Advancements in Vector Indexing Techniques: KD-Trees, HNSW, and Product Quantization for Vector Databases

Syed Arham Akheel
Senior Solutions Architect
Bellevue, WA arhamakheel@yahoo.com

Abstract—The growth of high-dimensional data in fields like computer vision, natural language processing, and data mining has necessitated the development of efficient vector indexing techniques. This paper presents a comparative analysis of three state-of-the-art vector indexing techniques—KD-Trees, Hierarchical Navigable Small World (HNSW) graphs, and Product Quantization (PQ). We provide a detailed exploration of each method's algorithmic foundations, mathematical formulations, advantages, and limitations, followed by a discussion of recent advancements aimed at optimizing these methods for large-scale databases. Through my analysis, I aim to offer a comprehensive understanding of how these techniques perform and can be adapted for various real-world applications.

Index Terms—Vector Indexing, KD-Trees, HNSW, Product
Quantization, High-Dimensional Data, Nearest Neighbor Search

I. INTRODUCTION

The nearest neighbor search is a fundamental problem in many applications, including machine learning, image retrieval, and recommendation systems. The key goal is to efficiently find the data points that are closest to a given query in high-dimensional space. With the growing volume of high-dimensional data, efficient vector indexing techniques are critical to maintain low query times and high accuracy.

Nearest neighbor search can be formally defined as follows: given a set of n points $P = \{p_1, p_2, \ldots, p_n\}$ in a $d$dimensional space $\mathbb{R}^d$ and a query point q, the objective is to find the point $p_i \in P$ such that the distance $d(q,p_i)$ is minimized. However, this search becomes computationally expensive as the number of dimensions (d) increases, leading to the well-known problem called the *curse of dimensionality*. As the dimensionality increases, the efficiency of traditional search methods diminishes due to the sparsity of data in highdimensional space [8].

Traditional approaches to the nearest neighbor problem include methods such as KD-Trees, which use recursive partitioning to build a tree structure for efficient searching. KDTrees work well in low-dimensional spaces, as they divide the data space into hierarchical hyperplanes that allow efficient point location queries. However, their performance degrades significantly in higher dimensions, making them impractical for many real-world high-dimensional problems [2]. In particular, the efficiency of KD-Trees is compromised when the number of dimensions exceeds 20, as most of the data ends up near the boundaries of the partitions [1].

The limitation of KD-Trees in high dimensions has led to the development of more sophisticated vector indexing techniques such as Hierarchical Navigable Small World (HNSW) graphs and Product Quantization (PQ). These methods are designed to tackle the scalability and accuracy issues associated with high-dimensional data.

HNSW is a graph-based approach that builds on the concept of *navigable small-world networks*, which enable efficient nearest neighbor search by traversing a graph structure [3]. HNSW uses a multi-layer graph in which each node has connections to other nodes at multiple levels. The top layers provide a coarse overview of the data space, while the lower layers provide finer resolution for local searches. The search process in HNSW starts from a high level and proceeds layer-by-layer to more fine-grained graphs, ultimately reaching the nearest neighbors. This approach ensures logarithmic complexity scaling, making HNSW highly efficient for high-dimensional spaces. According to Malkov and Yashunin, HNSW is particularly advantageous for datasets where the data points are highly clustered, providing both high accuracy and low search latency [3]. Product Quantization (PQ), on the other hand, is a method that aims to reduce the memory and computational costs of nearest neighbor search by compressing highdimensional vectors into compact codes. PQ divides the original high-dimensional space into a Cartesian product of lowerdimensional subspaces, and each subspace is quantized separately [11]. This quantization results in a set of centroids for each subspace, and each vector is represented by a short code that indexes the closest centroid for each subspace. By doing so, the distance between a query vector and a database vector can be efficiently approximated by summing the distances between their corresponding subspace centroids [4]. Optimized Product Quantization further enhances this technique by minimizing the quantization distortion through careful space decomposition and codebook optimization, thereby improving the search accuracy for high-dimensional data [4].

