HAKES: Scalable Vector Database for Embedding Search Service

Guoyu Hu[1], Shaofeng Cai[1], Tien Tuan Anh Dinh[2], Zhongle Xie[3],
Cong Yue[1], Gang Chen[3], Beng Chin Ooi[1]
[1]National University of Singapore [2]Deakin University
[3] State Key Laboratory of Blockchain and Data Security, Zhejiang University
{guoyu.hu,shaofeng,yuecong,ooibc}@comp.nus.edu.sg,anh.dinh@deakin.edu.au,{xiezl,cg}@zju.edu.cn

ABSTRACT

Modern deep learning models capture the semantics of complex, unstructured data by transforming them into high-dimensional embedding vectors. Emerging applications, such as retrieval-augmented generation, use nearest neighbors search in the embedding vector space to find similar data. Existing vector databases provide indexes for efficient approximate nearest neighbor (ANN) searches, with graph-based indexes being the most popular due to their low latency and high recall in real-world datasets with high dimensionality. However, these indexes are costly to build and suffer from significant contention under concurrent read-write workloads. They also 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 a novel ANN index whose parameters are fine-tuned using a lightweight machine learning technique. We introduce early termination check to dynamically adapt the search process for each query. Next, we add support for writes while ensuring high search performance by decoupling the management of the learned parameters. Finally, we design *HAKES*, a distributed vector database that uses the new index and is based on a disaggregated architecture. We evaluate our index and system against 12 state-of-the-art indexes and three distributed vector databases, using high-dimension embedding datasets generated by deep learning models. The experimental results confirm that our index outperforms partitioning-based indexes and the recent graph-based indexes for high-recall searches. Furthermore, *HAKES* is scalable and achieves up to 16× higher throughputs than the baselines do.

PVLDB Reference Format:
Guoyu Hu, Shaofeng Cai, Tien Tuan Anh Dinh, Zhongle Xie, Cong Yue, Gang Chen, Beng Chin Ooi. *HAKES*: Scalable Vector Database for Embedding Search Service. PVLDB, 18(1): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:

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

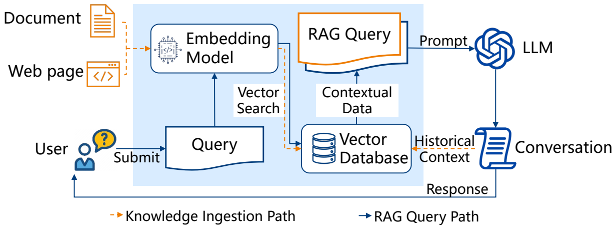


Figure 1: Vector database in retrieval augmented generation.

1 INTRODUCTION

High-dimension embedding vectors generated by deep learning models are becoming an important form of data representation for complex, unstructured data such as images [45, 61], audios [9], and texts [36, 70]. The models take semantically similar data points as inputs and generate vectors that are close to each other in the embedding space. As a result, the embedding vectors capture data semantics through their relative positions in a high-dimension space. A typical embedding vector has in the order of 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 [11, 45] , recommendation systems [41, 52] to retrieval-augmented generation (RAG) [7, 39] These applications require efficient, high quality search as well as support for database updates. Figure 1 shows an example of how a vector database is used in RAG applications. A user submits a query to RAG, which turns the query into a vector. In the next step, RAG performs nearest neighbor search to find semantically similar data stored in a vector database. It then augments the query with the data found in the previous step and sends the new query to an LLM. To improve future responses, RAG frequently updates the vector database with data representing new knowledge, such as new documents, web pages, and past user interactions.

Vector databases use indexes to support efficient nearest neighbor search. Since searching for exact nearest neighbors is too costly due to the curse of dimensionality [31], existing works on vector indexes focus on approximate nearest neighbor (ANN) search. The vast number of proposed ANN indexes can be classified as graph-based or partitioning-based indexes [19, 21, 32, 38, 43, 46, 49, 51, 66, 85]. These indexes are 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 such as RAG [4, 26, 38, 51]. We identify three limitations of vector databases built around the existing ANN indexes to support modern applications.

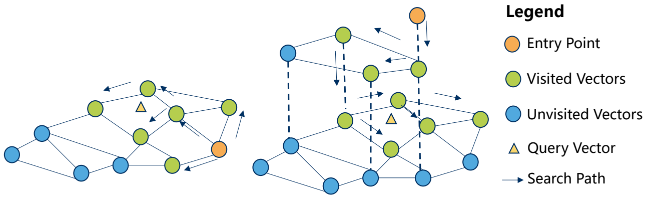
Proceedings of the VLDB Endowment, Vol. 18, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

The first limitation is *the computation overhead under high dimension spaces*. In particular, comparing a vector against its neighbors becomes more expensive with higher dimensions. Graph-based indexes [18, 19, 43, 51, 64] are also 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 [51, 74, 77, 85], or out-of-place using a separate data structure and performing periodic consolidation [17, 68, 75]. Graph indexes perform in-place updates, and require fine-grained locking over the neighborhood of the nodes on its traversal path [51, 68]. 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 [11, 25, 75]. They partition the data and build an independent graph index for each partition. However, we observe that to achieve high recall in high-dimension spaces, nearly all data partitions have to be searched. The large number of searches per query leads to low throughputs.

Our goal is to build a scalable vector database that achieves high throughput and high recall under concurrent read-write workloads. We present a new vector database *HAKES* that employs a novel index in a disaggregated architecture. Our index is based on a filter-and-refine scheme with a compressed partitioning-based index, enhanced with self-supervised learning. It addresses the first limitation discussed above by compressing the vectors in the filter stage to reduce the cost of vector comparison. The refine stage uses full-precision vectors to search a small set of candidates for high recall. We use a novel, lightweight machine-learning technique to improve the quality of candidate vectors. Specifically, this technique reduces similarity distribution distortion after dimension reduction and quantization, focusing on candidates close to the query vector. We introduce an additional optimization that determines if a search can terminate early in the filter stage based on the candidates already collected. Our index addresses the second limitation of prior works, i.e. support for concurrent read-write workloads, by decoupling the index parameters used for updates from those used during search. By ensuring the trained parameters do not affect the compression of new vectors, an update only locks the index when appending the compressed code to memory, which reduces read-write contention. *HAKES* addresses the scalability limitation of existing vector databases by exploiting the two-stage nature of our index to manage index and data separately in a disaggregated architecture. It distributes the memory and computation cost over multiple nodes, thereby achieving high throughput at scale.

In summary, we make the following contributions:

- We propose a novel index, *HAKES-Index*, that combines a compressed partitioning-based index with dimension 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-dimension 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.



(a) Graph index **(b) HNSW**

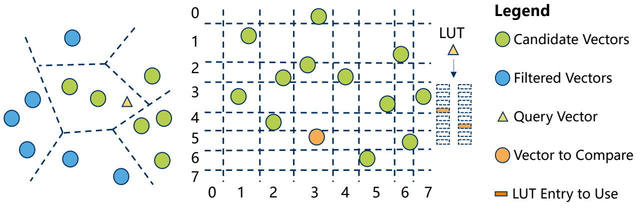
Figure 2: Graph-based ANN index.

