Accelerating Approximate Nearest Neighbor Search in Hierarchical Graphs: Efficient Level Navigation with Shortcuts

Zengyang Gong
Hong Kong University of Science and Technology
zgongae@cse.ust.hk

Yuxiang Zeng
State Key Laboratory of Complex & Critical Software Environment, Beihang University
yxzeng@buaa.edu.cn

Lei Chen
Hong Kong University of Science and Technology (Guangzhou) & FYTRI Hong Kong University of Science and Technology
leichen@cse.ust.hk

ABSTRACT

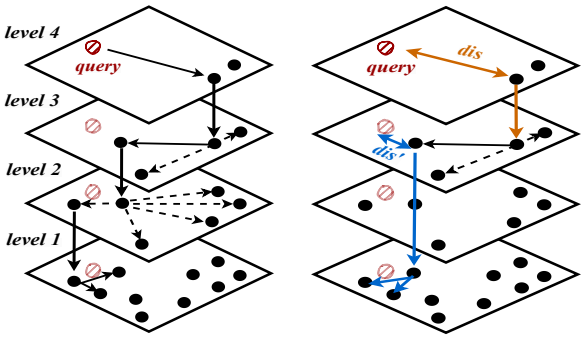
Approximate Nearest Neighbor (ANN) search is a foundational yet computationally demanding query in vector databases, critical for applications such as information retrieval and generative AI inference. Hierarchical graph-based methods have attracted significant attention due to their promising query performances compared to other indexes for ANN search. However, these methods still face efficiency bottlenecks because they rely on exhaustive and level-by-level traversals within hierarchical graphs. This paper introduces SHG, a novel hierarchical graph-based index that enhances search efficiency by bypassing intermediate and redundant levels. Specifically, SHG leverages a hierarchical vector compression method to reduce the time spent on distance computations, and employs a new data structure called shortcuts to determine the number of intermediate levels that can be safely skipped. Extensive experiments demonstrate that our solution achieves 1.5–1.8× speedup compared to state-of-the-art methods. Meanwhile, our method significantly improves the robustness of ANN search, boosting recall by up to 20% for certain queries on benchmark datasets.

PVLDB Reference Format:
Zengyang Gong, Yuxiang Zeng, and Lei Chen. Accelerating Approximate Nearest Neighbor Search in Hierarchical Graphs: Efficient Level Navigation with Shortcuts. PVLDB, 18(10): 3518 - 3530, 2025.
doi:10.14778/3748191.3748212

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

1 INTRODUCTION

With the emergence of large language models (LLMs), the vector databases [40, 41] garner significant attention for their abilities to support efficient data storage [25, 53], information retrieval [44, 45], and retrieval-augmented generation [9, 32]. A fundamental query type within vector databases is the Approximate k-Nearest Neighbor (k-ANN) search. k-ANN has become a focal point of research interest due to the challenge posed by high-dimensional vector data, often referred to as the "curse of dimensionality" [28, 48]. Indexing is a key solution to this challenge and often provides an effective balance between search efficiency and the recall of query answers [8, 34, 52, 56]. Among existing ANN indexes, graph-based indexes demonstrate superior performance compared to other in-memory indexes, including tree-based, hash-based, and quantization-based methods [5, 47, 50]. Accordingly, this work is dedicated to an in-depth exploration and enhancement of graph-based indexes.



(a) Overview of HNSW [36] (b) Motivation of our method

Figure 1: Motivation example: search k-ANN ($k = 3$) of the query vector (edges at each level are omitted for clarity)

Graph-based ANN Indexes. Graph-based ANN methods build *proximity graphs* where each vector is represented as a node, and edges between nodes signify neighbor relationships. By leveraging these relationships, search algorithms iteratively expand the necessary neighborhoods of currently traversed nodes, enabling rapid query processing. Recent advancements in hierarchical graph-based indexes have shown promising improvements in query performance [18, 34, 36, 38]. Notably, Hierarchical Navigable Small World (HNSW) [36] stands out as the most prevalent hierarchical graph-based index in vector databases [40, 41], widely recognized for its efficiency and scalability. The following example illustrates the main structure of the HNSW index.

Example 1 (Hierarchical Graph: HNSW [36])**.** Fig. 1a illustrates an example of HNSW with four levels of proximity graphs (*i.e.*, hierarchical graphs). At each level, edges connect nearby vectors, though they are omitted for clarity. If a vector appears at level x, it also appears at all lower levels down to level 1. Consequently, the proximity graph at level 1 encompasses the entire dataset. When searching kNN of the query vector marked in red, the process begins at level 4 to identify the locally nearest vector, which serves as the

Proceedings of the VLDB Endowment, Vol. 18, No. 10 ISSN 2150-8097.
doi:10.14778/3748191.3748212

entry point for level 3. This procedure is repeated, moving down through levels until reaching level 1. Finally, k nearest vectors are identified by traversing the graph at level 1 using a heap.

Our Observation and Motivation. A hierarchical graph-based index typically traverses all proximity graphs from the top-most level to the bottom-most level. This procedure involves numerous distance computations over high-dimensional vector data, which can become a significant bottleneck for search efficiency. However, in practice, the nearest neighbor of the query vector at a higher level x often remains as the nearest neighbor at lower levels (say until level $y < x$), as long as their distance is sufficiently short. In such cases, we can skip the levels above y and directly navigate to level y, thereby reducing unnecessary computations. To illustrate this motivation, consider the following example.

Example 2 (Shortcut-enabled HNSW)**.** Fig. 1b shows the key insight from our observations in HNSW. When searching at level 4, let dis denote the distance between the query vector and its nearest neighbor at this level. The relatively long distance suggests the need for a more granular search. Upon moving to level 3, the nearest distance significantly decreases to $dis' \ll dis$, suggesting that we have approached the area close enough to the query vector. Now, imagine an additional data structure, the shortcut (marked by blue arrows across levels), which would allow us to skip level 2 and directly navigate to level 1. This would eliminate the time spent traversing level 2, thereby improving search efficiency.

Main Idea of Our Solution. Inspired by above observations, a Shortcut-enabled Hierarchical Graph (SHG) index is proposed to facilitate efficient level navigation within hierarchical graphs for k-ANN search. Specifically, we first propose a **hierarchical vector compression method** for reducing data dimensions across levels. This method not only quickly approximates vector distances using compressed coordinates, but also ensures a constant relative error between adjacent levels. Then, we develop a novel data structure called **learned shortcut**, which effectively infers the number of levels that can be safely skipped. The shortcut is represented by piecewise linear models trained on the samples generated from the nodes in hierarchical graphs using k-th nearest neighbor density estimation. Finally, we propose efficient algorithms for constructing our index SHG and performing k-ANN search using SHG.

Contribution. The main contributions are listed as follows:

- We are the first to identify that (i) certain level navigations in hierarchical graphs are redundant and degrade search efficiency, and (ii) introducing shortcuts can eliminate these unnecessary steps, thereby reducing computational cost.
- We propose a hierarchical vector compression method that accelerates distance computations while providing approximation guarantees and facilitating shortcut construction.
- We design a new index SHG enhanced with shortcuts. Under this structure, we propose an efficient construction method and a scalable k-ANN search algorithm.
- Comprehensive experiments on 8 datasets show that our solution accelerates k-ANN search by 1.5–1.8× compared to the state-of-the-art methods [34, 36, 56]. Moreover, our approach exhibits superior robustness in recall performance. In our test, our SHG achieves up to 20% higher recall in queries where existing indexes struggle, with an average improvement from 72% to 81% across all queries.

Table 1: Summary of major notations

Notation	Description
$\mathcal{D} \subseteq \mathbb{R}^d$	A dataset $\mathcal{D}$ of d-dimensional vectors
$o = [v_1, v_2, \cdots, v_d]$	A d-dimensional vector o
$dis(o, o')$	Exact distance between o and o'
$level_l(o)$	Compressed representation of o at level l
$dis(level_l(o), level_l(o'))$	Approximate distance between o and o' at level l
$S = \{(\widehat{dis}_i, h_i) \mid i\}$	Training samples for construct learned shortcut
$f(\widehat{dis})$	Infer skipped levels via learned shortcut
q, k, A	Query vector q, integer k, and query answer A

Roadmap. The rest of this paper is organized as follows. Sec. 2 defines the query and analyzes limitations of prior hierarchical graph based indexes. In Sec. 3, we introduce our hierarchical vector compression method and analyze its relative error. Our index SHG and ANN search algorithm are detailed in Sec. 4. Finally, we conduct experiments, review related work, and conclude in Sec. 5-7.

2 PRELIMINARIES

This section defines the query (Sec. 2.1), presents the limitations of existing methods (Sec. 2.2), and highlights the research opportunities that inspire our solution (Sec. 2.3). Table 1 summarizes the frequently used notations in the rest of this paper.

2.1 Problem Definition

Vectors effectively represent structured and unstructured data (*e.g.*, images and words) via embedding models. These high-dimensional data representations enable semantic search and power modern AI systems like LLMs. A formal definition of vector data is as follows.

Definition 1 (Vector Data)**.** *A vector data object o is represented by a point within the d-dimensional real coordinate space $\mathbb{R}^d$, i.e., $o = [v_1, v_2, \cdots, v_d]$, where v_i denotes the i-th coordinate of the vector.*

We use $\mathcal{D}$ to define a set of d-dimensional vector data, where the data cardinality is denoted by $n = |\mathcal{D}|$. The distance between any two vectors $o, o' \in \mathcal{D}$ is denoted as $dis(o, o')$. In practice, $dis(\cdot, \cdot)$ can be instantiated as Euclidean distance or Lp-norm.

