SymphonyQG: Towards Symphonious Integration of Quantization and Graph for Approximate Nearest Neighbor Search

YUTONG GOU[†], Nanyang Technological University, Singapore
JIANYANG GAO[†], Nanyang Technological University, Singapore
YUEXUAN XU, Nanyang Technological University, Singapore
CHENG LONG[*], Nanyang Technological University, Singapore

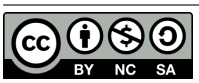
Approximate nearest neighbor (ANN) search in high-dimensional Euclidean space has a broad range of applications. Among existing ANN algorithms, graph-based methods have shown superior performance in terms of the time-accuracy trade-off. However, they face performance bottlenecks due to the random memory accesses caused by the searching process on the graph indices and the costs of computing exact distances to guide the searching process. To relieve the bottlenecks, a recent method named NGT-QG makes an attempt by integrating quantization and graph. It (1) replicates and stores the quantization codes of a vertex's neighbors compactly so that they can be accessed sequentially, and (2) uses a SIMD-based implementation named FastScan to efficiently estimate distances based on the quantization codes in batch for guiding the searching process. While NGT-QG achieves promising improvements over the vanilla graph-based methods, it has not fully unleashed the potential of integrating quantization and graph. For instance, it entails a re-ranking step to compute exact distances at the end, which introduces extra random memory accesses; its graph structure is not jointly designed considering the in-batch nature of FastScan, which causes wastes of computation in searching. In this work, following NGT-QG, we present a new method named SymphonyQG, which achieves more symphonious integration of quantization and graph (e.g., it avoids the explicit re-ranking step and refines the graph structure to be more aligned with FastScan). Based on extensive experiments on real-world datasets, SymphonyQG establishes the new state-of-the-art in terms of the time-accuracy trade-off: at 95% recall, SymphonyQG achieves 1.5x-4.5x QPS compared with the most competitive baselines and achieves 3.5x-17x QPS compared with the classical library HNSWlib across all tested datasets. At the same time, its indexing is at least 8x faster than NGT-QG.

CCS Concepts: • **Information systems → Information retrieval query processing**; • **Theory of computation → Data structures design and analysis**.

Additional Key Words and Phrases: Approximate Nearest Neighbor Search, High-Dimensional Vector, Vector Quantization

ACM Reference Format:
Yutong Gou[†], Jianyang Gao[†], Yuexuan Xu, and Cheng Long[*]. 2025. SymphonyQG: Towards Symphonious Integration of Quantization and Graph for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 3, 1 (SIGMOD), Article 80 (February 2025), 26 pages. https://doi.org/10.1145/3709730

[†]Equally contributed to this work.
[*]Corresponding author.
Authors' addresses: Yutong Gou[†], Nanyang Technological University, Singapore, yutong003@e.ntu.edu.sg; Jianyang Gao[†], Nanyang Technological University, Singapore, jianyang.gao@ntu.edu.sg; Yuexuan Xu, Nanyang Technological University, Singapore, yuexuan001@e.ntu.edu.sg; Cheng Long[*], Nanyang Technological University, Singapore, c.long@ntu.edu.sg.



ACM 2836-6573/2025/2-ART80
https://doi.org/10.1145/3709730

1 INTRODUCTION

Nearest neighbor search on high-dimensional vector data is a fundamental problem and has many real-world applications, including databases [15, 59, 64, 78], information retrieval [7, 51, 59, 70], large language models [14, 52–54] and recommendation system [71, 80, 83]. Since the cost of searching for exact nearest neighbors is expensive when the database of vectors is large, researchers and practitioners usually aim to develop algorithms for approximate nearest neighbors (ANN) search, so as to achieve a better accuracy-efficiency trade-off.

Many methods have been developed for ANN query [4, 6, 10–13, 16, 18, 19, 21, 22, 30–33, 36, 38–44, 48, 49, 57, 58, 67, 74–76, 82, 84] (a more detailed review will be presented in Section 5). Among them, graph-based methods show superior search performance, which has been widely observed in benchmarks and competitions [8, 23, 56, 62, 63, 79]. In these methods, each vector in the database is viewed as a vertex in the graph. During indexing, these methods build a graph-based index by adding edges among the vertices. When a query comes, these methods usually adopt a greedy beam search algorithm [31, 32, 48, 58] to find NN (more details will be provided in Section 2.1).