The remainder of the paper is structured as follows. Section 2 provides the background on ANN search and the state-of-the-art ANN indexes. Section 3 and Section 4 describe the design of our index and the distributed vector database. Section 5 evaluates our designs against state-of-the-art indexes and systems. Section 6 reviews the related works, before Section 7 concludes.

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-dimension space is expensive due to the curse of dimensionality [31]. 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.

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 [19, 43, 51]. 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



(a) Index partitioning **(b) PQ**

Figure 3: Partitioning-based ANN index.

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 [18, 51, 64]. 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 [18, 19, 51, 85].

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) [2, 20, 38, 59, 86], clustering [12, 32, 50], quantization [6, 24, 34, 35, 54, 55, 69], or neural networks [10, 27, 42]. New vectors are added to the corresponding partition by computing its partition assignment. A search for vector $\mathbf{x}$ starts by finding the partitions closest to $\mathbf{x}$, 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 [12, 34] uses k-means to partition the vectors. Specifically, a sample set of vectors is used to determine the cluster centroids, and then vectors belonging to the 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, for million-scale datasets, high recalls can be achieved when N_c is in 1000s and *nprobe* is in 10s to 100s.

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 $\mathbf{C}^{\mathbf{PQ}} \in \mathbb{R}^{N_c \times d}$ is the concatenation of subspace centroids $\mathbf{C}^{\mathbf{PQ}}{}_j \in \mathbb{R}^{N_c \times d'}$, i.e., $\mathbf{C}^{\mathbf{PQ}} = [\mathbf{C}^{\mathbf{PQ}}{}_1, \mathbf{C}^{\mathbf{PQ}}{}_2, ..., \mathbf{C}^{\mathbf{PQ}}{}_m]$. 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 $p_j(\mathbf{v}) = \arg\min_i ||\mathbf{C}^{\mathbf{PQ}}{}_j[i] - \mathbf{v}_j||$ denotes the index of the closest centroid in the j^{th} subspace centroids $\mathbf{C}_{\mathbf{j}}^{\mathbf{(PQ)}}$. Let $q_j(\mathbf{v}) = \mathbf{C}^{\mathbf{PQ}}{}_j[p_j(\mathbf{v})]$ 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:

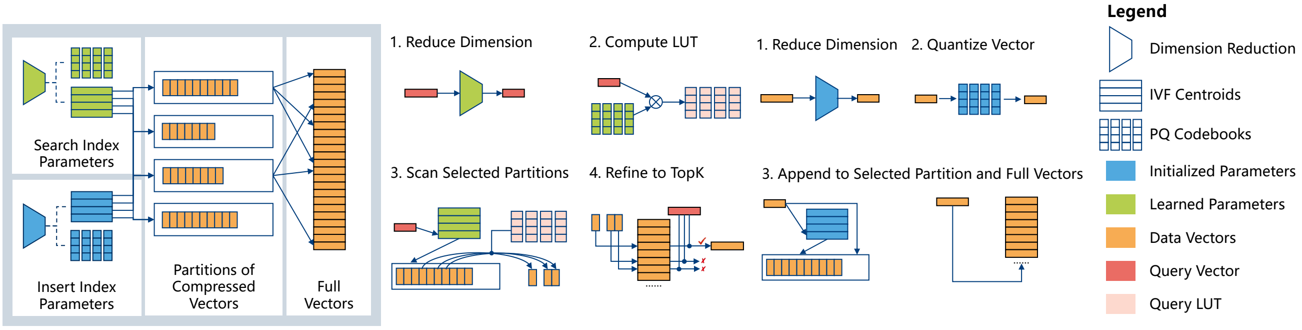
$$d(\mathbf{x}, \mathbf{v}) \approx d(\mathbf{x}, q(\mathbf{v})) = \sum_{j=1..m} d(\mathbf{x}_j, q_j(\mathbf{v})). \tag{1}$$

PQ enables efficient comparison of $\mathbf{x}$ against the candidate vectors. During a search, the query vector is split into m subvectors, each of which is compared against all the centroids in its corresponding subspace, $\mathbf{C}^{\mathbf{PQ}}{}_j$. The resulting similarity scores are stored in a lookup table, LUT $\in \mathbb{R}^{N_c \times m}$. Given Equation 1, the similarity between $\mathbf{x}$ and any vector $\mathbf{v}$ can then 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 yield a LUT small enough to fit in CPU caches, and optimized the quantized vector layout for efficient SIMD implementation thereby achieving significantly higher throughputs [3, 12, 26].

While PQ enables faster comparison between vectors, its quantization is inherently lossy, meaning that PQ indexes may have low recall [43]. Existing works address this limitation by reducing approximation error [26, 35], replacing k-means for generating codebooks [72], or modifying the subspace decomposition [6, 69]. However, these approaches increase either memory consumption, pre-processing overhead, or query complexity. In practice, quantization is often used together with a coarse-grained partitioning technique such as IVF to filter out a large number of vectors before applying quantization. Furthermore, an IVF partition stores vectors in a contiguous memory region, resulting in fast scanning of the quantization codes due to memory prefetching. Quantization can also be used with graph indexes to speed up comparison of the query vector with the graph vertices [1, 37, 64]. Specifically, during traversal, comparison against each vertex is based on the quantized vectors instead of the original ones.

3 HAKES-INDEX

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



(a) *HAKES-Index* **(b) Search** **(c) Insert**

Figure 4: *HAKES-Index* overview.

of the main components, as well as the search and insert workflow, in Section 3.1. The index building is described in Section 3.2. We present the detail of our self-supervised learning techniques to obtain the search index parameters in Section 3.3. We discuss the search optimizations of *HAKES-Index* in Section 3.4, followed by a discussion on the rationale of our index design in Section 3.5.

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 dimension-reduction module, IVF centroids, and a quantization codebook. The compressed vectors are partitioned by the IVF centroids and the compression involves dimension reduction followed by quantization guided by the codebook. The dimension 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, $\mathbf{C}^{\mathbf{IVF}}$, 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, $\mathbf{C}^{\mathbf{PQ}}$, are used to generate the quantized vector stored in the partitions and compute the Lookup table for search. Note that dimension reduction is placed at the front, which speeds up all subsequent computations.

For search query processing, there are four steps, as shown in Figure 4b. The first step reduces the dimensionality of the query vector from d to d_r, as marked in dark orange in the figure. Next, the output of the dimension reduction is used to compute the lookup table (LUT) with the learned quantization codebook. The third step evaluates the d_r-dimension query vector with the IVF centroids to select the closest partitions for scanning using the LUT. $k' > k$ candidate vectors are selected and the top k of them are obtained by comparing the query vector to their full vectors in the last step. 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 from the full search space. 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 dimension 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 partition of the full vectors.