The use of vector indexing techniques such as PQ and HNSW has found significant applications in areas such as image retrieval, natural language processing, and recommendation systems. For instance, PQ has been used extensively for indexing large image datasets, allowing for efficient approximate nearest neighbor (ANN) searches while significantly reducing memory usage [11]. HNSW, due to its dynamic and highly connected graph structure, has also been employed in semantic search applications, including vector similarity search for natural language data [7].

Recent studies have also highlighted the importance of similarity metrics in vector indexing. For instance, metrics such as Euclidean distance, cosine similarity, and Jaccard similarity are used to evaluate the closeness of vectors depending on the type of data and application [8]. Euclidean distance is widely used for continuous data, whereas cosine similarity is preferred for text data due to its focus on vector direction rather than magnitude. Euclidean distance is given by:

$$d(q,p) = \sqrt{\sum_{i=1}^{d}(q_i - p_i)^2}$$

The rest of this paper explores the mathematical and practical aspects of these vector indexing techniques, comparing their use cases, performance, and optimization strategies. Section II provides a detailed discussion of KD-Trees, including their algorithmic design, limitations, and recent improvements. Section III discusses HNSW graphs, explaining their hierarchical design and advantages for high-dimensional datasets. Section IV focuses on Product Quantization, covering its mathematical formulation and recent optimizations. Finally, we present a comparative analysis of these methods in Section V, followed by a conclusion summarizing their relative advantages and trade-offs.

II. VECTOR INDEXING TECHNIQUES

A. KD-Trees

KD-Trees are among the earliest and most well-known methods for spatial indexing, introduced by Bentley in 1975. They work by recursively partitioning the data space along different dimensions, creating a hierarchical data structure that facilitates efficient searches for low-dimensional data [2]. KDTrees are binary trees where each node represents a hyperplane in the feature space, allowing for efficient partitioning.

1) *Tree Construction:* The construction of a KD-Tree involves dividing the dataset along one dimension at a time. The algorithm selects a splitting dimension and a splitting value, typically choosing the median value along the current dimension to ensure a balanced tree. This ensures that roughly half of the points lie on either side of the split, which helps in maintaining the efficiency of the search operations [1].

Formally, let $P = \{p_1,p_2,...,p_n\}$ be a set of points in a d-dimensional space. To construct the KD-Tree: 1. Select a splitting dimension k (usually done in a round-robin fashion or by selecting the dimension with the highest variance). 2. Find the median value along the chosen dimension k. 3. Partition the points into two subsets: those with values less than or equal to the median on the left, and those greater than the median on the right. 4. Recursively apply the above steps to construct the left and right subtrees.

The resulting structure is a balanced tree where each node represents a splitting hyperplane perpendicular to the selected dimension, and each leaf node contains a small subset of points.

2) *Nodes and Search Mechanism:* Each node in the KDTree represents a point and a corresponding hyperplane that divides the data space. A KD-Tree node can be defined as:

Node = (p, left child, right child, k)

where p is the point stored at the node, left child and right child are the left and right subtrees, respectively, and k is the splitting dimension. The recursive nature of the KDTree construction ensures that each node splits the data space into progressively smaller hyper-rectangles.
To search for the nearest neighbor, the KD-Tree algorithm traverses the tree in a depth-first manner. It first follows the branch corresponding to the side of the hyperplane that contains the query point, q. Once a leaf node is reached, it backtracks to explore other branches if there is a possibility of finding a closer point. The process of backtracking is governed by the distance from the query point to the current best candidate and the distance to the splitting hyperplane.

