-based entry point selection in LSH-APG in high-dimensional space. The gap between HNSW and ELPIS shows that sharding a global graph index into smaller subgraphs degrades the overall performance. We analyze that phenomenon in distributed vector databases in Section 5.5.

Table 2: Ablation study where recall is in the 0.99 region. Each cell shows the QPS (recall) value.

	Base	Learn	Learn + SQ	All
DPR-768	1233 (0.979)	1238 (0.991)	1280 (0.990)	1280 (0.990)
OPENAI-1536	977 (0.990)	974 (0.994)	976 (0.994)	1389 (0.991)
MBNET-1024	2393 (0.977)	2413 (0.991)	2579 (0.991)	2791 (0.991)
RSNET-1024	2330 (0.982)	2350 (0.992)	2444 (0.992)	2615 (0.992)
GIST-960	610 (0.944)	625 (0.989)	631 (0.989)	747 (0.988)

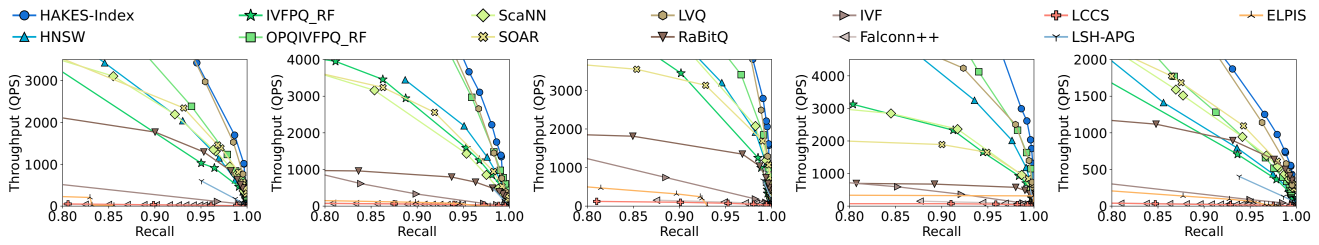
Read-write workload. For indexes supporting inserts, we first evaluate their performance with sequential read-write workloads. We focus on high-recall regions of 0.99, and vary the write ratio from 0.0 to 0.5. Figure 9 reveals that as the write ratio increases, partitioning-based indexes have a clear advantage over graph indexes. Both LVQ's and HNSW's performance decrease as the write ratio increases, because inserting new data into a graph is slower than serving an ANN search. The reverse is true for partitioning-based indexes, since insert does not involve comparison with existing vectors. The exceptions are ScaNN in RSNET and SOAR, which select quantized code with additional constraints. In particular, SOAR assign a vector to multiple partitions based on their correlation which is more costly than a single assignment used by other partitioning-based indexes. *HAKES-Index* outperforms all baselines across all datasets, because of its efficient search and insert.

We further evaluate the indexes supporting concurrent read-write workloads. The HNSW implementation in hnswlib supports concurrent read-write with fine-grained locking on the graph nodes, and our FAISS extension supports partition locking for IVFPQ_RF, and OPQIVFPQ_RF similar to *HAKES-Index*. We use 32 clients and vary the ratio of write requests. Figure 10 shows that partitioning-based indexes are better than HNSW, due to low contention and a predictable memory access pattern. We note that even IVFPQ_RF reaches a comparable or higher throughput than HNSW for concurrent read. The performance gaps increase with more writes.

Memory consumption. The cost of storing the original vectors dominates the index's memory consumption. We discuss the memory overhead for representative baselines on OPENAI-1536 as an example. We measure memory usage before and after loading the indexes. HNSW maintains the connection information for each node at each level on top of the original data, increasing the memory from 5.72 to 6.01 GiB. For IVFPQ_RF, OPQIVFPQ_RF, and *HAKES-Index*, the main overhead is storing the compressed vectors. IVFPQ_RF consumes 5.92 GiB, whereas OPQIVFPQ_RF takes 5.86 GiB due to dimensionality reduction. *HAKES-Index* consumes 5.86 GiB similar to OPQIVFPQ_RF, as the additional index parameters are small.