Definition 2 (k-ANN Search)**.** *Given a query vector $q \in \mathbb{R}^d$ and a vector dataset $\mathcal{D}$, k-Approximate Nearest Neighbor (k-ANN) search is to identify a subset $A \subseteq \mathcal{D}$ containing k vectors that are close to the query vector q, with the objective of maximizing the recall as follows:*

$$Recall = \frac{|A \cap A^*|}{k} \tag{1}$$

where $A^* \subseteq \mathcal{D}$ denotes the exact k-nearest neighbors of q. Formally, for all $o \in A^*$ and for all $o' \notin A^*$, $dis(o, q) \leq dis(o', q)$ holds.

There are also previous studies that focus on the (c, k)-ANN search problem [28, 52], where the parameter c is explicitly employed to control search quality, such that $dis(q, o_i) \leq c \cdot dis(q, o_i^*)$ with o_i^* being the exact i-th nearest neighbor of q. However, recent studies and modern vector databases emphasize recall-based approaches more prominently [34, 36, 47, 50].

Algorithm 1: k-ANN search using HNSW [36]

Input: Dataset $\mathcal{D}$, a query vector q, and parameter m_L
Output: Approximate k-nearest neighbors A to q
// **Index Construction Phase:**
1 **foreach** *Vector* $o \in \mathcal{D}$ **do**
2 $RanLevel \leftarrow \lfloor -\ln(\text{uniform}(0,1)) \cdot m_L \rfloor$;
3 **if** *RanLevel is larger than current top level* **then**
4 $ep \leftarrow o$;
5 **for** $l \leftarrow RanLevel$ *to* $BaseLevel$ **do**
6 Select neighbors at level l with $dis(o, \cdot)$;
7 Build the NSW graph with neighbors at level l;
// k**-ANN Search Phase:**
8 $ep \leftarrow$ the entry point at the top level;
9 **for** *level* $l \leftarrow TopLevel$ *to* $(BaseLevel + 1)$ **do**
10 $ep \leftarrow$ find the nearest neighbor to q using exact distance $dis(q, \cdot)$ at level l;
11 Query result $A \leftarrow$ search k-ANN of ep at the base level;
12 **return** A;

Remark. Indexing has become a cornerstone of state-of-the-art methods for recall-oriented k-ANN search. Existing indexes can be classified into tree-based, LSH-based, graph-based, and other types. Among these, graph-based indexes have demonstrated superior query performance [5, 40, 50]. Consequently, this work primarily focuses on enhancing the performance of graph-based indexes.

2.2 Understanding Hierarchical Graph-based Indexes for k-ANN Search

Hierarchical graph-based indexes have emerged as a promising solution for k-ANN search, offering a balance between high efficiency and excellent recall. By leveraging their hierarchical structures, this kinds of graph-based indexes enable rapid vector searches and are commonly adopted in vector databases.

Mainstream Index: HNSW. Hierarchical Navigable Small World (HNSW) [36] is one of the most prevalent indexes for query optimization in modern vector databases (*e.g.*, Milvus [2], AnalyticDB-B [51], SingleStore-V [10], and Pinecone [4]). The basic structure of HNSW is illustrated in Fig. 1a, where vectors are connected in hierarchical graphs. This unique structure has led to substantial improvements in query efficiency. For clarity, we briefly introduce the construction and search routine of HNSW.

HNSW-Enabled Search Routine. Algo. 1 shows the routine of HNSW-based methods [36], with lines 1-7 outlining the index construction phase and lines 8-12 outlining the k-ANN search phase. During the *construction phase*, each vector $o \in \mathcal{D}$ is assigned a level (*i.e.*, *RanLevel*) that follows an exponentially decaying probability. This allows the vector o to exist at all levels from *RanLevel* down to *BaseLevel*. At each level, the neighbors of o are identified by calculating the exact distances to other vectors within that level (line 6). By connecting o to its neighbors, a proximity graph, called Navigable Small World (NSW) [35], is constructed in line 7. In the *k-ANN search phase*, the procedure begins at the top level, where a data point ep is selected as the entry point. The algorithm then traverses each level sequentially, identifying the vector with the nearest distance to ep at each level and designating this vector as the new entry point ep for the subsequent level. This process continues until reaching the level just above *BaseLevel* (lines 9-10). Finally, the approximate query result A is determined by a local search within the graph at the bottom level *BaseLevel*.

Efficiency Bottleneck of HNSW-like Indexes. Despite their popularity, hierarchical graph-based indexes, such as HNSW-like indexes [34, 36, 38], still face efficiency bottlenecks during ANN search. These bottlenecks primarily stem from two critical issues:

- **Intensive Distance Computations.** For high-dimensional vector data, distance computations across all dimensions are usually the predominant factor affecting time efficiency.
- **Exhaustive Level-by-Level Navigation.** Prior methods require a brute-force hierarchical navigation, scanning all levels from the top-most level down to the lowest level. At each level, a proximity graph needs to be explored, which limits overall scalability.

Both issues contribute substantially to the computational overhead. Thus, addressing these limitations is imperative for enhancing the performance and scalability of this kind of indexes.

2.3 Overview of Our Solution

To address the above efficiency bottlenecks, we propose a novel hierarchical graph-based index, referred to as SHG.

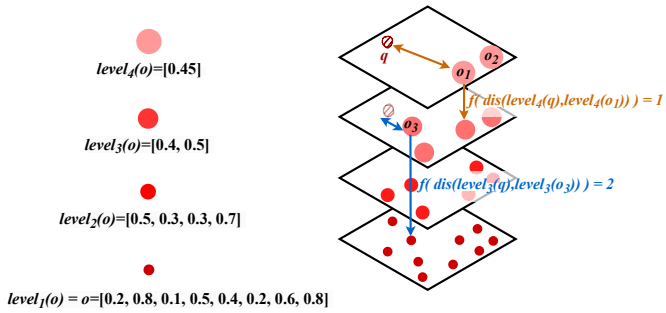
Key Ideas of Our Index. Fig. 2 illustrates the core ideas underlying our solution. The key innovations in our index SHG are twofold:

- **Approximate Distance Computations with Compressed Vectors (Sec. 3)**. To reduce the computational cost of (exact) distance computations, we propose an approximation method that progressively compresses vectors across hierarchical levels. This method enables efficient computations of distances using compressed vectors while maintaining a proper approximation error (see Theorem 1).
- **Shortcut Structure Between Adjacent Levels (Sec. 4)**. To avoid exhaustive traversal through all levels, we devise a specialized data structure called **shortcut**. A shortcut connects the entry points between different hierarchical levels. By using the distance between the query vector and the entry point at the top level, a shortcut can skip unnecessary levels and directly access a suitable bottom level. Consequently, this approach saves a substantial portion of runtime by bypassing certain levels and avoids traversing proximity graphs at those levels.

Together, these techniques enhance the efficiency and recall of k-ANN search, as will be validated by experiments in Sec. 5.

3 HIERARCHICAL VECTOR COMPRESSION FOR APPROXIMATE DISTANCES

This section first introduces a new vector compression strategy that progressively compresses vector representations as the level increases. Then, it provides an analysis of the relative error between any two vectors across levels within the hierarchical structure.



(a) Hierarchical vector compression ($d = 8$) **(b) Shortcut enabled index**

Figure 2: Overview of our solution

3.1 Hierarchical Vector Compression via Progressive Mean Aggregation

A d-dimensional vector o is represented as $[v_1, v_2, \cdots, v_d]$, where v_i is the coordinate value of the i-th dimension. For simplicity, we assume $d = \eta^{\mathcal{L}-1}$ for an integer $\mathcal{L}$ (with default value of η set to 2). We use $level_l(o)$ to denote the compressed representation of this vector's coordinates at the level l, where $1 \le l \le \mathcal{L}$.

Main Idea. Eq. (2) presents the key steps of our hierarchical compression method by using progressive mean aggregation.

$$level_l(o) = \begin{cases} \left[v_1, v_2, \cdots, v_d\right] & \text{if } l = 1 \\ \left[mean^o_{l,1}, mean^o_{l,2}, \cdots, mean^o_{l,\eta^{\mathcal{L}-l}}\right] & \text{if } l > 1 \end{cases} \tag{2}$$

Specifically, at the *base level* (*i.e.*, $l = 1$), vectors maintain their original uncompressed representation: $level_1(o) = [v_1, v_2, \cdots, v_d]$. For *upper levels* (*i.e.*, $l > 1$), we use $mean^o_{l,j}$ to denote the average of the coordinate values within the j-th segment at level $l-1$, which covers dimensions from $\eta(j-1)+1$ to $\eta(j-1)+\eta$. Accordingly, $mean^o_{l,j}$ can be progressively calculated as follows:

$$mean^o_{l,j} = \begin{cases} v_j & \text{if } l = 1 \\ \frac{1}{\eta}\sum_{k=1}^{\eta} mean^o_{l-1,k+\eta(j-1)} & \text{if } l \ge 2 \end{cases} \tag{3}$$

Example 3. As depicted in Fig. 2a, consider a vector o with $d = 8$ dimensions and $\eta = 2$. At level 1, o is represented in its original form as $level_1(o) = [0.2, 0.8, 0.1, 0.5, 0.4, 0.2, 0.6, 0.8]$. At level 2, the dimensionality is reduced to 4. Based on Eq. (3), we have $mean^o_{2,1} = (0.2+0.8)/2 = 0.5$, $mean^o_{2,2} = (0.1+0.5)/2 = 0.3$, $mean^o_{2,3} = (0.4+0.2)/2 = 0.3$, and $mean^o_{2,4} = (0.6+0.8)/2 = 0.7$. According to Eq. (2), the compressed vector at level 2 is $level_2(o) = [0.5, 0.3, 0.3, 0.7]$. This process continues until reaching the level $\mathcal{L} = 4$.

Remark. In practice, the dimensionality of vector data is typically in the form of $2^{\mathcal{L}-1}$. If this condition is not met, we can handle the discrepancy by padding the vector with 0.