3) *Nearest Neighbor Search:* The nearest neighbor search problem is solved by minimizing the distance between the query point q and the points stored in the KD-Tree. The Euclidean distance between a query point $q = (q_1, q_2, \ldots, q_d)$ and a point $p = (p_1, p_2, \ldots, p_d)$ where d represents the dimensionality of the space. The KD-Tree algorithm searches for the point p in the tree that minimizes this distance. During the search, the algorithm keeps track of the current best distance and prunes branches that cannot possibly contain a closer point.
The process involves the following steps:
Traversal to Leaf Node: The query point q is compared with the hyperplane at each node, and the branch corresponding to the side containing q is traversed first.
Backtracking: Upon reaching a leaf node, the current best distance is recorded. The algorithm then backtracks, checking whether other branches could potentially contain a closer point.
Bounding Box Pruning: The algorithm prunes branches by calculating the minimum possible distance from q to the bounding box of the unexplored branch. If this distance is greater than the current best distance, that branch is skipped.
In low-dimensional spaces, this approach is very efficient, as it limits the number of points that need to be evaluated explicitly. However, as the number of dimensions increases, the efficiency degrades due to the increased number of points lying near the boundaries of the hyperplanes, requiring more backtracking and reducing the benefits of the tree structure [2].

4) *Limitations and the Curse of Dimensionality:* The primary limitation of KD-Trees is their poor performance in high-dimensional spaces, a phenomenon known as the *curse of dimensionality*. As the dimensionality d increases, the volume of the space grows exponentially, and the data points become sparse. Consequently, the splitting hyperplanes lose their effectiveness in partitioning the space, resulting in more backtracking during the nearest neighbor search. This, in turn, leads to performance that is often no better than a brute-force search [1], [8].
Empirical studies have shown that KD-Trees work well for dimensions up to about 20. Beyond this threshold, the efficiency of the tree structure declines rapidly, and the cost of traversal approaches that of a linear scan [2]. To address these challenges, recent research has explored approximate nearest neighbor techniques that adapt KD-Trees with random rotations or combine them with other algorithms to improve efficiency [3].
The nearest neighbor of a query point q is found by minimizing the Euclidean distance. The challenge in higher dimensions is that the probability of a point being close to q decreases significantly due to the sparsity of points. The hyperrectangles defined by the KD-Tree splits do not effectively capture the nearest neighbors as the dimensionality increases, leading to inefficient pruning and high computational costs [1],
[2].
Recent improvements in KD-Tree-based methods include the use of random rotations before constructing the tree. These rotations help in balancing the splits and reducing the alignment of data points with the axes, thus improving the accuracy of nearest neighbor searches in higher dimensions [3]. Additionally, ensembles of KD-Trees are also used to provide better results by combining the outputs of multiple independently constructed trees.
In summary, KD-Trees provide an efficient method for nearest neighbor search in low-dimensional spaces by partitioning the data space into smaller, manageable regions. However, their efficiency diminishes rapidly as the dimensionality increases due to the curse of dimensionality. As a result, KDTrees are often combined

with other techniques or used in approximations to maintain performance in higher dimensions.

B. Hierarchical Navigable Small World Graphs (HNSW)

HNSW graphs are graph-based indexing structures designed to overcome the inefficiencies of KD-Trees in highdimensional data. They are based on the concept of navigable small world networks, enabling efficient nearest neighbor searches by traversing layers of graphs [3].

1) Navigable Small World Networks: The concept of navigable small world networks is central to the functioning of HNSW. A small world network is characterized by the presence of short paths between nodes, even in a large graph, and a clustering of nodes into locally dense regions. In HNSW, this concept is extended to create a graph where nodes are connected in such a way that the nearest neighbor search can be performed efficiently by leveraging these short paths [3]. The efficiency of a small world network is derived from its ability to balance global and local connectivity, allowing a search to proceed from a random starting point to the desired node through a combination of long-range and short-range links.

In an HNSW, nodes are organized in a hierarchical manner, with nodes in higher layers having fewer connections, representing more global and coarse-grained relationships, while nodes in lower layers have more connections, representing fine-grained relationships. This multi-layered approach provides a natural way to navigate from a general overview of the data to a precise local neighborhood, significantly reducing search times compared to flat graph structures.