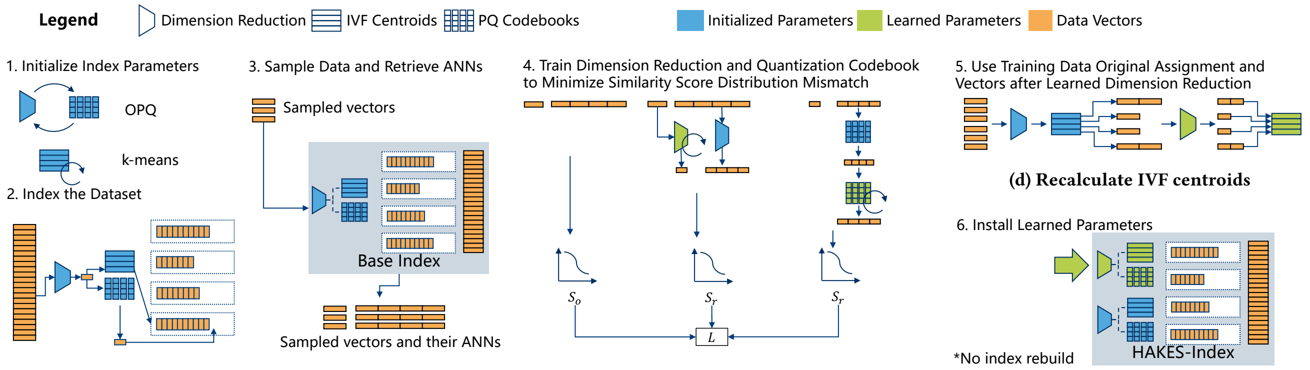
3.2 Index Construction

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 $\mathbf{C}^{\mathbf{PQ}}$, 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, $\mathbf{C}^{\mathbf{IVF}}$. Next, the training set is prepared by sampling a set of vectors and obtaining their neighbors with the base index, as shown 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 compression parameters, $\mathbf{A}$, $\mathbf{b}$ and $\mathbf{C}^{\mathbf{PQ}}$. The learned parameters are used for search 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 next subsection. After training, the IVF centroids $\mathbf{C}^{\mathbf{IVF}}$ are updated by recalculating the centroids with the learned $\mathbf{A}$ and $\mathbf{b}$ for the sampled vectors (Figure 5d). Finally, the newly learned $\mathbf{A}$, $\mathbf{b}$, $\mathbf{C}^{\mathbf{PQ}}$, and $\mathbf{C}^{\mathbf{IVF}}$, are installed in the index, as shown completed in 5e, serving subsequent search queries.

We note that the training process can run independently from the serving system. In practice, the index is first built with existing OPQ and IVF. As it serves requests, the system records samples, and the training process runs in the background. Once the training is finished, the new parameters can be used immediately for serving queries. In other words, *HAKES-Index* can be updated incrementally.

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 there are many true nearest neighbors in the


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.

set of k' candidate vectors. The *compression parameters* in *HAKES-Index* include $\mathbf{A}$ and $\mathbf{b}$ for dimensionality reduction, and $\mathbf{C}^{\mathbf{PQ}}$ for the PQ codebooks. The goal is to fine-tune compression parameters so that they capture the similarity of vectors in high-dimension spaces.

$\mathbf{A}$ and $\mathbf{C}^{\mathbf{PQ}}$ are initialized with OPQ, and the bias vector $\mathbf{b}$ is initialized with zero. We then jointly optimize $\mathbf{A}$, $\mathbf{b}$, and $\mathbf{C}^{\mathbf{PQ}}$. The objective is to minimize the mismatch of the similarity of vectors in the original d-dimensional space and that based on quantized vectors:

$$d(\mathbf{x},\mathbf{v}) \approx d(R(\mathbf{x}), q(R(\mathbf{v}))) \tag{2}$$

where $R(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$ is the dimensionality reduction for vector $\mathbf{x}$.

We only focus on the mismatch between query vectors with their close neighbors, which can be efficiently retrieved by running an ANN search on the base index. The training objective of our training process is defined based on the similarity score distributions of a sampled query vector $\mathbf{x}$ and its close neighbors ANN_x. Specifically, the similarity score distributions before and after the dimension reduction are:

$$S_{\mathbf{o},\mathbf{x}} = \text{softmax}([d(\mathbf{x},\mathbf{v_1}),\dots,d(\mathbf{x},\mathbf{v_K})]) \tag{3}$$

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

where the softmax function converts the similarity scores to a distribution, and $K = |ANN_x|$ is the number of retrieved close neighbors. While the distribution of the similarity scores after quantization follows Equation 2, the quantization process $q'(\cdot)$ is different for these close neighbors:

$$S_{q,\mathbf{x}} = \text{softmax}([d(R(\mathbf{x}), q'(R_{fixed}(\mathbf{v_1})),\dots]) \tag{5}$$

where $q'(\cdot)$ denotes the vector being quantized by the initialized codebook but its approximation uses the learned codebook and neighbor vectors are transformed by the fixed dimension reduction $R_{fixed}(\cdot)$.

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 3) and that in the vector space after dimension reduction (Equation 4). The other pair is between (Equation 3) and the distribution of similarity scores calculated between a query vector after dimension reduction and its quantized close neighbors (Equation 5). The overall training objective is as follows:

$$L = -\sum_{\mathbf{x}\in D_{sample}} S_o \log \frac{S_{r,\mathbf{x}}}{S_{o,\mathbf{x}}} - \lambda \sum_{\mathbf{x}\in D_{sample}} S_{o,\mathbf{x}} \log \frac{S_{q,\mathbf{x}}}{S_{o,\mathbf{x}}} \tag{6}$$

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 the parameters to minimize the loss defined in Equation 6 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 optimization is that it quantizes each dimension of the IVF centroids into INT8 representation. This allows using SIMD to evaluate 4× more dimensions in a single instruction. Although quantization can be lossy, the centroids are only used for partition assignment for a large number of partitions, which has a high tolerance for such representation errors. For example, in many real-world datasets, *nprobe* is often set to 1/10 partitions to reach high recall at 0.99, thus the partitions containing the true nearest neighbors are likely to be selected even if their centroids are slightly shifted.

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-dimension space, it is challenging to determine the *nprobe* based solely on the centroids. *HAKES-Index* introduces a heuristic condition for early stopping the partition scans based on the intermediate search results. The key idea is that the search process ranks the partitions by the distance of their centroids to the query, and it scans the partitions in order, so as the search 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 achieves efficiency by using IVF to filter out a large number of vectors, then using quantized vectors in the filter stage for computing vector similarity. It achieves high recall because the first stage returns vectors close to the query vector, and the second phase refines these candidates using the original, non-compressed vectors. We observe in our experimental evaluation that deep embeddings vectors can be aggressively compressed with d_r as small as $1/4$ or $1/8$ of the original dimension d, and with 4-bit PQ with $m = 2$. Our empirical results also demonstrate that the training process can improve the quality of the vectors returned by the filter stage, i.e., the candidate vector set is likely to contain true nearest neighbors. The training process focuses on the close vectors, since in the search process distant vectors in IVF are filtered out and never evaluated. That is significantly different from prior works on product quantization and dimensionality reduction, which have different optimization goals. For instance, PQ [34] and its variants [6, 24, 26, 48, 69] focus on minimizing the reconstruction error, i.e., $d(\mathbf{v}, q(\mathbf{v}))$, while the principle component analysis (PCA) algorithm aims to preserve variance after dimensionality reduction. Moreoever, most existing works use the latest updated codebook for assignment during every training iteration. However, we note that codebook update requires expensive re-encoding of the vectors when applying the trained parameters to serving queries. Therefore, we fix PQ code assignment during training, such that the training outputs are applied to the index only once. We also regularize the training to mitigate the distribution shift between vectors before and after dimensionality reduction.