3.2 Distance Approximation over Compressed Vectors and Relative Error Analysis

This subsection introduces how to compute the approximate distance for compressed vectors at the same level. It also analyzes the relative error under commonly-used distance functions (*e.g.*, Euclidean distance and Lp-norm).

Approximate Distance Computation. For any two vectors o and o' at level l, we can use their compressed coordinates to approximate their true distance, *i.e.*, $dis(level_l(o), level_l(o'))$. Intuitively, this approach reduces the computational cost of distance computations by using lower-dimensional representations.

Relative Error Analysis in Euclidean Space. Our approximate distance computation method guarantees a constant relative error between consecutive hierarchical levels, as proved in Theorem 1.

Theorem 1. *Given any two d-dimensional vectors $o, o' \in \mathcal{D}$ under Euclidean space, for any level $l \in [1, \mathcal{L})$, the approximate distances between them at levels l and $l+1$ satisfy the following inequality:*

$$dis(level_{l+1}(o), level_{l+1}(o')) \le \frac{1}{\sqrt{\eta}} \cdot dis(level_l(o), level_l(o')) \tag{4}$$

Proof. The proof relies on Lemma 1 (which will be proved later) and the vector compression scheme described in Eq. (2) and Eq. (3).

Under Euclidean space, we can derive the squared distance between any two compressed vectors at level $l+1$ as:

$$dis\left(level_{l+1}(o), level_{l+1}(o')\right)^2 = \sum_{j=1}^{\eta^{\mathcal{L}-l}} \left|mean^o_{l+1,j} - mean^{o'}_{l+1,j}\right|^2 \tag{5}$$

Based on Eq. (3), we know:

$$mean^o_{l+1,j} = \frac{1}{\eta}\sum_{k=1}^{\eta} mean^o_{l,k+\eta(j-1)} \tag{6}$$

By substituting the expressions for $mean^o_{l+1,j}$ and $mean^{o'}_{l+1,j}$ from Eq. (6) into the right-hand side (RHS) of Eq. (5), we have:

$$\left|mean^o_{l+1,j} - mean^{o'}_{l+1,j}\right|^2 = \left|\frac{1}{\eta}\sum_{k=1}^{\eta}\left(mean^o_{l,k+\eta(j-1)} - mean^{o'}_{l,k+\eta(j-1)}\right)\right|^2$$

By applying Lemma 1, the RHS of this equation is bounded by:

$$\left|mean^o_{l+1,j} - mean^{o'}_{l+1,j}\right|^2 \le \frac{1}{\eta}\sum_{k=1}^{\eta}\left|mean^o_{l,k+\eta(j-1)} - mean^{o'}_{l,k+\eta(j-1)}\right|^2$$

Consequently, the RHS of Eq. (5) is bounded by:

$$\frac{1}{\eta}\sum_{j=1}^{\eta^{\mathcal{L}-l}}\sum_{k=1}^{\eta}\left|mean^o_{l,k+\eta(j-1)} - mean^{o'}_{l,k+\eta(j-1)}\right|^2 \tag{7}$$

Under Euclidean space, the squared distance between the compressed vectors at level l is given by:

$$dis\left(level_l(o), level_l(o')\right)^2 = \sum_{j=1}^{\eta^{\mathcal{L}-l+1}} \left|mean^o_{l,j} - mean^{o'}_{l,j}\right|^2 \tag{8}$$

By grouping and summing each block of η consecutive terms of $|mean^{o}_{l,j} - mean^{o'}_{l,j}|^2$, the RHS of Eq. (8) can be rewritten as:

$$\sum_{j=1}^{\eta^{\mathcal{L}-l}} \sum_{k=1}^{\eta} \left| mean^{o}_{l,k+\eta(j-1)} - mean^{o'}_{l,k+\eta(j-1)} \right|^2 \tag{9}$$

By substituting Eq. (9) into the upper bound in Eq. (7), we obtain:

$$dis\left(level_{l+1}(o), level_{l+1}(o')\right)^2 \leq \frac{1}{\eta} \cdot dis\left(level_l(o), level_l(o')\right)^2$$

This completes the proof. □

Lemma 1. Using our vector compression method in Euclidean space, the following inequality holds for the corresponding coordinates of any two vectors $o, o' \in \mathcal{D}$ at any level $l \in [1, \mathcal{L}]$:

$$\left| \frac{1}{\eta} \sum_{k=1}^{\eta} \left(mean^{o}_{l,k+\eta j} - mean^{o'}_{l,k+\eta j} \right) \right|^2 \leq \frac{1}{\eta} \sum_{k=1}^{\eta} \left| mean^{o}_{l,k+\eta j} - mean^{o'}_{l,k+\eta j} \right|^2$$

Proof. The proof relies on the finite form of Jensen's inequality [13], which is presented as follows:

$$F(w_1 x_1 + w_2 x_2 + \cdots + w_n x_n) \leq \sum_{k=1}^{n} w_k F(x_k) \tag{10}$$

where $F(\cdot)$ is a convex function, and $w_1, \cdots, w_n$ are non-negative real values such that $\sum_{k=1}^{n} w_k = 1$. By substituting $F(x) = |x|^2$, $w_1 = w_2 = \cdots = w_n = \frac{1}{\eta}$, and $n = \eta$, Eq. (11) simplifies into:

$$\left| \frac{1}{\eta} \cdot \sum_{k=1}^{\eta} x_k \right|^2 \leq \frac{1}{\eta} \cdot \sum_{k=1}^{\eta} |x_k|^2 \tag{11}$$

Finally, substituting x_k with $mean^{o}_{l,k+\eta j} - mean^{o'}_{l,k+\eta j}$ in Eq. (11) yields the inequality presented in this lemma. □

Relative Error Analysis in Lp-norm. In Theorem 2, we extend the relative error from Euclidean space to the Lp-norm. Due to page limitations, please refer to our full paper [24] for the detailed proof.

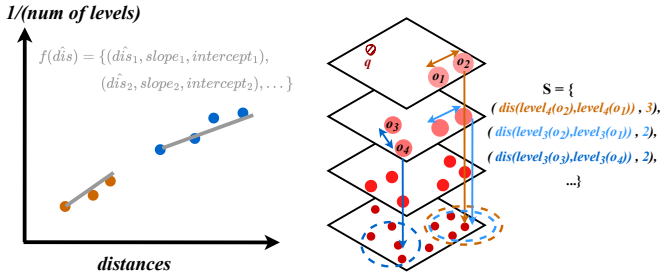
Theorem 2. *Given any two d-dimensional vectors $o, o' \in \mathcal{D}$ in Lp-norm ($p \geq 1$), for any level $l \in [1, \mathcal{L})$, the approximate distances between them at levels l and $l+1$ satisfy the following inequality:*

$$dis(level_{l+1}(o), level_{l+1}(o')) \leq \eta^{-1/p} \cdot dis(level_l(o), level_l(o'))$$

Remark. Theorems 1 and 2 indicate that the relative error between distances at consecutive levels is bounded by a constant factor of $\eta^{-1/p}$ ($\eta = 2$ in our implementation). In practice, the cumulative error $(\eta^{-1/p})^{l-1}$ usually remains bounded, since the number of levels $\mathcal{L}$ is typically small (*e.g.*, 3-5) in real datasets (see Sec. 5). As exact distances are maintained at level 1, *distances become progressively more accurate when navigating from upper to bottom levels.*

Why We Need Progressively Accurate Distances. The deliberate design of such distance patterns serves two purposes:

(i) Balancing Recall and Efficiency. At level 1, computing distances involves all dimensions, since the final answer is determined by traversing the graph at this level. As the level increases, the vector data becomes sparser, allowing for greater tolerance of distance errors. Thus, lower-dimensional representations are used to compress vector data at higher levels, thereby enhancing efficiency.



(a) Learned Shortcut (b) Learned shortcut construction

Figure 3: Shortcut construction

(ii) Facilitating Shortcut Construction. Our index allows for skipping certain levels by shortcuts. The distance between the entry vector and the query vector will determine the number of skipped levels. Intuitively, a fixed entry vector should bypass to the same target level from different starting levels, thereby skipping varying numbers of levels. Using different approximation scales across levels, shortcuts effectively capture this property (see Sec. 5.4).

4 SHORTCUT-ENABLED k-ANN INDEX

This section introduces our index SHG for k-ANN search. We define the shortcut in Sec. 4.1, present the construction method in Sec. 4.2, propose the search algorithm in Sec. 4.3, and provide a complexity analysis in Sec. 4.4. Finally, we discuss index maintenance in Sec. 4.5.

4.1 Key Component of Our Index: Shortcut

The shortcut is a critical component of our index designed to enhance the search efficiency. Its *primary role* is to infer the number of levels that can be skipped when traversing hierarchical graphs. This allows the shortcut to serve as an auxiliary data structure that complements ANN indexes and significantly improves the query efficiency. In this section, we use the state-of-the-art hierarchical graph-based index, HNSW [36], to introduce our solution.

Shortcut. The concept of shortcuts is formally defined as follows:

Definition 3 (Shortcut). *Given a query vector q and an entry vector ep in the hierarchical graph at a certain level l, the shortcut leverages the (approximate) distance $\widehat{dis}$ between q and ep to infer the number of levels, denoted by $f(\widehat{dis})$, that can be skipped from l. The shortcut also guarantees that ep remains the entry vector for the hierarchical graph at the target level with high probability.*

As shown in Fig. 2b, the entry vector at level l is the nearest neighbor to the query vector q in the proximity graph at level $l-1$. Then, the entry vector serves as the initial traversal point at level l. Intuitively, the choice of entry vector is critical for both recall and efficiency. Thus, inferring the number of skipped levels necessitates a solution that is both *cautious and accurate*. Meanwhile, the shortcut should be also *space-efficient*.