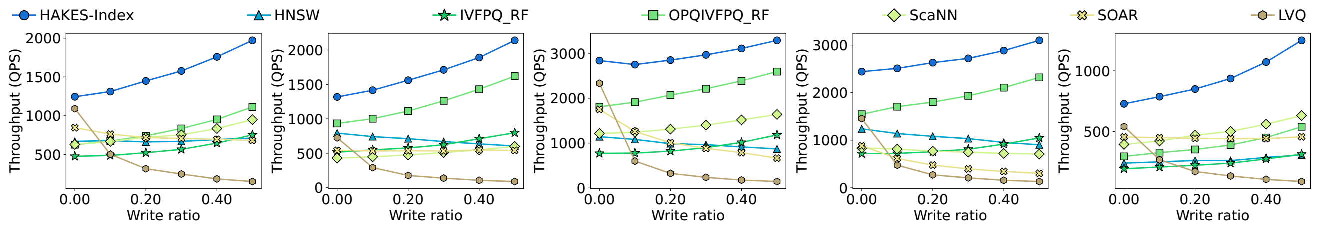
5.4 *HAKES-Index* Analysis

Performance gain breakdown. Table 2 shows how the different techniques contribute to the performance of *HAKES-Index*. We report the results at the search configurations that achieve *recall* ≈ 0.99 with the learned parameters. The learned compression contributes the most as it improves the throughput-recall tradeoff over the base settings. Scalar quantization of IVF centroids and early termination provide further improvement to throughput without significantly degrading recall. We used the same setting



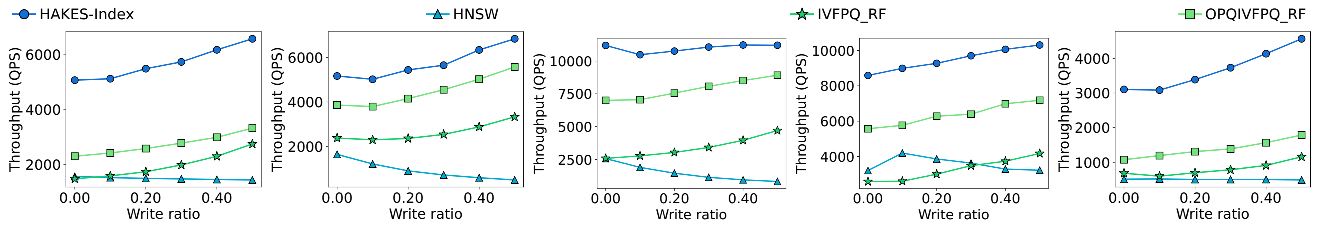
(a) DPR-768 **(b) OPENAI-1536** **(c) MBNET-1024** **(d) RSNET-2048** **(e) GIST-960**

Figure 8: Throughput vs. recall for sequential reads (recall ≥ 0.8).



(a) DPR-768 **(b) OPENAI-1536** **(c) MBNET-1024** **(d) RSNET-2048** **(e) GIST-960**

Figure 9: Performance under sequential read-write workloads. (Recall=0.99).



(a) DPR-768 **(b) OPENAI-1536** **(c) MBNET-1024** **(d) RSNET-2048** **(e) GIST-960**

Figure 10: Performance under concurrent read-write workloads. (Recall=0.99).

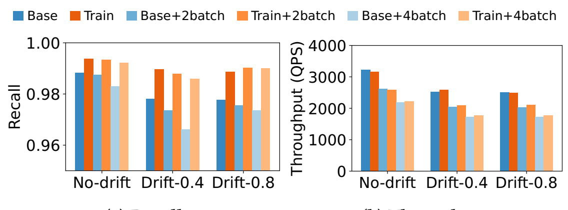
$t = k'/200$ and $n_t = 30$ for early termination, which improves the throughputs considerably on 4 of the 5 datasets. However, as discussed in the previous subsection, the heuristic can terminate the search prematurely and miss the true close neighbors, thereby leading to lower recall. We note that careful tuning on a dataset can achieve better performance for a specific recall target.