To support adding new vectors for search in *HAKES-Index*, 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. The reason is closely related to how the lightweight supervised training is designed. As discussed in Section 3.3, we use the prebuilt base index and fix PQ code assignment for efficient retrieval and training, where all the data vectors are processed only once using the original set of parameters. Consequently, the newly added vectors should also follow this processing procedure, and use the same set of parameters for consistency. Empirical observations also confirm that using the learned parameters for inserting new vectors leads to recall degradation, as will be shown in Section 5. Therefore, *HAKES-Index* decouples the management of the two sets of parameters, for efficient training and effective serving respectively.

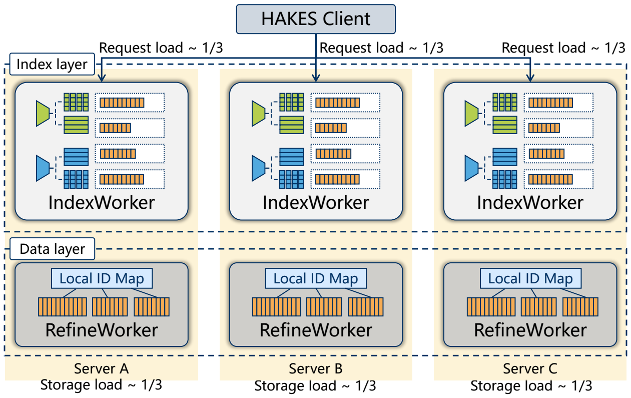


Figure 6: *HAKES* architecture.

The aggressive compression employed by *HAKES-Index* not only significantly speeds up the filtering stage, but also reduces the memory overhead of maintaining the index. We now analyze the memory cost of *HAKES-Index*. The original vectors take $(N \cdot 4 \cdot d)$ bytes, the dimension reduction matrices and the bias vector take $(2 \cdot 4 \cdot d \cdot d_r + 4 \cdot d_r)$ bytes, the IVF centroids $(N_c \cdot 4 \cdot d_r + N_c \cdot d_r)$ bytes, the quantization codebooks $(2 \cdot 2^4 \cdot 4 \cdot d_r)$ bytes for the 4-bit quantization and the compressed vectors $(N \cdot (1/2) \cdot (d_r/m))$ bytes. It can be seen that the original, non-compressed vectors dominate the overall memory cost.

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 [83]). After model training, rebuild the index is necessary.

4 THE *HAKES* DISTRIBUTED VECTORDB

In this section, we present our distributed vector database, named *HAKES*. It exploits the two-stage design of *HAKES-Index* to achieve scalability in a disaggregated architecture. We first discuss its architecture and how it differs to state-of-the-art distributed vector databases. We then describes the system's components, its query workflow, and its consistency.

4.1 Overview

A search query using *HAKES-Index* goes through two stages, namely filter and refine. We observe that these stages access different types and numbers of vectors, leading to distinct memory and compute requirements. 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 similarity computation over a large number of vectors.

We design an architecture that exploits the observation above to decouple the two stages. In particular, it separates the filter-stage index from the full-precision, original vectors used only in the refine stage. This disaggregation allows the computation of each stage to be carried out independently. In particular, there are two sets of nodes, the IndexWorkers and the RefineWorkers. The former are responsible for the filter stage, managing the replicated compressed vectors. The latter perform the refine stage, storing shards of the original vectors. Figure 6 shows an example in which a physical node runs both an IndexWorker and a RefineWorker. However, we stress that 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.

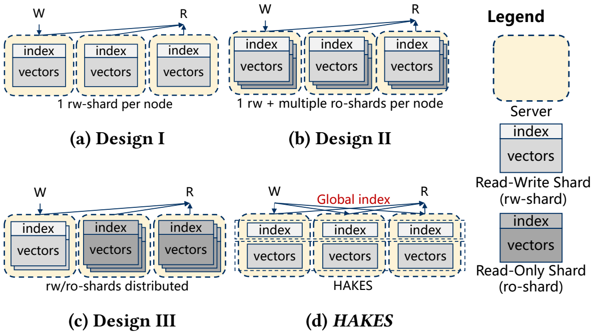
Discussion We discuss how *HAKES*'s architecture is different to that of existing distributed vector databases. Figure 7 compares four architectures with distinct shard layouts and communication for read and write. In the first architecture (Figure 7a), adopted by [11, 58, 74], each server hosts a single read-write shard and maintains an index. A read request merges the 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 [37], each node maintains one read-write shard and multiple read-only shards to reduce the read-write contention. Figure 7c third architecture extends the first two, by employing multiple read-write shards and multiple read-only shards. It is adopted by [25, 68], and supports scaling out of read or of write by adding servers for the required type of shards. We note that in these three architecture, 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 node, since the filter stage *HAKES-Index* has a small size and supports efficient update.

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.

Search and insert 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.


W
R
index
vectors
1 rw-shard per node
1 rw + multiple ro-shards per node
Legend
Server
Read-Write Shard (rw-shard)
Read-Only Shard (ro-shard)
rw/ro-shards distributed
Global index
HAKES

(a) Design I **(b) Design II**

(c) Design III **(d) *HAKES***

Figure 7: Different architectures of distributed vector databases.

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.

Consistency and failure recovery *HAKES* does not guarantee strong consistency, which is acceptable because applications relying on vector search can tolerate that [68, 75]. 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 first benchmark *HAKES-Index* against state-of-the-art ANN indexes to validate our analysis and study the effectiveness of our index design in Section 5.3. We then provide an in-depth analysis of the self-supervised learning scheme that generates the compression parameters in Section 5.4. Last but not least, we evaluate *HAKES* against state-of-the-art distributed vector databases, comparing their throughput at high-recall regions in Section 5.5.

5.1 Implementation

We implement *HAKES-Index* by extending the FAISS library [12]. IndexWorker and RefineWorker are implemented on top of the index *HAKES-Index*, and they are accessible via an HTTP server implemented using libuv [44] and llhttp [53]. The index extension and serving system take approximately 7000 LoC in C++. The index training is implemented in around 1000 LoC in Python, using Pytorch@1.12.1. The *HAKES* client is implemented in Python in around 500 LoC.

Table 1: High-dimension 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

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 million scale, and we them for index benchmarking.

- DPR-768[15] is generated by the Dense Passage Retrieval (DPR) context encoder model [36] on text records sampled from the Sphere web corpus dataset [28, 60].
- OPENAI-1536 [16] is generated by OpenAI's embedding service [57] on DBpedia text data [67].
- MBNET-1024: is generated by pretrained MobileNet [30] on one million ImageNet data [62].
- RSNET-2048: is generated by pretrained ResNet [29] on 1 million ImageNet data.
- GIST-960: is a widely in the literature for benchmarking ANN indexes [4, 22, 43]. 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: is the embedding model as DPR-768 but on 10 million Sphere text records.
- E5-1024-10m: is generated with E5-large text model [70] on 10 million Sphere text records.

All datasets above are normalized. We use the inner product for the similarity metric, and recall10@10 for search quality. The ground-truth nearest neighbors for the queries are generated by brute-force search over the entire datasets.

Training setup Index training is conducted on a 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 $nprobe = 1/10$ of IVF partitions 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. The distributed experiments are run in a cluster of servers with the same hardware specification above.