Nevertheless, graph-based methods usually face performance bottlenecks in both memory and CPU during querying. To be more specific, in each iteration of the search process, these methods require to access raw data vectors of the current vertex's neighbors in order to compute their exact distances from the query vector [30, 32, 48, 58, 67]. This process is costly because (1) it would incur many random memory accesses since these vectors are stored separately in memory (please refer to Figure 2(a)), which is unfriendly to cache, and (2) computing exact distances between high-dimensional vectors frequently is expensive.

To boost the search performance of vanilla graph-based indices, a method called NGT-QG is proposed in the open-source library NGT by Yahoo Japan [47]. NGT-QG proposes a novel framework, namely *QG*, by integrating *quantization* [49] and a graph-based index for reducing the aforementioned two types of cost as follows. (1) To mitigate random memory accesses, NGT-QG compresses raw vectors into short codes (namely quantization codes) using product quantization (PQ) [49]. And for each vertex, it duplicates and stores the quantization codes of its neighbors compactly on its side in main memory (see Figure 1). Recall that during the search process of vanilla graph-based methods, the algorithm uses raw data vectors to compute exact distances between the current vertex's neighbors and the query. In NGT-QG, this process is replaced by using the quantization codes of the vertex's neighbors for estimating their distances. Since these quantization codes are stored compactly, the memory is accessed in a sequential pattern, thus avoiding random memory accesses in each search iteration. (2) To avoid the costs of computing the exact distances, NGT-QG estimates distances using FastScan [4, 5]. Specifically, it packs every 32 quantization codes in a batch (i.e., the compactly stored quantization codes) and re-organizes the data layout. In this way, during searching, it can estimate distances for 32 data vectors simultaneously via a series of SIMD-based operations, which has been shown to be highly efficient [4]. According to the well-acknowledged ANN-Benchmark [8], NGT-QG is one of the algorithms that achieve the state-of-the-art performance in terms of time-accuracy trade-off. For example, when reaching 95% recall on the SIFT dataset, it has been reported to be 2.4x faster than the classical open-source library HNSWlib [85].

Despite the promising search performance of NGT-QG, the potential of integrating quantization and graph-based indices has not been fully unleashed, and we observe the following limitations of NGT-QG in querying and indexing. **Querying**: The limitation in querying lies in its quantization method and search algorithm. Specifically, the quantization method used in NGT-QG (i.e., PQ) has no guarantee on the accuracy of its estimated distances and has been observed to fail disastrously on some real-world datasets [35] (also observed in our experiments in Section 4). This is highly

undesirable because ANN is used as an infrastructure in support of various downstream applications. The disastrous failure of ANN could cause unforeseen accidents in production environments. As for the search algorithm, recall that NGT-QG uses estimated distances when searching on a graph. This entails a re-ranking step to achieve high recall. To be more specific, at the end of the search process, it requires to compute exact distances of selected candidates (i.e., vertices with small estimated distances) with their raw data vectors. Subsequently, candidates with the smallest exact distances are returned as the final result for the given query. The re-ranking process introduces extra random memory accesses since it needs to access raw data vectors that are not stored compactly.

Indexing: The limitation in indexing lies in the efficiency of indexing and the graph structure. On the one hand, NGT-QG only uses FastScan for accelerating the search algorithm and still incurs long time for indexing. On the other hand, the graph structure is not fully aligned with the search algorithm. Recall that NGT-QG adopts FastScan to estimate distances for each vertex's neighbors, and FastScan always estimates a batch of distances simultaneously [5] by SIMD-based operations. When the number of a vertex's neighbors is not a multiple of the batch size, FastScan would estimate a non-full batch of distances and waste some computation that could have been used for estimating distances for more vectors. For example, in the graph of NGT-QG on Deep-1M dataset [1], about 23% of vertices have non-full batches. In summary, the limitation of NGT-QG includes both the issues of its quantization technique and the issue that graph-based indices and quantization have not been fully symphoniously integrated.