Recall improvement by learned compression. Table 3 reports the recalls for different search configurations, for 10-million scale datasets. We note that the training process does not affect the cost of performing dimensionality reduction and of scanning quantized vectors. In other words, given the same search parameters, the performance is only affected by the IVF partition selection, which we observe to be negligible. Table 3 shows consistent improvement across all configurations on the 10-million scale datasets. The improvement is between 0.07 to 0.14 for $k'/k = 10$ and 0.01 to 0.07 for those settings with recall over 0.9. It is higher for smaller filter candidate sets (i.e. smaller k'/k), which is expected because the impact of high-quality candidate vectors is higher. This improvement allows *HAKES-Index* to reach high recall with a smaller *nprobe* and k'/k, which translates to higher throughput. We attribute the high recalls to the training process that results in the refine stage having more true nearest neighbors. We discuss the results on 1-million scale datasets in our extended version [26].

Training cost. The cost of constructing *HAKES-Index* consists of the cost of building the base index and of training the compression parameters. Deploying the trained parameters incurs negligible overhead, as it only loads a small dimensionality reduction matrix, bias vector, quantization codebooks, and IVF centroids into memory. For the 10-million scale datasets, building the base index takes 179.2s and 219.22s for initializing the OPQ and IVF parameters, and 103.2s and 125.6s to insert the 10 million vectors for DPR-768-10m and E5-1024-10m, respectively. It takes 52.9s and 60.9s for the two datasets respectively to sample the training set with 1/100 ratio and compute the approximate nearest neighbors with *nprobe* set to the 1/10 partitions and $k'/k = 10$. Training takes 34.9s and 45.6s, respectively. In a server cluster, the time to insert vectors and prepare the training set can be reduced linearly with the number of nodes. In comparison, constructing the HNSW graph takes 5736.4s and 9713.21s on the two datasets, which are 15.5× and 21.5× higher than the cost of building *HAKES-Index*. We note that in production, *HAKES-Index* can use the initialized parameters to serve requests, while training is conducted in the background using GPUs. The

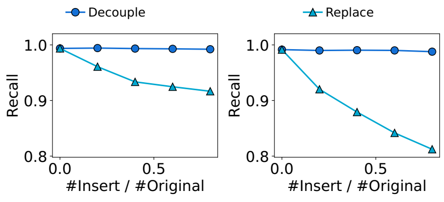
Table 3: Recall improvement at different search configurations.

IVF *nprobe* (total 8192)		200			400			600			800		
k'/k		10	50	200	10	50	200	10	50	200	10	50	200
DPR-768-10m	Base	0.722	0.904	0.968	0.725	0.909	0.976	0.725	0.910	0.979	0.725	0.911	0.980
	Learned	**0.859**	**0.963**	**0.980**	**0.866**	**0.973**	**0.993**	**0.868**	**0.976**	**0.996**	**0.869**	**0.976**	**0.997**
E5-1024-10m	Base	0.765	0.896	0.942	0.773	0.910	0.959	0.777	0.914	0.966	0.778	0.917	0.969
	Learned	**0.843**	**0.932**	**0.953**	**0.856**	**0.950**	**0.974**	**0.860**	**0.955**	**0.979**	**0.862**	**0.958**	**0.983**



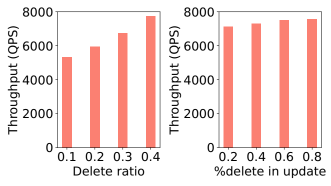
(a) Recall (b) Throughput

Figure 11: Tolerance against data drift (MBNET-1024).



(a) MBNET-1024 (b) RSNET-2048

Figure 12: Decoupling index parameters.



(a) R+D (b) 60%R

Figure 13: Performance under delete (R: read, D: delete).

learned parameters can be seamlessly integrated once available, without rebuilding the index.