Index baselines We select 12 state-of-the-art in-memory ANN index baselines, covering both IVF partitioning-based and graph-based indexes.

- 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 [3, 12]. The RF denotes a refine stage
- OPQIVFPQ_RF uses OPQ [24] to learn a rotation matrix that minimizes the PQ reconstruction error. An OPQ matrix $\in \mathbb{R}^{d \times d_r}$ can reduce the vector dimension to d_r as implemented in faiss.
- HNSW [51] is the index used in almost all vector databases.
- ELPIS [5] partitions the dataset [13, 73] and maintains an HNSW graph index for each partition, representative for maintaining multiple subgraph indexes.
- LSH-APG [85] leverages LSH to identify close entry points on its graph index to reduce the search path length.
- ScaNN [26], SOAR [65], and RaBitQ [23] are recent proposed quantization scheme used with partitioning-based index.
- Falconn++ [59] and LCCS [38] are recent, state-of-the-art LSH indexes.
- LVQ [1] is a state-of-the-art graph index with quantization and it is optimized for hardware acceleration and prefetching.

We do not compare against recent indexes that are optimized for secondary storage [8, 71, 77] which report lower performance than in-memory indexes.

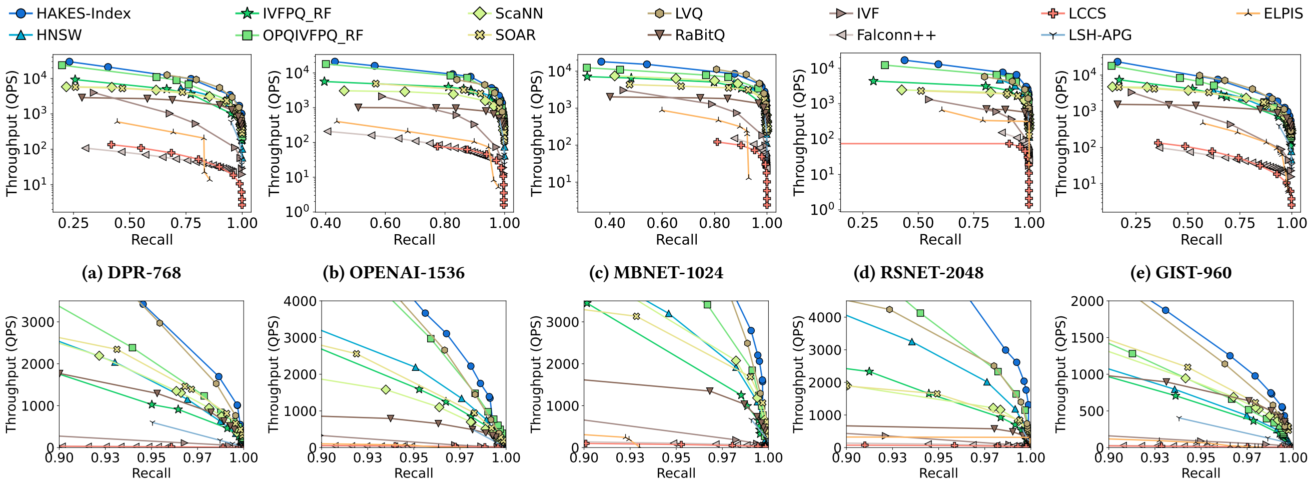
Distributed vector database baselines We select three popular distributed vector databases that employ in-memory ANN indexes. They cover the three architectures described in Section 4.1. We set up the systems according to the recommendations from their respective official documentation.

- Weaviate [74] 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 [37] adds support for vector search recently [17] 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 [14], a graph index that only searches quantized vectors, similar to DiskANN [64].
- Milvus [25, 68] 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 employ the base index, that is, without parameter training or optimizations.

5.3 Index Benchmarking and Analysis

For each index, we first build it with the recommended configurations in the corresponding paper and 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 trade-off. Due to space constraints, we provide the complete



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

(f) DPR-768 (recall ≥ 0.9) **(g) OPENAI-1536 (recall ≥ 0.9)** **(h) MBNET-1024 (recall ≥ 0.9)** **(i) RSNET-2048 (recall ≥ 0.9)** **(j) GIST-960 (recall ≥ 0.9)**

Figure 8: Throughput vs. recall for sequential reads.

set of explored and the selected configurations for all indexes in the extended version.

Sequential read workload Figure 8 compares the throughput-recall frontiers of the 13 indexes for the full recall range, then for high recall (above 90%). The results demonstrate the cost of each index to reach different recalls, where the top right curve represents the best trade-off. Across the different datasets, *HAKES-Index* achieves the state-of-the-art throughput-recall trade-off. 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 our analysis that with a refine stage, deep embeddings can be compressed with quantization and dimension reduction for filtering, thereby achieving good performance trade-off. Across the deep embedding datasets, *HAKES-Index* reduces the vector dimension to 1/4 or 1/8 of the original dimension.

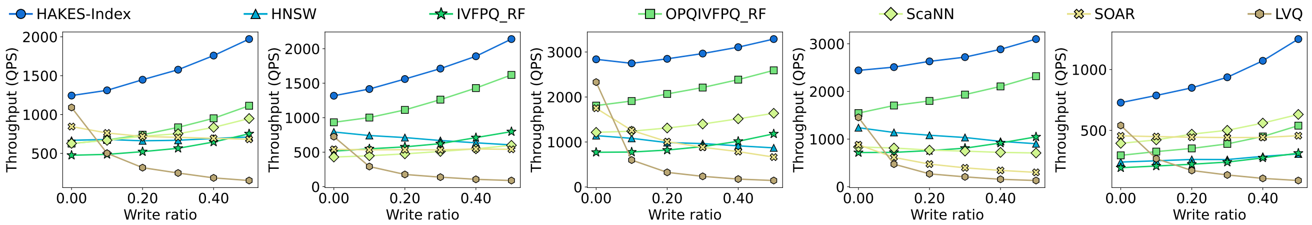
Recent quantization-based indexes, namely ScaNN, SOAR and RaBitQ, show mixed results compared to IVFPQ_RF which uses standard PQ and fast scan implementation. ScaNN improves the quantization for inner product approximation; SOAR aims to reduce the correlation of multiple IVF partitions assignment for one vector; and RaBitQ uses LSH to generate binary code representation and early terminate a search with error bounds. 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-dimension deep embedding.

The performance of Falconn++ and LCCS rank below IVF, confirming that LSH-based indexes are less effective in filtering vectors than the data-dependent approaches in high-dimension space [4, 23, 43, 85]. Among graph-based indexes, 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-dimension space. The gap between HNSW and ELPIS shows that sharding a global graph index into smaller subgraphs degrades the overall performance. We analyze the impact on distributed vector databases in Section 5.5.

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.