Learned Shortcut. For precise navigation across levels with minimal memory cost, we draw inspirations from learned index structures [30], which utilize machine learning models to fit data distributions. Prior studies [14, 16, 17, 33, 43] demonstrate that linear model based learned indexes are more lightweight and efficient compared

to traditional data structures. Thus, we also adopt *piecewise linear models* to learn the number of levels that can be skipped, since it provides a worst-case error bound under various distributions [14, 16, 17]. A *learned shortcut* is defined as follows:

Definition 4 (Learned Shortcut)**.** *Given a list of distance-level tuples* $S = \{(\widehat{dis}_0, h_0), (\widehat{dis}_1, h_1), \cdots\}$ *as the training data, where* $\widehat{dis}_i$ *is the approximate distance between two compressed vectors, and* h_i *is the corresponding number of levels that can be safely skipped. Let* X *denote the distances and* Y *denote the levels. The mapping* $X \rightarrow Y$ *is fitted using a set of piecewise linear functions:* $f(\widehat{dis}) = \{(\widehat{dis}_1, slope_1, intercept_1), (\widehat{dis}_2, slope_2, intercept_2), \cdots\}$. *Each triple* $(\widehat{dis}_i, slope_i, intercept_i)$ *fits a certain range of tuples through the function:* $f(\widehat{dis}) = \widehat{dis} \times slope_i + intercept_i$. *The learned shortcut is represented by the piecewise linear model* $f(\widehat{dis})$.

Example 4. As shown in Fig. 3a, the orange and blue points represent the distance-level tuples in the training data set S. The x-axis indicates distances, while the y-axis shows the reciprocal of the number of levels that can be skipped. The learned shortcut $f(\widehat{dis})$ comprises two piecewise linear functions. The parameters of the first function are trained using yellow data points, while the second function is fitted to the blue data points. Since each linear function only requires a limited number of parameters, the learned shortcut remains space-efficient even for large-scale datasets.

Remark. For non-linear models like neural networks, inference time and memory usage are significantly higher than linear models for learning shortcuts (see our full paper [24] for evaluations).

4.2 Index Construction

Technical Challenge. Although piecewise linear models have achieved significant success in building learned indexes [17, 30], we still face two major challenges to build learned shortcuts:

(i) Determining Safe Skip Levels. There is currently no established guideline for deciding the number of levels that can be safely skipped. Prior work on ANN search has not considered building shortcuts in hierarchical graphs, making this a novel challenge.

(ii) Generating Sufficient Training Samples. Unlike existing studies on learned indexes [39, 46, 49], the training data for our model is not explicitly provided by the input dataset. Therefore, we need to devise an effective algorithm to generate sufficient training samples to build an accurate learned shortcut.

To tackle these challenges, we first introduce Lemma 2 as the guideline of *determining safe skip levels* and present the detailed method for *generating sufficient training samples* in Algo. 2.

Guideline for Skipping Levels. We propose a *density-based criterion* to determine how many levels can be safely skipped. Specifically, consider o as the nearest neighbor of the query vector q at level x, which will serve as the entry point at level $x+1$. If there are already numerous nearby vectors at level x that exhibit short distances to q (but longer than o), it is likely that o will remain the nearest neighbor to q at level $x+1$. In such cases, traversing the proximity graph from o at level $x+1$ becomes redundant. Since o is the nearest vector to q, the nearby vectors of q should also be close to o. Therefore, we use the density around the data vector $o \in \mathcal{D}$ to reflect the presence of nearby vectors relative to the query vector q, and derive Lemma 2 as the final guideline.

Lemma 2. Given a vector o in the hierarchical graph at level x, we say that the levels from x down to $y+1$ can be skipped if y is the first level lower than x satisfying the following condition:

$$\begin{aligned}&\left(dis(level_x(o), level_x(o_{x,k}))\right)^{d_x}\\ \leq\ &\frac{n_y \cdot \pi^{\frac{d_y}{2}} \cdot \Gamma(\frac{d_x}{2}+1)}{n_x \cdot \pi^{\frac{d_x}{2}} \cdot \Gamma(\frac{d_y}{2}+1)} \cdot \left(dis(level_y(o), level_y(o_{y,k}))\right)^{d_y}\end{aligned}$$

The parameter η in hierarchical vector compression is 2.

Proof. We adopt the k nearest neighbor (NN) based estimation method [7] to estimate the density of a vector $o \in \mathcal{D}$ as follows:

$$Density(o) = \frac{k}{n \cdot V_d \cdot (dis(o, o_k))^d} \tag{12}$$

Here, $V_d = \frac{\pi^{d/2}}{\Gamma(d/2+1)}$ is the volume of the d-dimensional unit ball, $\Gamma(d) = (d-1)!$ is the gamma function, and o_k denotes the kth NN to the vector o in the dataset $\mathcal{D}$.

Extending Eq. (12) to a vector o from the vector collection at any level x, we estimate its density as:

$$Density(o, x) = \frac{k}{n_x \cdot V_{d_x} \cdot \left(dis(level_x(o), level_x(o_{x,k}))\right)^{d_x}} \tag{13}$$

where n_x is the number of vectors at level x, $d_x = \frac{d}{2^{x-1}}$ is the corresponding vector dimension, and $o_{x,k}$ is o's kth NN obtained through traversing the proximity graph at level x. Similarly, the density of a vector o at level y is denoted by $Density(o, y)$.

The key condition for safely skipping from level x directly to the lower level y requires that the density around o at level x is no less than its density at level y:

$$Density(o, x) \geq Density(o, y) \tag{14}$$

Substituting the density expressions from Eq. (13) into both sides of the condition in Eq. (14) yields the following inequality:

$$\begin{aligned}&\left(dis(level_x(o), level_x(o_{x,k}))\right)^{d_x}\\ \leq\ &\frac{n_y}{n_x} \cdot \frac{V_{d_y}}{V_{d_x}} \cdot \left(dis(level_y(o), level_y(o_{y,k}))\right)^{d_y}\\ =\ &\frac{n_y \cdot \pi^{\frac{d_y}{2}} \cdot \Gamma(\frac{d_x}{2}+1)}{n_x \cdot \pi^{\frac{d_x}{2}} \cdot \Gamma(\frac{d_y}{2}+1)} \cdot \left(dis(level_y(o), level_y(o_{y,k}))\right)^{d_y}\end{aligned} \tag{15}$$

This inequality matches the lemma's skipping condition. □

Rationality of Lemma 2. Please refer to our full paper [24].

Algorithm Details. Based on Lemma 2, we now introduce the index construction algorithm in Algo. 2. Lines 1-8 follow the construction method of HNSW [36] to build hierarchical graphs, as introduced in Algo. 1, where $\mathcal{L}$ denotes the number of levels. The *primary distinction* is that line 7 computes approximate distances using compressed vectors instead of exact distances. This modification accelerates the construction time. Subsequently, lines 9-19 generate the training data samples S and build the learned shortcut. Specifically, each vector $o \in \mathcal{D}$ is treated as a query vector to search its nearest neighbor at each level (lines 10-11). For each level x, lines

Algorithm 2: Construct our index SHG

Input: Dataset $\mathcal{D}$, normalization factor m_L for index level
Output: Our index SHG

```
// Hierarchical Graph Construction Phase:
SHG ← ∅;
foreach Vector o ∈ D do
    l ← ⌊− ln (uniform(0, 1)) · m_L⌋;
    if l is larger than current top level then
        ep ← o;
    for j ← l to 1 do
        Select neighbors at level l with approximate distance
          dis(level_l(o), level_l(·));
        Build the NSW graph with neighbors at level l;
// Learned Shortcut Construction Phase:
Training data set S ← ∅;
foreach vector o ∈ dataset D do
    Search nearest neighbor (NN) to o at each level;
    for level x ← L to 1 do
        for level y ← 1 to x − 1 do
            nn_x, nn_y ← o's nearest neighbor at levels x, y;
            if the condition in Lemma 2 is satisfied then
                d̂is ← dis(level_x(o), level_x(nn_x));
                h ← x − y, then add (d̂is, h) into S;
                break;
Learned shortcut f(·) ← train on dataset S;
return SHG with learned shortcut f(·);
```

13-18 identify the lowest level y that satisfies Lemma 2. If such y exists, the tuple $(\widehat{dis}, h)$ is added to S, where $\widehat{\text{dis}}$ is the approximate distance and h is the number of skipped levels. To train the shortcut, S is first divided into segments, and then linear models are used to fit the data within each segment (see [17] for more details).

Example 5. As shown in Fig. 3b, we simplify this example by considering only two data points, o_2 and o_4. Starting at level 4, o_1 is the nearest vector to o_2 with an approximate distance $\widehat{dis} = dis(level_4(o_2), level_4(o_1))$. Upon checking level 1, we assume that the nearest neighbor to o_2 meets the criteria of Lemma 2. This means we already find a vector (o_1) at level 4 that is close enough to o_2, even compared to its nearest neighbor at level 1. Consequently, we can safely skip from level 4 down to level 1 at o_2, pruning $h = 3$ levels. We add $(\widehat{dis}, h)$ into the training samples S. Next, at level 3, o_3 is the nearest vector to o_4. At level 1, we assume the nearest neighbor to o_4 satisfies the distance condition in Lemma 2. Therefore, at level 3, we have also identified a vector with a sufficiently short distance to o_4, allowing us to skip from level 3 down to level 1. We subsequently add the corresponding tuple to S. Similarly, we traverse the hierarchical graphs and generate the training data S.