Drift tolerance. We prepare 1-million datasets derived from the ImageNet dataset. We reserve 1/10 categories for generating drift. We use a mixing ratio of vectors from the reserved categories and those from the original categories (not in the 1 million for index building) to create workloads with different drifts. The workloads consist of 4 batches of 200k vectors for insertion and 1k query vectors, such that both insertion and query exhibit drift. Figure 11 shows the recall and throughput as we insert data batches and then run ANN queries with a mixing ratio from 0 to 0.8. The *nprobe* and k'/k are selected to be the best search configuration with recall $\geq$ 0.99. The throughput descrease as more vectors are added resulting in more vectors to scan in each partition. For search quality, we observe at this high recall, the recall improvement of training persists across different drifts. As more data are added the recall degrades slightly. The result showed the robustness of IVF and *HAKES-Index* training process against moderate drifts for embeddings from the same model. We also evaluate on RSNET-2048 and observe similar results. For embeddings from different models or entirely distinct sources, we recommend building different indexes.

Decoupling index parameters for read and write. We start with an index on 1 million vectors and select the *nprobe* and k'/k for recall $\geq$ 0.99. We insert batches of 200k vectors and measure the recall at the same configuration. We derive the true nearest neighbor after each batch insert in prior. Figure 12 shows the importance of separating the learned parameters for search from the parameters for insert. If the learned parameters are used to compress new vectors during insert, the recall drops. The reason is that only keeping learned parameters is inconsistent with our training scheme and the approximated similarity will not follow the expected distribution, as discussed in Section 3.5. We observe in experiments that new vectors that are not nearest neighbors can have a higher approximated similarity than true neighbors, and the true neighbors in the added data can have significantly lower approximated similarity than those neighbors in the original dataset.

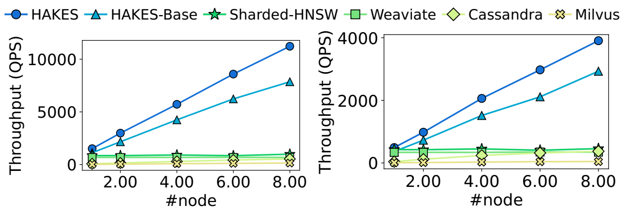
Deletion. We evaluate the index performance under deletion using the DPR-768 dataset and 32 clients. We select search parameters that achieve recall=0.99. Figure 13a shows that for workloads involving both search and deletion, the throughput increases with the ratio of deletions. The trend is similar to the results in Figure 10 when the ratio of insert increases. The higher throughput is because insertion and deletion are cheaper than search operations. Figure 13b shows that under workloads of 60% search and 40% of insert and delete, the throughput is only slightly higher when varying the ratio of delete, since insert operations calculate IVF assignment and compress the vectors. Since we do not modify the coarse-grained partitioning when deleting the data, the recall can be maintained, as the close neighbors are likely to be selected from nearby partitions.

For additional results on full-range throughput-recall tradeoff, effects of Euclidean distance metric, deletion, ablation study for training and early termination, please refer to [26].

5.5 System Comparison

We compare the performance of *HAKES* against the five distributed vector database baselines at 0.98 recall for $k = 10$, using the two 10-million scale datasets. For Cassandra, we use the same configuration for graph and beam search width during index construction. However, since it uses quantized vectors instead of the original vectors, we adjust k to be larger than 10, such that if the refine stage is performed, a recall of 0.98 is reached. Specifically, the system uses a quantized graph index to return a larger number of candidate vectors, which are then processed by a refine stage to achieve a recall of 0.98. For *HAKES*, Sharded-HNSW, Weaviate, and Cassandra, we run one shard per node. For Milvus, we run a number of virtual QueryNode according to the number of node used for other systems. The QueryNodes are evenly distributed among the physical nodes in a Kubernetes cluster. We use multiple distributed clients to saturate the systems, then report the peak throughputs.

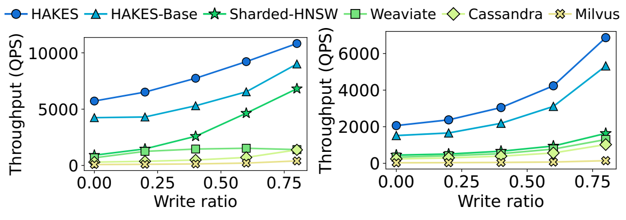
Scaling with the number of nodes. Figure 14 compares the systems' throughputs with varying numbers of nodes. It can be seen that *HAKES* and *HAKES*-Base scale linearly because the loads of



(a) DPR-768-10m

(b) E5-1024-10m

Figure 14: Scalability under read-only workload.