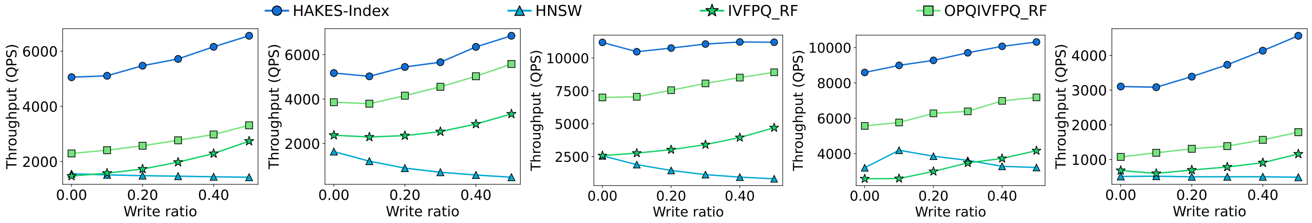
We further evaluate the indexes supporting concurrent read-write workloads. The HNSW implementation in hnswlib supports concurrent read-write with fine-grained locking on the graph nodes, and using our extension on faiss, IVFPQ_RF, and OPQIVFPQ_RF also support partition locking as our index does. We 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 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 We observe that 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. HNSW maintains vertices and edges 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,



(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).

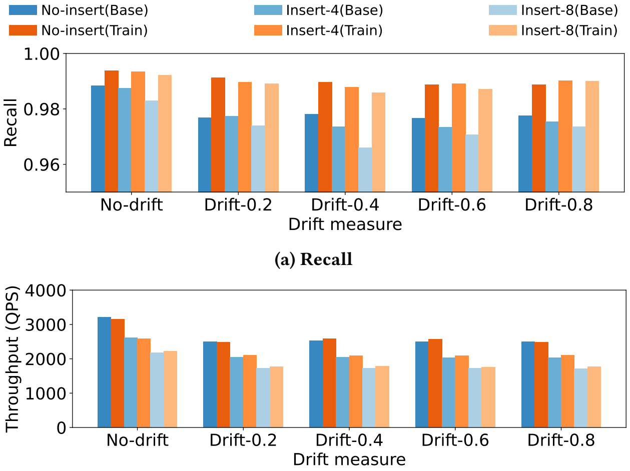
where OPQIVFPQ_RF consumes 5.86 GiB. *HAKES-Index* consumes 5.86 GiB, as the additional set query index parameters are small.

5.4 *HAKES-Index* Analysis

Recall improvement Table 2 reports the recalls for different search configurations, for million scale and 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 2 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 when the candidate set is small. 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.

For 1-million scale datasets (MBNET-1024 and RSNET-2048), there are cases where the learned parameters lead to slightly worse recalls. The reason is that the training process prioritizes loss of vectors in close proximity at the expense of approximation error for more distant vectors.

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 dimension 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. Training takes 34.9s and 45.6s, repsectively. When deployed on a cluster, the time to insert vectors and prepare the training set neighbors can be reduced linearly with the number of nodes. In comparison, constructing the HNSW graph takes 5736.4s and 9713.21s on DPR-768-10m and E5-1024-10m 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 learned parameters can be seamlessly integrated once available, without rebuilding the index.



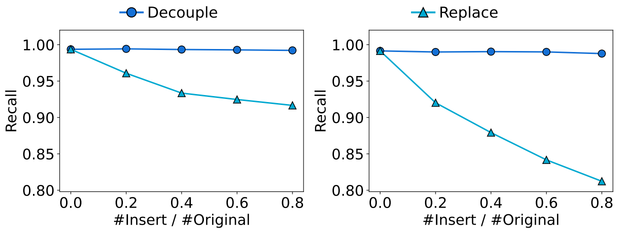
(a) Recall

(b) Throughput

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

Table 2: 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**
IVF *nprobe* (total 1024)		10			50			100			200		
k'/k		10	50	200	10	50	200	10	50	200	10	50	200
MBNET-1024	Base	0.852	**0.886**	**0.886**	0.930	**0.980**	**0.981**	0.938	0.991	**0.993**	0.940	0.995	**0.997**
	Learned	**0.879**	0.884	0.884	**0.972**	**0.980**	0.980	**0.983**	**0.992**	0.992	**0.986**	**0.996**	0.996
RSNET-2048	Base	0.911	**0.948**	**0.949**	0.947	0.992	**0.993**	0.951	0.997	**0.998**	0.952	0.998	**0.999**
	Learned	**0.943**	**0.948**	0.948	**0.986**	**0.993**	**0.993**	**0.980**	**0.998**	**0.998**	**0.992**	**0.999**	**0.999**



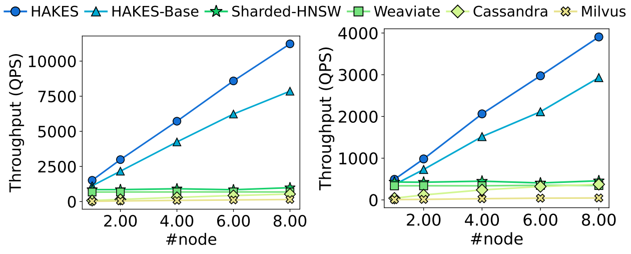
(a) MBNET-1024

(b) RSNET-2048

Figure 12: Impact of separating index parameters for read and write.

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 ≥ 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 by building the index for 1 million vectors, selecting *nprobe* and k'/k for recall ≥ 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 used for query from the parameters used 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 newly added data can have significantly lower approximated similarity than those neighbors in the original dataset.



(a) DPR-768-10m

(b) E5-1024-10m

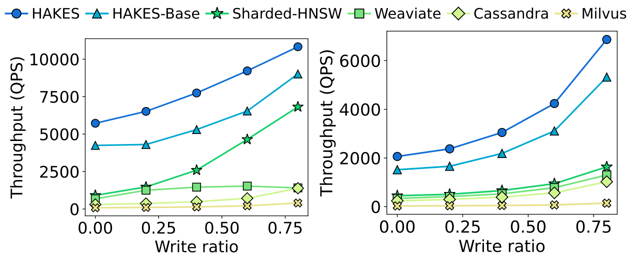
Figure 13: Scalability under read-only workload.

Ablation study Due to space constraints, we include further ablation studies and results in the extended version, including the impact of number of nearest neighbor used for training, different search optimizations and the heuristic for early termination.

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, it can reach the recall of 0.98. Specifically, the system uses quantized graph-index to return a larger number of candidate vectors, which are then processed by a refine stage to achieve recall10@10 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 settings used for other systems. The QueryNodes are evenly distributed among the physical nodes in a Kubernetes cluster. We use the multiple distributed clients to saturate the systems, then report the peak throughputs.

Scaling with the number of nodes Figure 13 compares the systems' throughputs with varying numbers of nodes. It can be seen



(a) DPR-768-10m **(b) E5-1024-10m**

Figure 14: Throughputs under concurrent read-write workload

that *HAKES* and *HAKES*-Base scale linearly because the load of both the filtering and refinement stages are distributed evenly across the nodes. The filtering 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 the same trend is observed. 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 (over 20 for Sphere-768-10m and over 30 for Sphere-1024-10m), all of which need to be searched, meaning that the throughputs are low. 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 14 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

Concurrent read-write workloads in vector databases Recent works recognize the need to continuously adding new data to vector databases [75, 77]. At the index level, HNSW [51], LVQ [1], and FreshDiskANN [63] rebuilds the graph locally. We showed in experiments that this process is expensive. SPFresh [77] considers partitioning-based indexes and proposes a scheme to keep the partition size small. This technique can be integrated to *HAKES-Index* to avoid index rebuilding when the dataset grows considerably. At the system level, sharding is employed to reduce the impact of inserting new vectors [11, 25]. *HAKES-Index* leverages the partitioning-based index for low read-write contention, and introduces learned compression to achieve high throughputs and recall.