4.3 k-ANN Search

Main Idea. Our index accelerates search efficiency in two ways:

Algorithm 3: k-ANN search using our index SHG

Input: Query vector q, index SHG
Output: Approximate k-nearest neighbors A to q

```
1  ep ← entry vector at level L of SHG, l ← L;
2  d̂is ← dis(level_L(q), level_L(ep));
3  while l − f(d̂is) ≥ 1 do
4      Skip levels from l down to (l − f(d̂is));
5      Current level l ← l − f(d̂is);
6      Entry vector ep ← search nearest neighbor to q at level
         l using approximate distance dis(level_l(q), level_l(·));
7      d̂is ← approximate distance dis(level_l(q), level_l(ep));
8  Start searching kNN to q from ep at level 1 using a heap W;
9  dis_k ← track kth nearest distance from vectors in W to q;
10 while |W| > 0 do
11     o ← extract the nearest vector in W to q ;
12     Pop the vector o from W ;
13     foreach Vector u ∈ o's neighbourhood do
14         lowerbound ← η^{L/2} · dis(level_L(q), level_L(u)) ;
15         if lowerbound ≤ dis_k then
16             Calculate exact distance dis(q, u) ;
17             if dis(q, u) < dis_k then
18                 Push vector u into W, and update dis_k;
19 return Query answer A ← the kNN to q extracted from W;
```

- Using the vector compression method in Sec. 3, upper levels require fewer dimensions to compute approximate distances. This significantly reduces computational costs.
- Using the learned shortcut in Sec. 4.1, we can infer the number of levels that are confident enough to be skipped. This reduces the time spent traversing unnecessary levels.

Algorithm Details. Algo. 3 presents our k-ANN search method. Lines 1-7 cover the search process of upper levels, starting from the entry ep and computing approximate distances to q. Lines 3-7 navigate hierarchical graphs to reach the base level, with lines 4-5 using learned shortcuts to skip certain levels with approximate distances. Lines 8-19 describe the base-level search, beginning from ep at level 1, with a k-sized heap W maintaining nearby neighbors to q. Lines 13–18 determine whether neighboring vectors of o in the proximity graph at the base level qualify as next candidates. Specifically, for each neighborhood vector u, line 14 computes its approximate distance to q, which serves as a lower bound for the exact distance based on Theorem 1. When this lower bound exceeds the distance threshold dis_k (*i.e.*, the kth nearest distance from vectors in W to q), u is eliminated from consideration. Otherwise, lines 17-19 compute u's exact distance to q and compare it with dis_k to decide whether u should be added to W as a candidate.

Example 6. As shown in Fig. 2b, the query vector q is marked in red, and we assume $ep = o_1$ at level 4. Initially, the learned shortcut infers that the approximate distance $dis(level_4(q), level_4(o_1))$ is large, indicating that a finer search is needed. Consequently, the shortcut outputs 1, resulting in a skip from level 4 down to level 3. At this level, we identify that o_3 has the nearest distance to q. Using

the distance, the shortcut then infers that 2 levels can be skipped. Based on this inference, traversal at level 2 is unnecessary, and o_3 serves as the entry vector to traverse the proximity graph at level 1. Since level 1 contains all data objects, we use a k-sized heap to maintain the final k-ANN when traversing the graph.

4.4 Complexity Analysis

This subsection offers a complexity analysis of *construction time*, *index size*, and *search time*. For simplicity, we denote the number of vectors by n and consider HNSW [36] as the hierarchical graph.

Construction Complexity. Lines 1-8 of Algo. 2 constructs HNSW in $O(n \log n)$ time [36]. Each loop of line 10 requires $O(\log n)$ time for searching HNSW [36] in line 11 and $O(\mathcal{L}^2)$ time for lines 12-18, where $\mathcal{L}$ is the number of levels in HNSW. With $O(n\mathcal{L})$ training samples, line 19 builds the learned shortcut using the method in [17] in $O(n\mathcal{L})$ time. The overall complexity remains $O(n \log n)$, as $\mathcal{L}$ is typically small (*e.g.*, 3-5) in real-world datasets (in Sec. 5).

Index Space Cost. In addition to the memory $\mathcal{S}$ required for hierarchical graphs, our index incurs two additional memory costs: *(i) compressed vector coordinates* and *(ii) learned shortcut*. Compressed vectors take $O(n)$ space, while the learned shortcut takes almost constant space compared to $\mathcal{S}$. For instance, in our evaluations, the space occupancy of the learned shortcut is less than 3 MB, while $\mathcal{S}$ is 2,503 MB for Deep100M. The overall space cost is $\mathcal{S} + O(n)$.

Search Complexity. Algo. 3 requires $O(1)$ time to infer the skipped levels. By leveraging our index, certain levels can be skipped during traversal of the hierarchical graphs. Thus, the search complexity remains no higher than that of HNSW [36], *i.e.*, $O(\log n)$.

4.5 Index Maintenance

Although ANN search is often studied over static data, some recent studies also consider the issue of *index maintenance* over dynamic updates. In general, dynamic updates involve two scenarios: *deleting existing vectors* and *inserting new vectors*. Our index SHG can effectively handle either scenario with the following strategies:

Deletion. In hierarchical graph based indexes, such as those implemented in the HNSW library [1], a prevalent method for deleting vectors is to mark them as deleted and verify these markers during graph traversal. This method is also generally effective for our index. However, if a substantial portion of the dataset (*e.g.*, 50%) has been deleted, up to half of the vectors at each level may be non-existent. In such cases, search efficiency might be reduced, so it may be beneficial to rebuild the entire index from scratch.

Insertion. Our index takes two main steps to insert a new vector:

(i) Insertion in Hierarchical Graph. This step adheres to the insertion routine for hierarchical graph-based indexes. Taking HNSW [36] as an example, inserting a new vector involves establishing new edges between existing vectors and the new vector. It takes $O(\log n)$ time [36] to insert a vector into a set of n vectors.

(ii) Insertion in Learned Shortcut. Our SHG additionally performs insertions in learned shortcuts. It inserts the training data associated with the new vector and leverages the foundation model, PGM index [17], to process these updates in $O(\log n)$ time [3].

Table 2: Summary of datasets

Dataset	Dim.	Card.	#(Query)	Type	#(Level)
OpenAI	1,536	1M	10,000	Text	4
Enron	1,369	94K	200	Text	3
GIST1M	960	1M	1,000	Image	3
Msong	420	992K	200	Audio	3
UQ-V	256	1M	10,000	Video	3
MsTuring10M	100	10M	1,000	Text	4
SIFT100M	128	100M	10,000	Image	4
Deep100M	96	100M	10,000	Image	4

5 EXPERIMENTAL STUDY

This section first presents the setup (Sec. 5.1), then evaluates (1) construction cost, (2) search performance, (3) effectiveness of our vector compression method, and (4) capability of updating data (Sec. 5.2–5.5). Finally, the main results are summarized in Sec. 5.6.

5.1 Experimental Setup

Datasets. As outlined in Table 2, we evaluate on eight real-world datasets spanning diverse cardinalities (94K-100M) and dimensions (96-1536). These datasets are widely considered as standard benchmarks for testing k-ANN search. For example, Enron, Msong, and UQ-V have been utilized in prior survey [50] of state-of-the-art (SOTA) graph-based methods, whereas the other datasets originate from the open-sourced competitions [42] hosted by the NeurIPS community. Each dataset also provides a test set as query vectors.

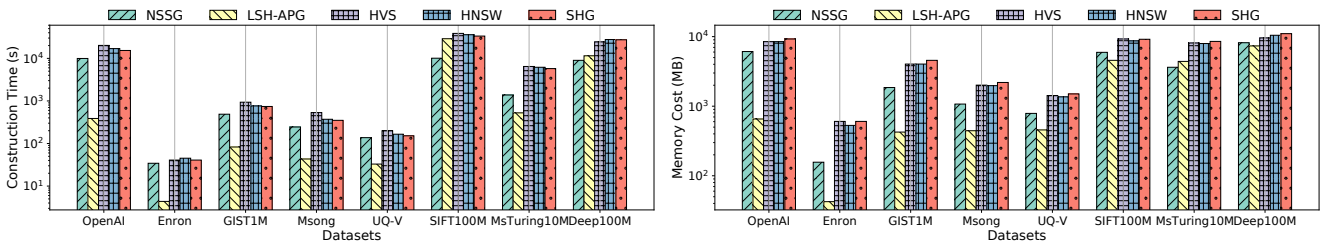
Compared Algorithms. We chose the graph-based methods HNSW [36], NSSG [19], and HVS [34] as our competitors because they outperform other in-memory ANNS techniques [47, 50], such as tree-based [6, 37] and quantization-based methods [23, 26]. Furthermore, we also incorporate LSH-APG [56] as a recent SOTA baseline, as it effectively integrates the advantages of locality-sensitive hashing (LSH) with the strengths of graph-based approaches.

- **HNSW** [36]. We set $M = 48$ and $efConstruction = 80$, aligning with the parameters used in existing work [52, 56].
- **NSSG** [19]. NSSG is an enhanced variant of NSG [20]. Following its original implementation, we configure the parameters as follows: $L = 500$, $R = 60$, and $Angle = 60$.
- **HVS** [34]. The parameters of HVS are set as $M = 32$, $T = 4$, and $efConstruction = 500$. Besides, we sample 100,000 vectors as training samples for each dataset. These settings are consistent with those in the original paper [34].
- **LSH-APG** [56]. We set $K = 16$, $L = 2$, $T = 24$, $T' = 2T$ and $p_\tau = 0.95$, which are the same as the settings in its code.

Both HNSW and SHG maintain $\mathcal{L} < 5$ levels across datasets (see Table 2). Please see our full paper [24] for evaluations on larger $\mathcal{L}$.

Metrics. We measure the construction costs of each index by assessing its *memory cost* and *construction time*. As for search performances, *recall* (defined in Eq. (1) with the default $k = 20$) is used to measure accuracy, and efficiency is assessed using *search time*, which represents the average time required to process each query.