In this paper, following NGT-QG, we develop a new method named SymphonyQG which targets more symphonious integration of a superior quantization technique and graph-based indices for ANN query. Specifically, SymphonyQG is built upon four major ideas. (1) We incorporate the RaBitQ [35] quantization method, which is recently available in the literature, to the scheme of searching graph-based indices with FastScan. RaBitQ is superior over PQ in that it provides an unbiased estimator with rigorous error bounds and is shown to have better practical performance [35]. Nevertheless, RaBitQ is originally used with IVF [35] and has not been used with graphs. We solve some technical challenges such as those in normalizing the vectors and in efficiently estimating distances with FastScan when searching on graphs. (2) We jointly design a new search algorithm and a specialized data layout to further reduce random memory accesses. Specifically, our method stores the raw data vectors compactly with the data needed in graph-based searching (e.g., the neighbors and their quantization codes). When the method visits a vertex during searching, it computes its exact distances based on the raw vectors and updates the NN (i.e., implicitly conducting re-ranking). This removes the explicit re-ranking step which incurs random memory accesses. In addition, a new search algorithm is designed, which uses multiple estimated distances to guide searching so that the true NN is less likely to be missed in the aforementioned implicit re-ranking. (3) We propose to leverage FastScan to accelerate the process of building a graph-based index. Specifically, we build a graph-based index by starting with an initialized graph and then adjusting the graph iteratively. In each iteration, we use our search algorithm (which is equipped with FastScan) for finding nearest neighbors of a vertex as candidates to be linked from the vertex in the graph. According to our experimental results, our index construction method is at least 8x faster than NGT-QG on real-world datasets. (4) We propose a graph refinement strategy which supplements edges for the graph so that the graph-based index surely has the out-degree of every vertex to be a multiple of the batch size. Thus, the refined graph is better aligned with FastScan and the in-batch computation of FastScan is fully utilized.

In summary, this paper makes the following contributions:

[1]We build the index of NGT-QG according to parameters provided in ANN-Bechmarks [8]. More details can be found in Section 4.

(1) **Novel ANN Algorithm**. We propose SymphonyQG, an algorithm which achieves more symphonious integration of quantization and graph-based indices. Compared with NGT-QG, our symphonious integration brings superior and more stable performance on ANN query and requires significantly shorter time in indexing.
(2) **Open-Source Library** [2]. Based on SymphonyQG, we provide a well-optimized open-source library for ANN query. The library provides both header-only implementations in C++ and convenient APIs in Python. Therefore, the research community and practitioners can easily evaluate and utilize our library with a few lines of Python codes.
(3) **Extensive Experiments**. We conduct extensive empirical studies on real-world datasets. The results verify that SymphonyQG establishes the new state-of-the-art in terms of the time-accuracy trade-off. At 95% recall ($K = 10$), SymphonyQG achieves 1.5x-4.5x QPS compared with the most competitive baselines and achieves 3.5x-17x QPS compared with the classical library HNSWlib across all tested datasets. The experiments conducted on two datasets with 100 million vectors verify its scalability. The ablation studies verify the effectiveness of each individual proposed technique.

The remainder of this paper is organized as follows. Section 2 reviews graph-based ANN methods, FastScan and RaBitQ. Section 3 presents the SymphonyQG method. Section 4 reports extensive experimental results on real-world datasets. Section 5 discusses related work. Section 6 provides conclusions and discussions.

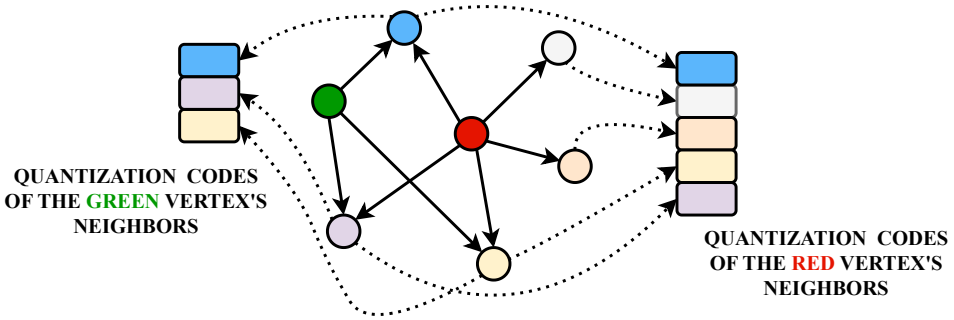


Fig. 1. An illustration of data layout. As illustrated, quantization codes of the red vertex's neighbors are packed and stored together at the red vertex's side. The same applies to the green vertex.

2 PRELIMINARIES

In this section, we briefly introduce the ANN query and the general workflow of graph-based methods in Section 2.1. To make the paper self-contained, we present a quick revisit to RaBitQ and FastScan in Section 2.2.

2.1 ANN and Graph-based Methods

Approximate Nearest Neighbor Search. Given a database in which each object is a high-dimensional vector, nearest neighbor (NN) query targets to find the vector that has the smallest distance to a query vector. In practice, the latency of finding exact NN is unacceptable due to the increasing size of the database and curse of dimensionality [43]. Thus, researchers have developed approximate nearest neighbor (ANN) search algorithms for achieving better trade-offs between

[2]https://github.com/gouyt13/SymphonyQG

accuracy and efficiency. In this paper, we focus on in-memory ANN query, where data vectors and indices are all stored in RAM. Besides, ANN query often targets to find K approximate nearest neighbors. When describing algorithms, we omit K for the convenience of presentation, while all methods developed in this paper are applicable to a general K.