2) Algorithm Overview: HNSW builds a hierarchical multilayer graph structure, where nodes in higher layers have fewer connections, providing an overview of the data. The search starts from the top layer and moves downward, progressively narrowing the search to more fine-grained layers. Each layer of the HNSW structure is essentially a proximity graph where nodes are connected based on their similarity, and edges represent the closeness of nodes [3]. The search process starts at the topmost layer, which has the sparsest graph, and moves down the hierarchy, using each subsequent layer for more precise search results.

The construction of an HNSW graph begins by placing each new element into a random layer, determined by an exponentially decaying distribution, ensuring that higher layers have fewer nodes. The connections for each element are established by finding the closest nodes already present in the graph. This creates a structure where the number of neighbors for each node follows a power law, which is critical for ensuring efficient traversal during search operations.

3) Layered Proximity Graphs: The HNSW structure uses layered proximity graphs to efficiently navigate through the data space. Each layer can be seen as a different level of granularity for exploring the graph, with higher layers providing broader and less connected views, and lower layers offering more detailed and densely connected views. The goal is to strike a balance between exploration and exploitation during the search. At each level, the algorithm performs a greedy search by moving to the neighbor that minimizes the distance to the query point [3]. The mathematical formulation for the greedy search in HNSW is where q is the query point, n_i is the current node, and $N(i)$ is the set of neighbors of node i. The greedy approach continues until no neighbor is found that is closer to the query point than the current node.

This process is repeated for each layer, starting from the top and moving to the bottom layer. The transition between layers ensures that the search starts from a global perspective and gradually focuses on a local neighborhood, which significantly improves the overall search efficiency. The multi-layer structure allows HNSW to achieve logarithmic complexity scaling with respect to the number of data points, which is a considerable improvement over traditional flat graph-based methods [3].

The HNSW search algorithm starts from an entry node, often chosen randomly or as a previously known approximate neighbor, and proceeds iteratively to move to the next nearest neighbor by comparing distances. The distance function used is typically the Euclidean distance where q is the query point and p is a candidate point. The algorithm's objective is to minimize this distance iteratively until no closer neighbor can be found.

In the hierarchical structure, the search at each level aims to refine the candidate set of neighbors by exploring the connections available at that level. The top-level graph has fewer nodes and connections, which helps in quickly narrowing down the search area, while the lower levels, which are more densely connected, allow for precise identification of the nearest neighbors. The hierarchical search strategy ensures that the algorithm can quickly converge to the optimal solution with high probability [3].

4) *Advantages and Limitations:* HNSW has demonstrated superior performance in both high-dimensional and clustered datasets due to its graph traversal capabilities. The hierarchical nature of the graph ensures that the search starts from a coarse resolution and progressively refines the search at each layer, leading to efficient and accurate results [3]. Moreover, the navigable small world property of the graph allows for efficient connectivity, enabling searches to quickly locate the target neighborhood even in large-scale datasets.

However, building the HNSW graph is computationally expensive and requires substantial memory overhead for storing the multi-layer connections. The cost of maintaining the multilevel structure and the additional memory needed for storing neighbors at each level are some of the primary challenges of this approach. Despite these challenges, the benefits of faster and more accurate search make HNSW a preferred choice for many high-dimensional search problems, particularly those involving dense and complex data distributions [3].

C. Product Quantization (PQ)

Product Quantization is a powerful technique for encoding high-dimensional vectors into compact codes to enable efficient nearest neighbor search. PQ divides the original high dimensional space into smaller subspaces, which are then quantized separately [11]. This approach drastically reduces both memory consumption and computational costs while still maintaining a high level of accuracy for similarity searches.