(a) DPR-768-10m

(b) E5-1024-10m

Figure 15: Throughputs under concurrent workload.

both the filter and refine stages are distributed evenly across the nodes. The filter stage of concurrent requests can be processed at different nodes in parallel. In Weaviate, a request is sent to all the shards. Although the graph index size and the number of vectors in each shard decrease with more nodes, the search cost at each shard does not decrease linearly. This is consistent with the results of ELPIS in Section 5.4, confirming that graph indexes do not scale well by partitioning. Sharded-HNSW achieves slightly better throughput than Weaviate, but shows the same trend. Milvus' throughput increases with the number of read shards, due to the reduced read load. However, the small read shard size of 1GiB leads to a large number of subgraphs (e.g., over 30 for E5-1024-10m), all of which are searched, limiting the throughput. In Cassandra, a single node contains multiple shards, the number of which is affected by its Log-structured merge (LSM) tree compaction process. We observe that the number of shards per node decreases as the number of nodes increases, which explains the increasing throughput. At 8 nodes, there is one shard per node and the performance is similar to that of Weaviate and Sharded-HNSW. The improvement of *HAKES* over *HAKES*-Base shows the benefit of *HAKES-Index* in reducing the search cost with its learned compression and optimizations.

Performance under concurrent read-write workloads. We fix 4 nodes for all systems and vary the write ratio. Figure 15 shows that all systems have higher throughput as the write ratio increases. For Weaviate, Sharded-HNSW, and Cassandra, the write request is only processed by one shard, as opposed to by all the shards for read requests. Sharded-HNSW has the highest performance among baselines that use graph-based indexes, due to its C++ implementation. *HAKES* and *HAKES*-Base outperform all the other baselines by a considerable margin, and *HAKES* has higher throughputs than *HAKES*-Base. Even though each write request needs to be processed by all IndexWorkers, *HAKES* is more efficient than the others in processing the write request, because it only computes the quantized vector and updates the IVF structure. In the other baselines, each node has to perform a read to identify neighbor vectors and network edges to be updated.

6 RELATED WORK

Managing ANN index update. Graph indexes, like HNSW [42] and LVQ [1], rebuild the graph connections locally. SPFresh [57] uses an in-memory graph to index a large number of partitions of vectors on disk and proposes a scheme to keep the partition size small for stable serving latency. At the system level, sharding is employed to reduce the impact of inserting new vectors [9, 21]. *HAKES-Index* appends vectors to the partitioning-based index and uses tombstone for deletion to minimize interference on search and maintain the high recall without changing the search configurations. The low read-write contention allows *HAKES* to maintain replicated global indexes for better scaling performance.

Adaptive query search. Several works exploit characteristics of the queries and the immediate search results to improve the vector search. Auncel [63], iDistance [28], and VBASE [61] leverage precise similarity scores to determine if a search can terminate early, making them unsuitable for the *HAKES-Index*'s filter stage that operates on compressed vectors. ADSampling [17] progressively uses more dimensions to compare vector pairs. Learning-based methods like LEQAT [36, 60] employ predictive models that incur costly training and inference overhead. In contrast, *HAKES-Index*'s early termination check is lightweight as it is based on simple computation over statistics available during search.

Vector quantization. ScaNN [22], SOAR [49], and QUIP [20] learn quantization to reduce the approximation error of the inner product. RabitQ [18] quantizes vectors into binary representations and provides a theoretical error bound on the similarity score. OPQ [19] and DOPQ [41] learn data transformation and quantization codebooks to reduce the error in reconstructing the original vectors. [48] learns a transformation matrix to spread out vectors for quantization assignment, yet keeping neighbors close. These works have a different optimization objective from ours. In particular, we learn the dimensionality reduction and quantization together to reduce the local similarity score distribution mismatch. Other works from the information retrieval community [56, 58, 59] propose to jointly train embedding models and PQ codebooks, yet the objectives differ, and they require access to the embedding model or semantic labels.