Improving index throughput and latency Vector indexes can be improved by exploiting characteristics of the queries and the immediate search results [22, 32, 40, 81, 82, 84]. ADSampling [22] transforms the query vector and uses incrementally more dimensions when evaluating the similarity as needed. However, this approach is not friendly to hardware acceleration that processes vectors in batches. Auncel, iDistance and VBASE require access to exact similarity scores to determine if a search can be stopped. These approaches are not suitable to the filter-stage in *HAKES-Index* where there are significant approximation errors in similarity computation. LEQAT [81] and [40] involve a predictive model that incurs significant overhead in serving a search request. The model also requires training on ground truth kNN distribution for a large number of queries, which is costly to obtain. In contrast , *HAKES-Index* only collects statistics that are readily available during search. Exploiting hardware acceleration is a common technique for improving index performance [1, 12, 23, 26, 33, 56]. Hardware features such as CPU vector instructions, GPU and FPGA, enable parallel processing over a batch of vectors. *HAKES-Index* leverages SIMD and 4-bit PQ fast scan. Its vector compression, scanning of quantization codes can be performed in parallel with predictive access patterns, thus *HAKES-Index* is highly amenable to hardware acceleration.

Learned compact representation and partitioning. Recent works from the information retrieval community propose to jointly train embedding models and product quantization codebooks to generate compact representations. The learning objective is to capture the semantic similarity between data vectors [76, 78–80]. However, they focus on retrieving the most relevant vectors, i.e., they optimize for Recall@K, NDCG@K, and MRR@K, and ignore performance metrics such as latency and throughput. Other learning-based solutions for improving coarse-grained partitioning either incur high overhead to for serving a query [48], or require costly training processes before the index can be used for queries [10, 27, 47]. *HAKES-Index* makes no assumptions about the vector generation process, and it avoids costly index rebuilding after training.

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.

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 (jul 2023), 3433–3446. https://doi.org/10.14778/3611479.3611537

[2] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and Optimal LSH for Angular Distance. In *Advances in Neural Information Processing Systems*, C Cortes, N Lawrence, D Lee, M Sugiyama, and R Garnett (Eds.), Vol. 28. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2015/file/2823f4797102ce1a1aec05359cc16dd9-Paper.pdf

[3] 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 (dec 2015), 288–299. https://doi.org/10.14778/2856318.2856324

[4] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374. https://doi.org/10.1016/j.is.2019.02.006

[5] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. ELPIS: Graph-Based Similarity Search for Scalable Data Science. *Proc. VLDB Endow.* 16, 6 (feb 2023), 1548–1559. https://doi.org/10.14778/3583140.3583166

[6] Artem Babenko and Victor Lempitsky. 2014. Additive Quantization for Extreme Vector Compression. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*. 931–938. https://doi.org/10.1109/CVPR.2014.124

[7] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. 2022. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*. PMLR, 2206–2240.

[8] 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 Neighbor Search. In *35th Conference on Neural Information Processing Systems (NeurIPS 2021)*.

[9] Soham Deshmukh, Benjamin Elizalde, Rita Singh, and Huaming Wang. 2023. Pengi: An audio language model for audio tasks. *Advances in Neural Information Processing Systems* 36 (2023), 18090–18108.

[10] Yihe Dong, Piotr Indyk, Ilya Razenshteyn, and Tal Wagner. 2020. Learning Space Partitions for Nearest Neighbor Search. (2020).

[11] 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 (dec 2021), 850–858. https://doi.org/10.14778/3503585.3503594

[12] 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). https://doi.org/10.48550/ARXIV.2401.08281 arXiv:2401.08281

[13] Karima Echihabi, Panagiota Fatourou, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2022. Hercules against data series similarity search. *Proc. VLDB Endow.* 15, 10 (jun 2022), 2005–2018. https://doi.org/10.14778/3547305.3547308

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

[15] Hugging Face. 2024. DPR. Retrieved April 12, 2024 from https://huggingface.co/docs/transformers/model_doc/dpr

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

[17] 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

[18] 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. https://doi.org/10.1109/TPAMI.2021.3067706

[19] 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 (jan 2019), 461–474. https://doi.org/10.14778/3303753.3303754

[20] 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 (KDD '15)*. Association for Computing Machinery, New York, NY, USA, 349–358. https://doi.org/10.1145/2783258.2783284

[21] Jinyang Gao, Hosagrahar Visvesvaraya Jagadish, Wei Lu, and Beng Chin Ooi. 2014. DSH: data sensitive hashing for high-dimensional k-nnsearch. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1127–1138. https://doi.org/10.1145/2588555.2588565

[22] 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 (jun 2023), 27 pages. https://doi.org/10.1145/3589282

[23] 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 (May 2024), 27 pages. https://doi.org/10.1145/3654970

[24] 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. https://doi.org/10.1109/TPAMI.2013.240

[25] 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 (aug 2022), 3548–3561. https://doi.org/10.14778/3554821.3554843

[26] 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 *International Conference on Machine Learning*. https://arxiv.org/abs/1908.10396

[27] 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 (KDD '22)*. Association for Computing Machinery, New York, NY, USA, 486–495. https://doi.org/10.1145/3534678.3539414

[28] Zain Hasan. 2022. The Sphere Dataset in Weaviate. Retrieved April 12, 2024 from https://weaviate.io/blog/sphere-dataset-in-weaviate

[29] 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 (CVPR)*. 770–778.

[30] 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).

[31] 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* (Dallas, Texas, USA) *(STOC '98)*. Association for Computing Machinery, New York, NY, USA, 604–613. https://doi.org/10.1145/276698.276876

[32] 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. https://doi.org/10.1145/1071610.1071612

[33] Wenqi Jiang, Shigang Li, Yu Zhu, Johannes De Fine Licht, Zhenhao He, Runbin Shi, Cedric Renggli, Shuai Zhang, Theodoros Rekatsinas, Torsten Hoefler, and Gustavo Alonso. 2023. Co-design Hardware and Algorithm for Vector Search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 87, 15 pages. https://doi.org/10.1145/3581784.3607045

[34] Herve 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. https://doi.org/10.1109/TPAMI.2010.57

[35] 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 (ICASSP)*. 861–864. https://doi.org/10.1109/ICASSP.2011.5946540

[36] 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 (EMNLP)*, Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 6769–6781. https://doi.org/10.18653/v1/2020.emnlp-main.550

[37] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.* 44, 2 (apr 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[38] 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* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2589–2599. https://doi.org/10.1145/3318464.3389778

[39] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[40] 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 (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2539–2554. https://doi.org/10.1145/3318464.3380600

[41] Sen Li, Fuyu Lv, Taiwei Jin, Guli Lin, Keping Yang, Xiaoyi Zeng, Xiao-Ming Wu, and Qianli Ma. 2021. Embedding-based Product Retrieval in Taobao Search. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (Virtual Event, Singapore) *(KDD '21)*. Association for Computing Machinery, New York, NY, USA, 3181–3189. https://doi.org/10.1145/3447548.3467101

[42] 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 (KDD '23)*. Association for Computing Machinery, New York, NY, USA, 1353–1362. https://doi.org/10.1145/3580305.3599406

[43] 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. https://doi.org/10.1109/TKDE.2019.2909204

[44] libuv. 2023. Asynchronous I/O made simple. Retrieved April 12, 2024 from https://libuv.org/

[45] 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 *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2125–2134.