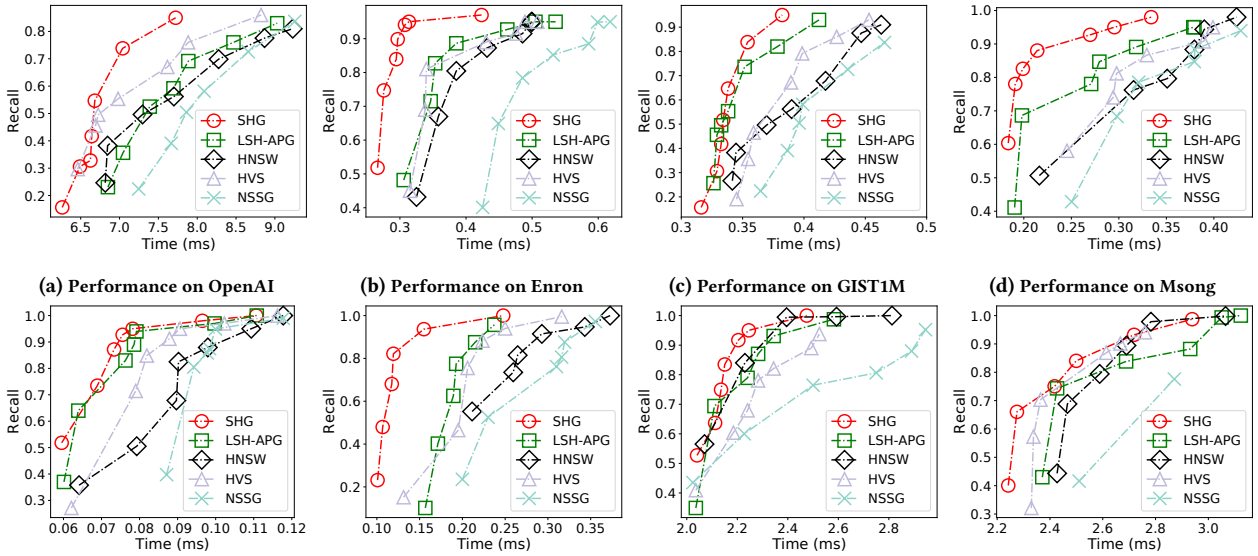
Implementation. We implement our method as well as all competing algorithms in C++, using the g++ compiler. All experiments are



(a) Comparisons on index construction time

(b) Comparisons on index memory cost

Figure 4: Evaluation of index construction on eight real-world datasets



(a) Performance on OpenAI

(b) Performance on Enron

(c) Performance on GIST1M

(d) Performance on Msong

(e) Performance on UQ-V

(f) Performance on MsTuring10M

(g) Performance on SIFT100M

(h) Performance on Deep100M

Figure 5: Search performance on eight real-world datasets (with default query parameter $k = 20$)

conducted in a single-threaded environment on a machine equipped with 40 Intel(R) Xeon(R) E5 2.30GHz processors with 1 TB of RAM.

5.2 Evaluation of Index Construction

Construction Time. Fig. 4a compares the construction time across eight datasets. While our SHG and other hierarchical graph-based indexes (*e.g.*, HNSW and HVS) require more time than LSH-APG, the total time cost remains acceptable given the data sizes. Our SHG completes both training sample generation and learned shortcut construction within 80 seconds across these datasets (see Table 3). However, by leveraging approximate distances of compressed vectors, our index SHG achieves a 9%-23% reduction in construction time compared to HNSW. Besides, our method reduces construction time by 12%-33% relative to HVS in Fig. 4a.

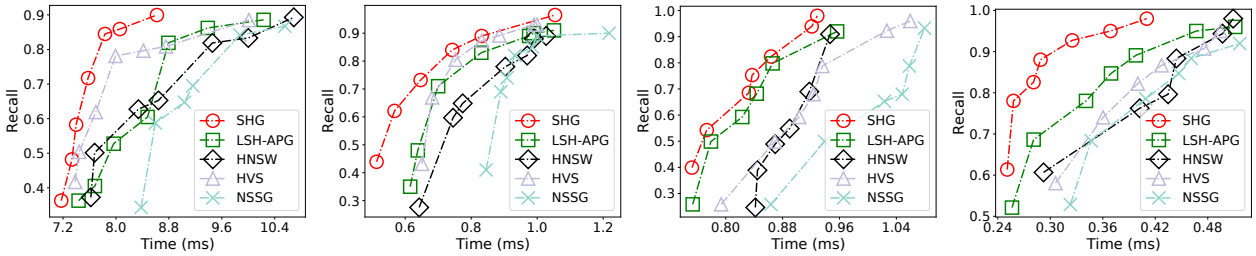
Memory Cost. Fig. 4b compares the memory costs associated with various methods. Compared to HNSW, our SHG requires additional but marginal space for learned shortcuts. For example, under the four largest datasets, this overhead remains below 3 MB as shown in Table 3. While LSH-APG incurs the lowest memory cost, all indexes can fit within the main memory of a modern server.

Table 3: Time and memory cost for learned shortcuts.

Dataset	OpenAI	MsTuring10M	SIFT100M	Deep100M
Time (s)	23.18	1.89	79.07	15.87
Memory (MB)	0.776	0.840	1.120	2.440

5.3 Evaluation of Search Performance

The evaluations encompass experiments with different settings of the query parameter k and assessments of performance robustness. Adhering to the benchmark evaluation protocols [31, 42, 50], we present the results via *recall-time curves*. Specifically, for each dataset, k-ANN searches are conducted using an independent query set. This query set is then divided into equally sized subsets, ordered by recall values. We report the average recall and average time for processing query vectors within each subset.



(a) Performance on OpenAI (b) Performance on Enron (c) Performance on GIST1M (d) Performance on Msong

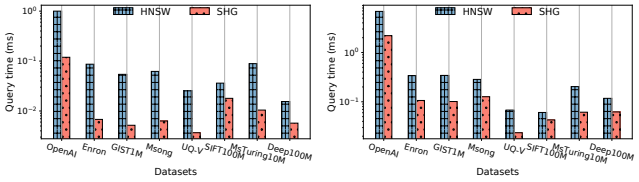
Figure 6: Search performance on real-world datasets (with larger query parameter $k = 50$)

Search Performance on Default Query Parameter. Fig. 5 illustrates the overall search performances of the compared methods (with default $k = 20$). Overall, the results demonstrate that our index SHG outperforms other methods, particularly excelling in achieving recall rates exceeding 80% within the shortest time. Moreover, our method shows notable advantages on those datasets (*e.g.*, Enron). For large-scale datasets with 100 million vectors (*i.e.*, Deep100M and SIFT100M), our method demonstrates superior search performance. On the SIFT100M dataset, our method achieves over 90% recall in 2.2 ms, whereas HNSW takes the same time but achieves less than 80% recall. Similarly, on the Deep100M dataset, our method reaches over 80% recall faster than HNSW.

Specifically, for datasets with vectors exceeding 1,000 dimensions, our method (SHG) requires less time (10 ms) than LSH-APG to achieve a recall greater than 85%. To reach a recall of 95% on the Enron dataset, SHG requires 20% less time than LSH-APG. On datasets with 10 million vectors, the advantages of SHG are even more pronounced. It takes about 1 ms for SHG to achieve a recall of over 85%, while all other methods consume more than 2.5 ms to reach the same recall on the MsTuring10M dataset. For Deep100M dataset, SHG is still the fastest, achieving over 85% recall in about 2.5 ms, whereas LSH-APG only reaches the recall less than 80%, and others perform even worse. For instance, HVS and HNSW achieves just above 75% recall. For other datasets, SHG demonstrates a performance improvement of 1.5× faster in reaching 90% recall than LSH-APG, and it can be 1.8× faster than HNSW. Moreover, our method is also fastest in achieving recall above 90% on the GIST1M, UQ-V, and SIFT100M datasets.

The efficiency gains stem from acceleration at both upper levels and the base level. As shown in the time breakdown in Fig. 7, our solution SHG achieves consistent time reductions across all levels, compared to the baseline HNSW. These improvements result from two key optimizations: (1) learned shortcuts that skip unnecessary level navigations, and (2) compressed vectors enabling approximate distance computations through fewer dimension scans. The results also demonstrate that searching upper levels account for 18%–42% (when $recall \geq 80\%$) of total search time in HNSW, indicating non-negligible computation overhead.

Search Performance on Varied Query Parameter. Fig. 6 shows the search performance when the query parameter k increases to 50. We observe that all methods exhibit increasing time to achieve the same recall value when $k = 20$. However, our method consistently maintains the fastest time than others to reach the same recall.



(a) Average time (upper levels) (b) Average time (base level)

Figure 7: Average search time across upper and base levels ($recall \geq 80\%$)

Specifically, our method takes less than 0.7 ms for the Enron dataset and less than 0.9 ms for the other datasets to achieve 90% recall. On the GIST1M dataset, our method consistently outperforms others when the recall exceeds 40%. Compared to the results in Fig. 5c, LSH-APG achieves a 50% recall slightly faster than SHG when $k = 20$. This trend indicates that the advantages of our index over others continue to grow as k increases.