7 CONCLUSION

We presented a scalable, distributed vector database *HAKES* that supports approximate nearest neighbor search with high recall and high throughput for online services that are subject to concurrent read-write workloads. HAKES employs a novel partitioning-based index that adopts a two-stage process with learned compression parameters. We proposed a lightweight training process and a separation of index parameters to support vector insert. HAKES adopts a disaggregated architecture specifically designed to exploit the access pattern of the new index. We compared HAKES against existing distributed vector databases, showing that our system achieves up to 16× throughputs over the baselines at high recall regions.

ACKNOWLEDGMENTS

We sincerely thank the reviewers for their valuable feedback. The work of NUS researchers is partially supported by the Lee Foundation in terms of Beng Chin Ooi's Lee Kong Chian Centennial Professorship fund.

REFERENCES

[1] Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore Willke. 2023. Similarity Search in the Blink of an Eye with Compressed Indices. *Proc. VLDB Endow.* 16, 11 (2023), 3433–3446.

[2] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2015. Cache Locality Is Not Enough: High-Performance Nearest Neighbor Search with Product Quantization Fast Scan. *Proc. VLDB Endow.* 9, 4 (2015), 288–299.

[3] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. *Information Systems* 87 (2020), 101374.

[4] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. ELPIS: Graph-Based Similarity Search for Scalable Data Science. *Proc. VLDB Endow.* 16, 6 (2023), 1548–1559.

[5] Artem Babenko and Victor Lempitsky. 2014. Additive Quantization for Extreme Vector Compression. In *2014 IEEE Conference on Computer Vision and Pattern Recognition.* 931–938.

[6] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-Efficient Billion-Scale Approximate Nearest Neighborhood Search. In *Advances in Neural Information Processing Systems.* 5199–5212.

[7] Soham Deshmukh, Benjamin Elizalde, Rita Singh, and Huaming Wang. 2023. Pengi: An Audio Language Model for Audio Tasks. In *Advances in Neural Information Processing Systems.* 18090–18108.

[8] Yihe Dong, Piotr Indyk, Ilya P. Razenshteyn, and Tal Wagner. 2020. Learning Space Partitions for Nearest Neighbor Search. In *8th International Conference on Learning Representations.*

[9] Ishita Doshi, Dhritiman Das, Ashish Bhutani, Rajeev Kumar, Rushi Bhatt, and Niranjan Balasubramanian. 2021. LANNS: A Web-Scale Approximate Nearest Neighbor Lookup System. *Proc. VLDB Endow.* 15, 4 (2021), 850–858.

[10] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss Library. *CoRR* abs/2401.08281 (2024).

[11] Jonathon Ellis. 2024. JVector. Retrieved April 12, 2024 from https://github.com/jbellis/jvector

[12] Hugging Face. 2024. KShivendu/dbpedia-entities-openai-1M. Retrieved April 12, 2024 from https://huggingface.co/datasets/KShivendu/dbpedia-entities-openai-1M

[13] The Apache Software Foundation. 2024. Apache Cassandra® 5.0: Moving Toward an AI-Driven Future. Retrieved April 12, 2024 from https://cassandra.apache.org/_/Apache-Cassandra-5.0-Moving-Toward-an-AI-Driven-Future.html

[14] Cong Fu, Changxu Wang, and Deng Cai. 2022. High Dimensional Similarity Search with Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 8 (2022), 4139–4150.

[15] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search with the Navigating Spreading-Out Graph. *Proc. VLDB Endow.* 12, 5 (2019), 461–474.

[16] Jinyang Gao, H.V. Jagadish, Beng Chin Ooi, and Sheng Wang. 2015. Selective Hashing: Closing the Gap Between Radius Search and k-NN Search. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* 349–358.

[17] Jianyang Gao and Cheng Long. 2023. High-Dimensional Approximate Nearest Neighbor Search: with Reliable and Efficient Distance Comparison Operations. *Proc. ACM Manag. Data* 1, 2, Article 137 (2023), 27 pages.