1) *Vector Decomposition:* The first step in Product Quantization involves decomposing the original vector into multiple sub-vectors. Given a vector $x \in \mathrm{R}^D$, PQ divides it into M sub-vectors, each of dimension D/M. Mathematically, this decomposition can be expressed as:

$$x = (x_1, x_2, \ldots, x_M)$$

where $x_i \in \mathrm{R}^{D/M}$ for $i = 1,2,\ldots,M$. The rationale behind this decomposition is to reduce the dimensionality of each sub-vector, making it easier and computationally cheaper to handle [11]. Each sub-vector is treated independently, allowing for quantization to be performed on smaller, more manageable components of the original vector.

2) *Quantization of Subspaces:* Once the vector is decomposed, each subspace is quantized independently using a set of predefined centroids. For each subspace, a codebook C_i is created that contains K centroids:

$$Ci = \{ci,1, ci,2, \ldots, ci,K\}, \quad ci,j \in \mathrm{R}D/M$$

where K is the number of clusters or centroids used for quantization in each subspace. The quantization function Q_i assigns each sub-vector x_i to its closest centroid in the codebook C_i:

$$Q_i(x_i) = \arg\min \|x_i - c\|_2$$
$$c \in C_i$$

This process is repeated for each of the M sub-vectors, resulting in a quantized representation of the original vector in terms of centroids from each subspace [11]. The goal is to minimize the quantization error, which is the discrepancy between the original sub-vector and its nearest centroid.

3) *Compact Representation:* The quantized representation of each sub-vector allows the entire vector to be stored as a series of indices pointing to the centroids in each codebook. Instead of storing the original D-dimensional vector, PQ stores M indices, each representing the nearest centroid in the corresponding subspace. Thus, the compact representation of vector x can be described as:

$$Q(x) = (Q_1(x_1), Q_2(x_2), \ldots, Q_M(x_M))$$

This compact representation significantly reduces memory usage. If K centroids are used per subspace, each index can be stored using $\log_2(K)$ bits, leading to substantial memory savings compared to storing full-precision floating-point values for each dimension [11].

4) *Distance Computation:* One of the key features of Product Quantization is the ability to efficiently compute approximate distances between a query vector and database vectors using their quantized representations. Given a query vector $q \in \mathrm{R}^D$ and a database vector x, the distance between them is

approximated by summing the distances between their corresponding sub-vectors:

M

$d(q,x) \approx X\|qi - ci,Qi(xi)\|2$

$i=1$

where $c_{i,Qi(xi)}$ represents the centroid in the i-th subspace that x_i is assigned to. This approximation leverages the precomputed distances between the query sub-vector q_i and the centroids in the codebook C_i, thus avoiding the need to compute the full distance directly for every dimension. This allows PQ to perform efficient nearest neighbor searches even in large datasets [4], [11].

5) *Encoding:* The encoding phase of PQ involves assigning each sub-vector x_i of the original vector x to its closest centroid in the corresponding codebook C_i. The resulting indices are stored to represent the compressed version of the vector. The encoding process can be formally expressed as: $x \longrightarrow (e_1,e_2,...,e_M)$, $e_i = Q_i(x_i)$

where e_i is the index of the centroid closest to x_i in codebook C_i. This encoded representation enables efficient storage and retrieval, as the vector is now represented by a series of indices rather than raw floating-point numbers.

6) *Optimized Product Quantization:* Recent advancements in PQ have focused on optimizing both the decomposition of the vector space and the quantization process to minimize quantization error. Optimized Product Quantization (OPQ) involves rotating the original data vectors before decomposition, which helps to better align the subspaces with the inherent data structure, thereby reducing quantization distortion [4]. Mathematically, this can be described as applying a rotation matrix $R \in \mathrm{R}^{D\times D}$ to the original vector:

$x' = R \cdot x$

where x' is the rotated version of the original vector x. This rotation is followed by the usual decomposition and quantization steps, but with improved alignment that leads to lower distortion. The iterative optimization of both the rotation matrix and the codebooks ensures that the quantized representation is as close as possible to the original data, enhancing both search accuracy and retrieval performance.