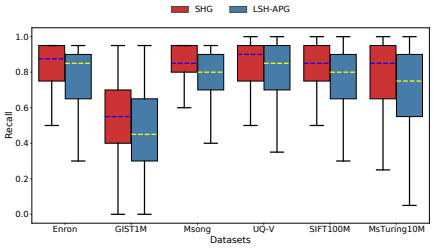
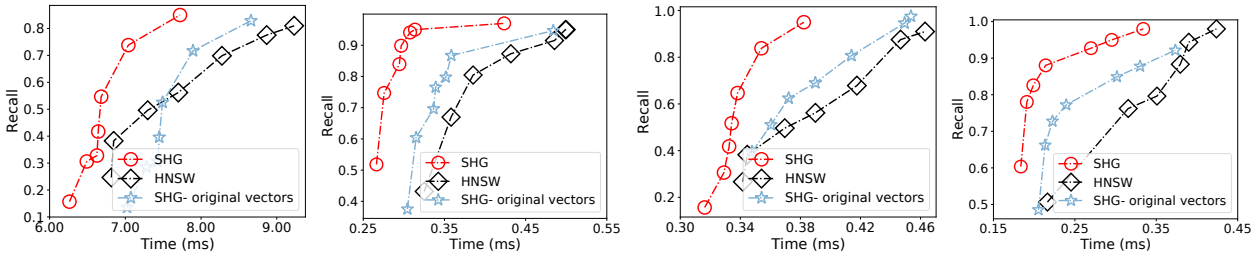


Figure 8: Recall robustness between SHG and LSH-APG

Relative Recall Robustness Between SHG and LSH-APG. This experiment only compares our SHG with LSH-APG due to its superior performance over other baselines. Specifically, we sample a new set of query vectors that are not included in the original dataset for index construction. We refer to them as "*unseen vectors*". As shown in Fig. 8, we use *boxplots* (*a.k.a.*, *box-and-whisker plots*) to show the distributions of recall values for these queries. In boxplots, the solid horizontal lines at the lower and upper ends (*i.e.*, *whiskers*) represent the lowest and highest recall values, respectively. The size of the rectangular *box* reflects the recall stability. The dashed horizontal line within each box represents the median recall value.



(a) Search on OpenAI (b) Search on Enron (c) Search on GIST1M (d) Search on Msong

Figure 9: Ablation study on our hierarchical vector compression method (SHG-*original vectors*)

Compared to previous evaluations, the search performance of LSH-APG significantly declines on these unseen vectors. For example, using LSH-APG to answer k-ANN search, the worst-case recall is consistently below 40% across all datasets. In contrast, SHG demonstrates more robust search performance than LSH-APG. For instance, on the Msong dataset, the worst recall achieved by SHG exceeds 60%. This indicates that using SHG instead of LSH-APG can improve recall by over 20% for certain queries. Moreover, the box sizes of SHG are always smaller than those of LSH-APG, and the median recall of SHG is consistently higher. These results demonstrate that SHG is more robust and stable than LSH-APG. The improvement in robustness is particularly important in real-world applications, since users continually launch various new search requests that often involve unseen data objects.

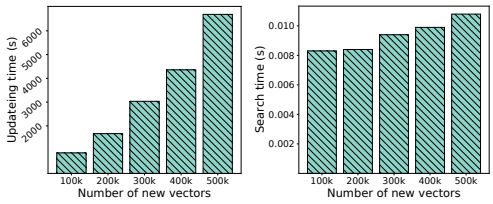
5.4 Ablation Study: The Effectiveness of Our Hierarchical Vector Compression Method

Setup. This ablation experiment is designed to verify the effectiveness of our hierarchical vector compression method. To this end, we construct our index using the original uncompressed vectors across all levels, denoting this solution as SHG-*original vectors*. Fig. 9 illustrates the comparisons between HNSW [36], SHG-*original vectors*, and SHG. Please refer to our full paper [24] for detailed evaluations.

Observation #1: SHG-*original vectors* outperforms HNSW in most cases. Across these datasets, the index SHG-*original vectors* performs better than HNSW [36], in most cases. For example, on the SIFT100M dataset, SHG-*original vectors* requires 8% less time than HNSW to reach a 90% recall. On the Deep100M dataset, HNSW takes 7% more time than SHG-*original vectors* to reach a recall of above 80%. This performance advantage indicates that shortcuts effectively reduce computational costs.

Observation #2: SHG achieves a superior query performance compared to SHG-*original vectors*. The comparisons demonstrate that our vector compression method, which employs varying approximation scales, facilitates more accurate inferences on the number of skipped levels via learned shortcuts.

Overall, our solution achieves superior search efficiency through hierarchical vector compression, at the additional memory cost for compressed vector representations.



(a) Update time (b) Search time

Figure 10: Update performance on the OpenAI dataset

5.5 Evaluation of Handling Data Updates

Setup. This evaluation is conducted on the OpenAI dataset, which contains a total of 2,321,096 vectors. For previous evaluations, we used the first one million vectors. To assess the capability of SHG in handling data updates, we further inserted the next $100k$, $200k$, $300k$, $400k$, $500k$ vectors into the previously built index.

Result. Fig. 10 presents the evaluation results. Specifically, the update time represents the total time spent on designated insertions. In Fig. 10a, each new vector takes 12 ms to be inserted, achieving rapid latency for handling updates. Meanwhile, the recall decreases slightly (see our full paper [24] for evaluations). Regarding search time, it increases marginally when inserting new vectors. Both trends are reasonable given the expanding data scalability. These results demonstrate that our index maintains a robust performance when handling data updates.

5.6 Summary of Experimental Results

After conducting the previous experiments, we summarize the major results as follows:

(i) Limitations of hierarchical graph-based indexes: The experimental study reveals two key limitations in hierarchical graph-based indexes (*e.g.*, HNSW): (1) some intermediate levels can often be avoided to improve search efficiency and (2) exact distance calculations are computationally expensive.

(ii) Superior search performance of our index: Our proposed index SHG, which leverages two novel methods (hierarchical vector compression and learned shortcuts), can accelerate HNSW by 1.8× while maintaining the same search accuracy. This enhanced performance is mainly achieved with an additional memory overhead of 3 MB for storing the shortcuts.

(iii) More robust effectiveness for searching arbitrary vectors: Our index SHG also shows a more robust search performance compared to the SOTA graph-based method, LSH-APG. For example, to reach a 90% recall, SHG is 1.5× faster than LSH-APG. Moreover, our method exhibits greater robustness when searching "unseen" vectors that are not included in the original dataset used for constructing indexes. In these k-ANN queries, which are common in real-world applications, our index can improve search recall by up to 20% compared to LSH-APG.

(iv) Capability to handle data updates in our index: Our index handles updates efficiently with 12 ms per insertion latency, and maintains over 95% recall and 10 ms search time.

6 RELATED WORK

This section reviews related studies from two perspectives: *ANN search over high-dimensional vector data*, and *learning-based indexes*.

6.1 ANN Search over High-Dimensional Vectors

To address the challenges of k-ANN search over large-scale high-dimensional vector data, there are three mainstream strategies: *product quantization*, *hashing*, and *graph-based indexes*.

Product quantization based methods [21, 29] partition the dataset into clusters based on quantization values, streamlining the identification of candidate vectors that match the query. Locality-Sensitive Hashing (LSH) is commonly used in *hashing* based methods [11, 22, 27, 52]. These types of hashing functions are used to project high-dimensional vectors into some low-dimensional hash buckets, allowing searches to check the buckets containing the query.

By comparison, *graph-based indexes* [34, 36, 56] have demonstrated superior search performance compared to other solutions, as demonstrated in recent experimental surveys [5, 31, 47, 50]. These indexes create a proximity graph where each node represents a dataset vector, and edges connect neighboring vectors for efficient ANN searches. NN-Descent [12] was first introduced to reduce construction complexity and has since been widely adopted [20, 54, 55]. To further improve construction efficiency, NSW [35] proposes a consecutive insertion strategy to insert vectors one by one to update the graph structure. LSH-APG [56] aims to integrate this strategy and LSH to devise a novel indexing mechanism. HNSW [36] is probably the most popular graph-based index in vector databases. It introduces a hierarchical structure to mitigate the issue of hubness, ensuring that the degree of each graph node is constrained at every level. Its superior performance leads to several important derivative methods, such as HCNNG [38] and HVS [34].

6.2 Learning-Based Indexing

The learned index [30] optimizes indexing by using machine learning models. It leverages the ML model to capture data distribution characteristics, effectively "replacing" traditional index structures. This technique significantly reduces both space and query costs.

ZM-Index [49], Flood [39], and LIMS [46] are recent popular learned indexes designed for multi-dimensional data, yet they are limited to data dimensions below 65, and struggle with scalability for high dimensions. ZM-Index [49] lacks support for k-nearest neighbors (kNN) queries and struggles to efficiently accommodate data updates. Flood [39] partitions the data space into grid cells across dimensions to ensure an even distribution of points within each partition. Intuitively, this strategy is more effective for low-dimensional data and may not perform as well in higher dimensions. As for LIMS [46], it focuses on exact similarity search instead of k-ANN search. PGM-index [15, 17] is an efficient learned index offering theoretical guarantees for query performance, space efficiency, and updates. By segmenting data and using piecewise linear approximations, it outperforms traditional index structures in space efficiency while maintaining high query efficiency. While we use PGM-index as the foundation model for learned shortcuts, this index is primarily suited for low-dimensional data and not for high-dimensional vector data.

7 CONCLUSION

This paper aims to accelerate k-ANN search over high-dimensional vector data. While hierarchical graph-based indexes are widely adopted in vector databases, their query efficiency still suffers from exhaustive navigations of proximity graphs across all levels. To mitigate this issue, we propose a new index named SHG that avoids traversing unnecessary levels. SHG index incorporates a hierarchical vector compression algorithm that progressively reduces vector dimensions at each level while ensuring approximation for computed distances. Moreover, we design a novel data structure called shortcut that learns from inter-level approximate distances to infer the number of levels that can be safely skipped. We evaluate our solution against state-of-the-art methods using eight benchmark datasets. Overall, our approach achieves a 1.5–1.8× speedup in search efficiency compared to existing methods and demonstrates superior robustness in terms of query recall.