[18] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 3, Article 167 (2024), 27 pages.

[19] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 4 (2014), 744–755.

[20] Ruiqi Guo, Sanjiv Kumar, Krzysztof Choromanski, and David Simcha. 2016. Quantization Based Fast Inner Product Search. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics.* 482–490.

[21] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, Zhenshan Cao, Yanliang Qiao, Ting Wang, Bo Tang, and Charles Xie. 2022. Manu: A Cloud Native Vector Database Management System. *Proc. VLDB Endow.* 15, 12 (2022), 3548–3561.

[22] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *Proceedings of the 37th International Conference on Machine Learning.* 3887–3896.

[23] Gaurav Gupta, Tharun Medini, Anshumali Shrivastava, and Alexander J. Smola. 2022. BLISS: A Billion Scale Index Using Iterative Re-Partitioning. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining.* 486–495.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition.* 770–778.

[25] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017).

[26] Guoyu Hu, Shaofeng Cai, Tien Tuan Anh Dinh, Zhongle Xie, Cong Yue, Gang Chen, and Beng Chin Ooi. 2025. HAKES: Scalable Vector Database for Embedding Search Service (Extended Version). https://github.com/nusdbsystem/HAKES-Search/tree/main/extended-version

[27] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing.* 604–613.

[28] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. 2005. iDistance: An Adaptive B^+-Tree Based Indexing Method for Nearest Neighbor Search. *ACM Trans. Database Syst.* 30, 2 (2005), 364–397.

[29] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-Point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems.* 13766–13776.

[30] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128.

[31] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in One Billion Vectors: Re-Rank with Source Coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing.* 861–864.

[32] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing.* 6769–6781.

[33] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (2010), 35–40.

[34] Yifan Lei, Qiang Huang, Mohan Kankanhalli, and Anthony K. H. Tung. 2020. Locality-Sensitive Hashing Scheme Based on Longest Circular Co-substring. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 2589–2599.

[35] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems.* 9459–9474.

[36] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. 2020. Improving Approximate Nearest Neighbor Search Through Learned Adaptive Early Termination. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 2539–2554.

[37] Wuchao Li, Chao Feng, Defu Lian, Yuxin Xie, Haifeng Liu, Yong Ge, and Enhong Chen. 2023. Learning Balanced Tree Indexes for Large-Scale Vector Retrieval. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining.* 1353–1362.

[38] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data — Experiments, Analyses, and Improvement. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (2020), 1475–1488.

[39] Tianwei Lin, Wenqiao Zhang, Sijing Li, Yuqian Yuan, Binhe Yu, Haoyuan Li, Wanggui He, Hao Jiang, Mengze Li, Song xiaohui, Siliang Tang, Jun Xiao, Hui Lin, Yueting Zhuang, and Beng Chin Ooi. 2025. HealthGPT: A Medical Large Vision-Language Model for Unifying Comprehension and Generation via Heterogeneous Knowledge Adaptation. In *Proceedings of the 42nd International Conference on Machine Learning.*

[40] Zheyuan Liu, Cristian Rodriguez-Opazo, Damien Teney, and Stephen Gould. 2021. Image Retrieval on Real-Life Images with Pre-trained Vision-and-Language Models. In *2021 IEEE/CVF International Conference on Computer Vision.* 2105–2114.

[41] Zepu Lu, Defu Lian, Jin Zhang, Zaixi Zhang, Chao Feng, Hao Wang, and Enhong Chen. 2023. Differentiable Optimized Product Quantization and Beyond. In *Proceedings of the ACM Web Conference 2023.* 3353–3363.

[42] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836.

[43] Priyanka Nigam, Yiwei Song, Vijai Mohan, Vihan Lakshman, Weitian (Allen) Ding, Ankit Shingavi, Choon Hui Teo, Hao Gu, and Bing Yin. 2019. Semantic Product Search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining.* 2876–2885.