Optimized PQ has been shown to significantly improve the trade-off between accuracy and memory usage, making it particularly suitable for applications like image and video retrieval where high-dimensional vectors must be indexed efficiently [4], [11].

In summary, Product Quantization is a highly effective method for compressing high-dimensional vectors and enabling efficient nearest neighbor search. By decomposing the vector into subspaces, quantizing each subspace independently, and storing compact representations, PQ achieves a significant reduction in both memory usage and computational complexity while maintaining a reasonable level of accuracy. The advancements in optimized PQ further enhance its effectiveness, making it a preferred choice for large-scale similarity search applications.

III. COMPARATIVE ANALYSIS

A. Performance Metrics

The comparison of KD-Trees, HNSW, and Product Quantization (PQ) can be discussed based on several key performance metrics, such as accuracy, scalability, and memory usage. Each method has strengths and weaknesses that determine its suitability for specific applications.

In terms of accuracy, PQ often yields the highest accuracy among the three methods due to its compact representation and the ability to approximate nearest neighbors with low quantization error. PQ works by splitting vectors into subspaces and quantizing them separately, which reduces the loss of information, allowing for higher recall rates when compared to KD-Trees [11]. HNSW also achieves high recall, particularly for high-dimensional datasets, due to its layered graph structure, which efficiently narrows down the search space by combining coarse and fine levels of search [3]. On the other hand, KD-Trees tend to struggle with maintaining accuracy as dimensionality increases, which is attributed to the curse of dimensionality [2].

HNSW is considered the most scalable due to its graph based structure, which allows it to handle large, high dimensional datasets effectively. The logarithmic complexity scaling of HNSW makes it an excellent choice for datasets with millions of elements, enabling fast search without significantly compromising on recall [3]. In contrast, KD-Trees become less efficient as the number of dimensions increases, with performance degrading rapidly beyond 20 dimensions. The hyper-rectangles defined by KD-Tree splits lose their efficacy, leading to extensive backtracking and diminishing scalability [1]. PQ also offers scalability benefits in terms of storage, as it compresses the data into compact representations. However, the speed of search can be affected depending on the number of centroids and the dimensionality of each subspace, which requires careful tuning to maintain efficiency [4].

Memory usage is an essential consideration when dealing with large-scale datasets. KD-Trees are relatively lightweight in terms of memory requirements, especially for lowdimensional data, as they primarily rely on a hierarchical structure to partition the data space [2]. However, the inefficiencies in high-dimensional spaces result in a lack of practical utility for KD-Trees in such scenarios. HNSW requires significant memory to store the multi-layered graph and multiple connections per node, which can be a bottleneck for very large datasets. The number of edges maintained at each layer grows proportionally, leading to increased memory consumption [3]. PQ, in contrast, offers a favorable tradeoff between memory usage and accuracy. By storing compact codes instead of raw vectors, PQ allows for significant memory savings, which is particularly advantageous for applications like image retrieval, where high-dimensional vectors must be stored efficiently [11].

B. Practical Considerations

Each of the three methods has practical use cases that best suit its strengths:

KD-Trees are most effective in low-dimensional spaces, typically where the dimensionality is less than 20. Their ability to efficiently partition the data space allows for fast exact nearest neighbor searches in such settings. For example, KD-Trees are widely used in geographic information systems (GIS) for spatial data indexing and nearest neighbor searches [2]. However, in high-dimensional settings, their inefficiency makes them less suitable, as backtracking becomes more prevalent and performance degrades.

HNSW is ideal for high-dimensional spaces and applications requiring high recall and fast retrieval times, such as natural language processing (NLP) tasks and semantic search. The multi-layer graph approach ensures that searches can start from a broad perspective and zoom into specific clusters efficiently, making HNSW highly effective for recommendation systems, image search, and even social media data indexing, where the data is high-dimensional and clustered [3], [7]. The trade-off comes in the form of memory usage and the computational cost of constructing the graph, but these are often justified by the performance benefits in recall and speed.