ACKNOWLEDGMENT

Lei Chen's work is partially supported by National Key Research and Development Program of China Grant No. 2023YFF0725100, National Science Foundation of China (NSFC) under Grant No. U22B2060, Guangdong-Hong Kong Technology Innovation Joint Funding Scheme Project No. 2024A0505040012, the Hong Kong RGC GRF Project 16213620, RIF Project R6020-19, AOE Project AoE/E-603/18, Theme-based project TRS T41-603/20R, CRF Project C2004-21G, Key Areas Special Project of Guangdong Provincial Universities 2024ZDZX1006, Guangdong Province Science and Technology Plan Project 2023A0505030011, Guangzhou municipality big data intelligence key lab, 2023A03J0012, Hong Kong ITC ITF grants MHX/078/21 and PRP/004/22FX, Zhujiang scholar program 2021JC02X170, Huawei collaboration project no. TC20230524054, Microsoft Research Asia Collaborative Research Grant, HKUST-Webank joint research lab and 2023 HKUST Shenzhen-Hong Kong Collaborative Innovation Institute Green Sustainability Special Fund, from Shui On Xintiandi and the InnoSpace GBA. Yuxiang Zeng's work is partially supported by National Key Research and Development Program of China under Grant No. 2023YFF0725103, National Science Foundation of China (NSFC) (Grant Nos. 62425202, U21A20516, 62336003), the Beijing Natural Science Foundation (Z230001), the Didi Collaborative Research Program and the State Key Laboratory of Complex & Critical Software Environment (SKLCCSE). Yuxiang Zeng and Lei Chen are the corresponding authors.

REFERENCES

[1] [n.d.]. HNSW Library. https://github.com/nmslib/hnswlib.

[2] [n.d.]. Milvus: The High-Performance Vector Database Built for Scale. https://milvus.io/.

[3] [n.d.]. The PGM Index. https://pgm.di.unipi.it/.

[4] [n.d.]. Pinecone: The vector database to build knowledgeable AI. https://www.pinecone.io/.

[5] Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.* 87 (2020), 101374.

[6] Alina Beygelzimer, Sham M. Kakade, and John Langford. 2006. Cover trees for nearest neighbor. In *ICML*, Vol. 148. 97–104.

[7] Gérard Biau and Luc Devroye. 2015. *Lectures on the nearest neighbor method.* Springer.

[8] Brankica Bratic, Michael E. Houle, Vladimir Kurbalija, Vincent Oria, and Milos Radovanovic. 2018. NN-Descent on High-Dimensional Data. In *WIMS*. 20:1–20:8.

[9] Sebastian Bruch. 2024. *Foundations of Vector Retrieval.* Springer.

[10] Cheng Chen, Chenzhe Jin, Yunan Zhang, Sasha Podolsky, Chun Wu, Szu-Po Wang, Eric Hanson, Zhou Sun, Robert Walzer, and Jianguo Wang. 2024. SingleStore-V: An Integrated Vector Database System in SingleStore. *PVLDB* 17, 12 (2024), 3772–3785.

[11] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*. 253–262.

[12] Wei Dong, Moses Charikar, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*. 577–586.

[13] Rick Durrett. 2010. *Probability: Theory and Examples, 4th Edition.* Cambridge University Press.

[14] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2020. Why Are Learned Indexes So Effective?. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research)*, Vol. 119. PMLR, 3123–3132. http://proceedings.mlr.press/v119/ferragina20a.html

[15] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2020. Why Are Learned Indexes So Effective?. In *ICML*, Vol. 119. 3123–3132.

[16] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2021. On the performance of learned data structures. *Theor. Comput. Sci.* 871 (2021), 107–120. https://doi.org/10.1016/J.TCS.2021.04.015

[17] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB* 13, 8 (2020), 1162–1175.

[18] Cong Fu and Deng Cai. 2016. EFANNA : An Extremely Fast Approximate Nearest Neighbor Search Algorithm Based on kNN Graph. *CoRR* abs/1609.07228 (2016).

[19] Cong Fu, Changxu Wang, and Deng Cai. 2022. High Dimensional Similarity Search With Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE Trans. Pattern Anal. Mach. Intell.* 44, 8 (2022), 4139–4150.

[20] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph. *PVLDB* 12, 5 (2019), 461–474.

[21] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *IEEE Trans. Pattern Anal. Mach. Intell.* 36, 4 (2014), 744–755.

[22] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB*. 518–529.

[23] Yunchao Gong, Svetlana Lazebnik, Albert Gordo, and Florent Perronnin. 2013. Iterative Quantization: A Procrustean Approach to Learning Binary Codes for Large-Scale Image Retrieval. *IEEE Trans. Pattern Anal. Mach. Intell.* 35, 12 (2013), 2916–2929.

[24] Zengyang Gong, Yuxiang Zeng, and Lei Chen. 2025. *Accelerating Approximate Nearest Neighbor Search in Hierarchical Graphs: Efficient Level Navigation with Shortcuts (Full Paper).* Technical Report. https://drive.google.com/file/d/161dSKDpCwUuYe3DzRl11M1A3VYFcmjBl/view?usp=sharing

[25] Yikun Han, Chunjiang Liu, and Pengfei Wang. 2023. A Comprehensive Survey on Vector Database: Storage and Retrieval Technique, Challenge. *CoRR* abs/2310.11703 (2023).

[26] Jae-Pil Heo, Zhe L. Lin, and Sung-Eui Yoon. 2019. Distance Encoded Product Quantization for Approximate K-Nearest Neighbor Search in High-Dimensional Space. *IEEE Trans. Pattern Anal. Mach. Intell.* 41, 9 (2019), 2084–2097.

[27] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-Aware Locality-Sensitive Hashing for Approximate Nearest Neighbor Search. *PVLDB* 9, 1 (2015), 1–12.

[28] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*. 604–613.

[29] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (2011), 117–128.

[30] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*. 489–504.

[31] 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 Trans. Knowl. Data Eng.* 32, 8 (2020), 1475–1488.

[32] Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, Chen Chen, Fan Yang, Yuqing Yang, and Lili Qiu. 2024. RetrievalAttention: Accelerating Long-Context LLM Inference via Vector Retrieval. *CoRR* abs/2409.10516 (2024).

[33] Qiyu Liu, Maocheng Li, Yuxiang Zeng, Yanyan Shen, and Lei Chen. 2025. How good are multi-dimensional learned indexes? An experimental survey. *VLDB J.* 34, 2 (2025), 17.

[34] Kejing Lu, Mineichi Kudo, Chuan Xiao, and Yoshiharu Ishikawa. 2021. HVS: Hierarchical Graph Structure Based on Voronoi Diagrams for Solving Approximate Nearest Neighbor Search. *PVLDB* 15, 2 (2021), 246–258.

[35] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Inf. Syst.* 45 (2014), 61–68.

[36] Yury A. Malkov and Dmitry A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020), 824–836.

[37] Marius Muja and David G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Trans. Pattern Anal. Mach. Intell.* 36, 11 (2014), 2227–2240.

[38] Javier Alvaro Vargas Muñoz, Marcos André Gonçalves, Zanoni Dias, and Ricardo da Silva Torres. 2019. Hierarchical Clustering-Based Graphs for Large Scale Approximate Nearest Neighbor Search. *Pattern Recognit.* 96 (2019), 106970.

[39] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *SIGMOD*. 985–1000.

[40] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of vector database management systems. *VLDB J.* 33, 5 (2024), 1591–1615.

[41] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector Database Management Techniques and Systems. In *SIGMOD*. 597–604.

[42] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamy, Gopal Srinivasa, Suhas Jayaram Subramanya, and Jingdong Wang. 2021. Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search. In *NeurIPS*, Vol. 176. 177–189.

[43] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned Index: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 16, 8 (2023), 1992–2004. https://doi.org/10.14778/3594512.3594528

[44] Bo Tang and Haibo He. 2015. ENN: Extended Nearest Neighbor Method for Pattern Recognition [Research Frontier]. *IEEE Comput. Intell. Mag.* 10, 3 (2015), 52–60.

[45] Yuan Tian, David Lo, and Chengnian Sun. 2012. Information Retrieval Based Nearest Neighbor Classification for Fine-Grained Bug Severity Prediction. In *WCRE*. 215–224.

[46] Yao Tian, Tingyun Yan, Xi Zhao, Kai Huang, and Xiaofang Zhou. 2023. A Learned Index for Exact Similarity Search in Metric Spaces. *IEEE Trans. Knowl. Data Eng.* 35, 8 (2023), 7624–7638.

[47] Yao Tian, Ziyang Yue, Ruiyuan Zhang, Xi Zhao, Bolong Zheng, and Xiaofang Zhou. 2023. Approximate Nearest Neighbor Search in High Dimensional Vector Databases: Current Research and Future Directions. *IEEE Data Eng. Bull.* 46, 3 (2023), 39–54.

[48] Michel Verleysen and Damien François. 2005. The Curse of Dimensionality in Data Mining and Time Series Prediction. In *IWANN*. 758–770.

[49] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *MDM*. 569–574.

[50] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *PVLDB* 14, 11 (2021), 1964–1978.

[51] 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. *PVLDB* 13, 12 (2020), 3152–3165.

[52] Jiuqi Wei, Botao Peng, Xiaodong Lee, and Themis Palpanas. 2024. DET-LSH: A Locality-Sensitive Hashing Scheme with Dynamic Encoding Tree for Approximate Nearest Neighbor Search. *PVLDB* 17, 9 (2024), 2241–2254.

[53] Hailin Zhang, Penghao Zhao, Xupeng Miao, Yingxia Shao, Zirui Liu, Tong Yang, and Bin Cui. 2023. Experimental Analysis of Large-scale Learnable Vector Storage Compression. *PVLDB* 17, 4 (2023), 808–822.

[54] Wan-Lei Zhao, Hui Wang, Peng-Cheng Lin, and Chong-Wah Ngo. 2022. On the Merge of k-NN Graph. *IEEE Trans. Big Data* 8, 6 (2022), 1496–1510.

[55] Wan-Lei Zhao, Hui Wang, and Chong-Wah Ngo. 2022. Approximate k-NN Graph Construction: A Generic Online Approach. *IEEE Trans. Multim.* 24 (2022), 1909–1921.

[56] 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. *PVLDB* 16, 8 (2023), 1979–1991.