[44] Beng Chin Ooi, Shaofeng Cai, Gang Chen, Yanyan Shen, Kian-Lee Tan, Yuncheng Wu, Xiaokui Xiao, Naili Xing, Cong Yue, Lingze Zeng, Meihui Zhang, and Zhanhao Zhao. 2024. NeurDB: an AI-powered autonomous data system. *Sci. China Inf. Sci.* 67, 10, Article 200901 (2024), 10 pages.

[45] Ninh Pham and Tao Liu. 2024. Falconn++: A Locality-Sensitive Filtering Approach for Approximate Nearest Neighbor Search. In *Advances in Neural Information Processing Systems*. 31186–31198.

[46] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. In *Proceedings of the 38th International Conference on Machine Learning*. 8748–8763.

[47] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.

[48] Alexandre Sablayrolles, Matthijs Douze, Cordelia Schmid, and Hervé Jégou. 2019. Spreading Vectors for Similarity Search. In *7th International Conference on Learning Representations*.

[49] Philip Sun, David Simcha, Dave Dopson, Ruiqi Guo, and Sanjiv Kumar. 2023. SOAR: Improved Indexing for Approximate Nearest Neighbor Search. In *Advances in Neural Information Processing Systems*. 3189–3204.

[50] Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. 2021. BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

[51] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.

[52] Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. 2022. Text Embeddings by Weakly-Supervised Contrastive Pre-training. *CoRR* abs/2212.03533 (2022).

[53] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. *Proc. ACM Manag. Data* 2, 1, Article 14 (2024), 27 pages.

[54] Weaviate. 2024. Weaviate. Retrieved April 12, 2024 from https://weaviate.io/

[55] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. *Proc. VLDB Endow.* 13, 12 (2020), 3152–3165.

[56] Shitao Xiao, Zheng Liu, Weihao Han, Jianjin Zhang, Defu Lian, Yeyun Gong, Qi Chen, Fan Yang, Hao Sun, Yingxia Shao, and Xing Xie. 2022. Distill-VQ: Learning Retrieval Oriented Vector Quantization by Distilling Knowledge from Dense Embeddings. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 1513–1523.

[57] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, Peng Cheng, and Mao Yang. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 545–561.

[58] Jingtao Zhan, Jiaxin Mao, Yiqun Liu, Jiafeng Guo, Min Zhang, and Shaoping Ma. 2022. Learning Discrete Representations via Constrained Clustering for Effective and Efficient Dense Retrieval. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*. 1328–1336.

[59] Hailin Zhang, Yujing Wang, Qi Chen, Ruiheng Chang, Ting Zhang, Ziming Miao, Yingyan Hou, Yang Ding, Xupeng Miao, Haonan Wang, Bochen Pang, Yuefeng Zhan, Hao Sun, Weiwei Deng, Qi Zhang, Fan Yang, Xing Xie, Mao Yang, and Bin Cui. 2024. Model-Enhanced Vector Index. In *Advances in Neural Information Processing Systems*. 54903–54917.

[60] Pengcheng Zhang, Bin Yao, Chao Gao, Bin Wu, Xiao He, Feifei Li, Yuanfei Lu, Chaoqun Zhan, and Feilong Tang. 2022. Learning-Based Query Optimization for Multi-probe Approximate Nearest Neighbor Search. *The VLDB Journal* 32, 3 (2022), 623–645.

[61] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. 2023. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation*. 377–395.

[62] Yang Zhang, Fuli Feng, Chenxu Wang, Xiangnan He, Meng Wang, Yan Li, and Yongdong Zhang. 2020. How to Retrain Recommender System? A Sequential Meta-Learning Method. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 1479–1488.

[63] Zili Zhang, Chao Jin, Linpeng Tang, Xuanzhe Liu, and Xin Jin. 2023. Fast, Approximate Vector Queries on Very Large Unstructured Datasets. In *20th USENIX Symposium on Networked Systems Design and Implementation*. 995–1011.

[64] Xi Zhao, Yao Tian, Kai Huang, Bolong Zheng, and Xiaofang Zhou. 2023. Towards Efficient Index Construction and Approximate Nearest Neighbor Search in High-Dimensional Spaces. *Proc. VLDB Endow.* 16, 8 (2023), 1979–1991.