Product Quantization is particularly well-suited for applications like image and video retrieval, where large datasets need to be queried with limited memory. The compact codes generated by PQ allow for efficient storage of massive datasets, reducing memory requirements significantly. PQ is also advantageous in scenarios where approximate nearest neighbor search is acceptable, and some loss of precision is tolerable in exchange for faster search and reduced memory footprint [5], [11]. Additionally, the optimization techniques introduced in Optimized PQ (OPQ) have further improved PQ's accuracy, making it a robust choice for large-scale similarity search applications [4].

C. Summary

KD-Trees are suitable for low-dimensional, exact searches, while PQ and HNSW are more suited for high-dimensional, approximate searches, with HNSW typically providing high recall and PQ providing the best trade-off between compression and accuracy. HNSW scales effectively with the size of high-dimensional datasets due to its hierarchical graph structure. KD-Trees struggle with scalability in high-dimensional spaces, and PQ requires tuning but can achieve good scalability due to compact encoding. KD-Trees have lower memory requirements for low-dimensional data but become inefficient in higher dimensions. HNSW requires substantial memory for multi-layer graph connections, whereas PQ optimizes memory usage by storing vectors as compact codes.

The choice of technique depends on the specific requirements of the application, including dimensionality, need for accuracy, speed, and memory constraints. HNSW is preferred when high recall and speed are required, PQ is ideal for memory efficiency, and KD-Trees work best for lowdimensional, exact searches. These distinctions help guide the selection of the appropriate indexing technique for a given use case, ensuring an optimal balance between performance and resource utilization [3], [4], [11].

IV. CONCLUSION

KD-Trees, HNSW, and PQ represent three distinct approaches to solving the nearest neighbor search problem for high-dimensional data. Each method offers unique advantages and trade-offs in terms of accuracy, scalability, and memory usage.

KD-Trees are among the earliest indexing structures developed for nearest neighbor search, and they perform well in low-dimensional spaces due to their efficient recursive partitioning. However, as dimensionality increases, KD-Trees suffer from the curse of dimensionality, leading to excessive backtracking and decreased efficiency. As a result, KD-Trees are often considered suitable only for applications where the dimensionality is moderate, and exact searches are required [1], [2]. The simplicity of KD-Trees and their relatively low memory footprint make them a good choice for small scale, low-dimensional problems, but they are limited by their inability to handle the complexity of high-dimensional data effectively.

Hierarchical Navigable Small World (HNSW) graphs, on the other hand, address the limitations of KD-Trees by introducing a hierarchical, graph-based structure that can efficiently manage high-dimensional data. HNSW leverages the concept of navigable small world networks to create a multilayered graph, where each layer progressively narrows the search space [3]. This hierarchical nature allows HNSW to provide high recall rates with logarithmic complexity, making it suitable for large-scale, high-dimensional datasets such as those used in natural language processing, image retrieval, and recommendation systems. The scalability of HNSW, combined with its high accuracy, makes it a preferred choice for real time applications requiring dynamic data updates and efficient retrieval. However, the computational cost of constructing the graph and the memory overhead associated with storing multilayer connections can be significant, particularly for very large datasets [3], [7].

Product Quantization (PQ) offers a different approach by focusing on reducing the storage requirements for high dimensional data. PQ compresses vectors by decomposing them into subspaces and quantizing each subspace independently, resulting in a compact representation that is well suited for approximate nearest neighbor searches [11]. PQ provides an efficient trade-off between memory usage and search accuracy, making it highly effective for applications where large datasets need to be stored in limited memory environments, such as image and video retrieval systems. The use of compact codes allows for efficient similarity searches without requiring the full precision of the original vectors, which can be beneficial for both storage and computational efficiency. Furthermore, optimized versions of PQ, such as Optimized Product Quantization (OPQ), have improved its effectiveness by minimizing quantization error through space rotation and better alignment with the data structure [4].