[46] Kejing Lu, Mineichi Kudo, Chuan Xiao, and Yoshiharu Ishikawa. 2021. HVS: hierarchical graph structure based on voronoi diagrams for solving approximate nearest neighbor search. *Proc. VLDB Endow.* 15, 2 (oct 2021), 246–258. https://doi.org/10.14778/3489496.3489506

[47] Zepu Lu, Jin Chen, Defu Lian, ZAIXI ZHANG, Yong Ge, and Enhong Chen. 2023. Knowledge Distillation for High Dimensional Search Index. In *Advances in Neural Information Processing Systems*, A Oh, T Neumann, A Globerson, K Saenko, M Hardt, and S Levine (Eds.), Vol. 36. Curran Associates, Inc., 33403–33419. https://proceedings.neurips.cc/paper_files/paper/2023/file/6a15378acabd1aef017ec79a3ed744d2-Paper-Conference.pdf

[48] 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 (WWW '23)*. Association for Computing Machinery, New York, NY, USA, 3353–3363. https://doi.org/10.1145/3543507.3583482

[49] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. VLDB Endowment, 950–961.

[50] Mikko I Malinen and Pasi Fränti. 2014. Balanced K-Means for Clustering. In *Structural, Syntactic, and Statistical Pattern Recognition*, Pasi Fränti, Gavin Brown, Marco Loog, Francisco Escolano, and Marcello Pelillo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 32–41.

[51] 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. https://doi.org/10.1109/TPAMI.2018.2889473

[52] Vijai Mohan, Yiwei Song, Priyanka Nigam, Choon Hui Teo, Weitian Ding, Vihan Lakshman, Ankit Shingavi, Hao Gu, and Bing Yin. 2019. Semantic product search. In *KDD 2019*. https://www.amazon.science/publications/semantic-product-search

[53] nodejs. 2024. llhttp. https://llhttp.org/

[54] Haechan Noh, Taeho Kim, and Jae-Pil Heo. 2021. Product Quantizer Aware Inverted Index for Scalable Nearest Neighbor Search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 12210–12218.

[55] Mohammad Norouzi and David J. Fleet. 2013. Cartesian K-Means. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*. 3017–3024. https://doi.org/10.1109/CVPR.2013.388

[56] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. 2023. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. arXiv:2308.15136 [cs.DS]

[57] OpenAI. 2022. New and improved embedding model. Retrieved April 12, 2024 from https://openai.com/blog/new-and-improved-embedding-model

[58] OpenAI. 2024. Qdrant - Vector Database. Retrieved April 12, 2024 from https://qdrant.tech/

[59] Ninh Pham and Tao Liu. 2024. Falconn++: a locality-sensitive filtering approach for approximate nearest neighbor search. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 2261, 13 pages.

[60] Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Dmytro Okhonko, Samuel Broscheit, Gautier Izacard, Patrick Lewis, Barlas Oguz, Edouard Grave, Wen-tau Yih, and Sebastian Riedel. 2021. The Web Is Your Oyster - Knowledge-Intensive NLP against a Very Large Web Corpus. *CoRR* abs/2112.09924 (2021). arXiv:2112.09924 https://arxiv.org/abs/2112.09924

[61] 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, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research)*, Marina Meila and Tong Zhang (Eds.), Vol. 139. PMLR, 8748–8763. http://proceedings.mlr.press/v139/radford21a.html

[62] 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 (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[63] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2021. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. *CoRR* abs/2105.09613 (2021). arXiv:2105.09613 https://arxiv.org/abs/2105.09613

[64] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. 2019. *DiskANN: fast accurate billion-point nearest neighbor search on a single node.*

[65] Philip Sun, David Simcha, Dave Dopson, Ruiqi Guo, and Sanjiv Kumar. 2023. SOAR: Improved Indexing for Approximate Nearest Neighbor Search. In *Thirty-seventh Conference on Neural Information Processing Systems*.

[66] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2009. Quality and efficiency in high dimensional nearest neighbor search. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 563–576. https://doi.org/10.1145/1559845.1559905

[67] 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)*. https://openreview.net/forum?id=wCu6T5xFjeJ

[68] 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 (SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2614–2627. https://doi.org/10.1145/3448016.3457550

[69] Jingdong Wang and Ting Zhang. 2019. Composite Quantization. *IEEE Trans. Pattern Anal. Mach. Intell.* 41, 6 (jun 2019), 1308–1322. https://doi.org/10.1109/TPAMI.2018.2835468

[70] 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). https://doi.org/10.48550/ARXIV.2212.03533 arXiv:2212.03533

[71] 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 V2mod014 (mar 2024), 27 pages. https://doi.org/10.1145/3639269

[72] Runhui Wang and Dong Deng. 2020. DeltaPQ: lossless product quantization code compression for high dimensional similarity search. *Proc. VLDB Endow.* 13, 13 (sep 2020), 3603–3616. https://doi.org/10.14778/3424573.3424580

[73] Yang Wang, Peng Wang, Jian Pei, Wei Wang, and Sheng Huang. 2013. A data-adaptive and dynamic segmentation index for whole matching on time series. *Proc. VLDB Endow.* 6, 10 (aug 2013), 793–804. https://doi.org/10.14778/2536206.2536208

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

[75] 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 (aug 2020), 3152–3165. https://doi.org/10.14778/3415478.3415541

[76] 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 *(SIGIR '22)*. Association for Computing Machinery, New York, NY, USA, 1513–1523. https://doi.org/10.1145/3477495.3531799

[77] 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 (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 545–561. https://doi.org/10.1145/3600006.3613166

[78] 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 (WSDM '22). Association for Computing Machinery, 1328–1336. https://doi.org/10.1145/3488560.3498443

[79] 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 *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 2396, 15 pages.

[80] Peitian Zhang, Zheng Liu, Shitao Xiao, Zhicheng Dou, and Jing Yao. 2023. Hybrid Inverted Index Is a Robust Accelerator for Dense Retrieval. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Singapore, 1877–1888. https://doi.org/10.18653/v1/2023.emnlp-main.116

[81] 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 (sep 2022), 623–645. https://doi.org/10.1007/s00778-022-00762-0

[82] 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 (OSDI 23)*. USENIX Association, Boston, MA, 377–395. https://www.usenix.org/conference/osdi23/presentation/zhang-qianxi

[83] 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* (Virtual Event, China) *(SIGIR '20)*. Association for Computing Machinery, New York, NY, USA, 1479–1488. https://doi.org/10.1145/3397271.3401167

[84] 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 (NSDI 23)*. USENIX Association, Boston, MA, 995–1011. https://www.usenix.org/conference/nsdi23/presentation/zhang-zili

[85] 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 (apr 2023), 1979–1991. https://doi.org/10.14778/3594512.3594527

[86] Yuxin Zheng, Qi Guo, Anthony K.H. Tung, and Sai Wu. 2016. LazyLSH: Approximate Nearest Neighbor Search for Multiple Distance Functions with a Single Index. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 2023–2037. https://doi.org/10.1145/2882903.2882930