In comparing these three methods, it is clear that each has specific strengths that make it suitable for different scenarios. KD-Trees excel in low-dimensional applications but struggle with scalability as dimensionality increases. HNSW provides an efficient solution for high-dimensional, large-scale datasets by using a hierarchical approach that balances accuracy and search time. Meanwhile, PQ is particularly advantageous in scenarios where memory efficiency is critical, and approximate results are acceptable.

The choice of the most appropriate indexing technique depends on the requirements of the specific application. For example, KD-Trees may be suitable for geographic information systems (GIS) or robotics, where the data is lowdimensional and exact matches are crucial [2]. HNSW is ideal for high-dimensional, real-time applications like recommendation systems and semantic search, where high recall and speed are essential [3], [7]. PQ, with its efficient use of memory, is highly applicable to multimedia retrieval systems and scenarios involving large-scale vector databases that require approximate nearest neighbor searches to balance storage and retrieval accuracy [5], [11].

In conclusion, KD-Trees, HNSW, and PQ provide diverse solutions for the nearest neighbor search problem, each tailored to different dimensionalities, accuracy requirements, and resource constraints. Understanding these differences is crucial for selecting the right tool for a given use case, ensuring optimal performance in terms of both computational and storage efficiency. The advancements in HNSW and PQ demonstrate that innovative approaches to vector indexing can significantly enhance the ability to manage high-dimensional data, ultimately enabling faster and more accurate data retrieval in a wide range of applications [3], [4].

REFERENCES

[1] R. Panigrahy, ”An Improved Algorithm Finding Nearest Neighbor Using KD-Trees,” LATIN 2008, Springer, 2008.

[2] P. Ram, K. Sinha, ”Revisiting KD-Tree for Nearest Neighbor Search,” ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD ’19), 2019.

[3] Y. A. Malkov, D. A. Yashunin, ”Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs,” IEEE Transactions, 2018.

[4] T. Ge, K. He, Q. Ke, J. Sun, ”Optimized Product Quantization for Approximate Nearest Neighbor Search,” IEEE CVPR, 2013.

[5] Y. Chen, T. Guan, C. Wang, ”Approximate Nearest Neighbor Search by Residual Vector Quantization,” Sensors, 2010.

[6] J. P. V. Pinheiro, L. R. Borges, B. F. M. da Silva, L. A. P. Paes Leme, M.
A. Casanova, ”Indexing High-Dimensional Vector Streams,” ICEIS 2023.

[7] J. Vita, J. Bishop, L. Kyada, N. Raines, S. Mehta, ”Semantic Vector Search for Twitter Data with HNSW Retrieval,” 2023.

[8] F. Dayton, ”Vector Similarity Search: A Deeper Dive,” CS168 Final Report, 2023.

[9] Y. Han, C. Liu, P. Wang, ”A Comprehensive Survey on Vector Database: Storage and Retrieval Technique, Challenge,” arXiv preprint arXiv:2310.11703, 2023.

[10] J. J. Pan, J. Wang, G. Li, ”Survey of Vector Database Management Systems,” arXiv preprint arXiv:2310.14021, 2023.

[11] H. Jegou, M. Douze, and C. Schmid, ”Product Quantization for´ Nearest Neighbor Search,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 1, pp. 117-128, Jan. 2011, doi: 10.1109/TPAMI.2010.57.

[12] Zilliz, ”Milvus: A Distributed and High-Performance Vector Database,” White Paper, 2020. [Online]. Available: https://milvus.io.

[13] J. Johnson, M. Douze, and H. Jegou, ”Billion-scale similarity search´ with GPUs,” *IEEE Transactions on Big Data*, 2019, doi: 10.1109/TBDATA.2019.2921572.

[14] P. Indyk and R. Motwani, ”Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality,” *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC)*, 1998, pp. 604-613, doi: 10.1145/276698.276876.