Graph-based ANN Algorithms. Many graph-based indices have been proposed for ANN query [10, 18, 19, 30–32, 41, 44, 48, 56–58, 67, 76, 79]. These methods produce the state-of-the-art search performance because they can usually find nearest neighbors by considering only a small number of vectors as candidates through graph-based searching [61]. Specifically, before querying, these methods build a graph on the database, where each data vector corresponds to a vertex in the graph. When a query comes, an algorithm named greedy beam search [31, 32, 48, 58] is usually adopted. During searching, the algorithm maintains a set named the beam set which has the size named the beam size of up to n_b. The search process starts by adding an entry point to the beam set and proceeds iteratively. In each iteration, the algorithm finds the vector which is nearest to the query vector in the beam set and has not been visited so far. Then, the algorithm marks it as visited and computes the exact distances between the query vector and all its neighbors on the graph. The neighbors are next inserted into the beam set. If the beam set has more than n_b objects after insertion, then only the nearest n_b vectors would be kept. The algorithm terminates when all n_b vectors in the beam set are marked as visited. The nearest vector in the beam set is then returned as the answer. It is clear that a larger beam size indicates that more vectors would be visited during searching, and thus, corresponds to higher accuracy and larger latency.

2.2 RaBitQ and FastScan

Quantization is a family of methods which compress high-dimensional vectors into short quantization codes [4, 11, 12, 35, 36, 39, 49]. During querying, they can utilize the short quantization code of a data vector to estimate the distance between the data vector and a query vector. A recent study [35] proposes a new quantization method called RaBitQ. Unlike PQ and its variants [4, 11, 12, 36, 39, 49], RaBitQ provides an unbiased estimator with a theoretical error bound for the distances. In addition, it can use shorter quantization codes to produce consistently better empirical accuracy than PQ and its variants on real-world datasets. Thus, we consider using RaBitQ in our method.

Specifically, RaBitQ first reduces the question of estimating distances between two vectors to that of estimating inner product between their normalized vectors, which is restated in the following equations.

$$\|\mathbf{o}_r - \mathbf{q}_r\|^2 = \|(\mathbf{o}_r - \mathbf{c}) - (\mathbf{q}_r - \mathbf{c})\|^2 \tag{1}$$

$$= \|\mathbf{o}_r - \mathbf{c}\|^2 + \|\mathbf{q}_r - \mathbf{c}\|^2 - 2\|\mathbf{o}_r - \mathbf{c}\| \cdot \|\mathbf{q}_r - \mathbf{c}\| \cdot \langle \mathbf{o}, \mathbf{q} \rangle \tag{2}$$

Here, $\mathbf{o}_r$ and $\mathbf{q}_r$ denote the data vector and the query vector, respectively; $\mathbf{c}$ denotes a vector used for normalizing the data vectors; $\mathbf{q} := \frac{\mathbf{q}_r - \mathbf{c}}{\|\mathbf{q}_r - \mathbf{c}\|}$ denotes the normalized query vector; and $\mathbf{o} := \frac{\mathbf{o}_r - \mathbf{c}}{\|\mathbf{o}_r - \mathbf{c}\|}$ denotes the normalized data vector. In [35], RaBitQ is used in combination with the IVF index, and uses the centroid of each cluster in IVF as the vector $\mathbf{c}$ for normalization. Therefore, in Equation (2), $\|\mathbf{o}_r - \mathbf{c}\|$ can be pre-computed during index construction; and $\|\mathbf{q}_r - \mathbf{c}\|$ can be computed once for each cluster and shared by all data vectors in the same cluster. Thus, the question is reduced to efficiently estimating the inner product of two unit vectors $\langle \mathbf{q}, \mathbf{o} \rangle$.

During the index phase, RaBitQ considers a set of randomly rotated bi-valued unit vectors as the quantization codebook, which is restated as follows.

$$C_{rand} := \left\{ P\mathbf{x} \;\middle|\; \mathbf{x}[i] \in \left\{ +\frac{1}{\sqrt{D}}, -\frac{1}{\sqrt{D}} \right\}, i = 1, 2, 3, ..., D \right\} \tag{3}$$

Here, P is a random orthogonal matrix [50] (a type of Johnson-Lindenstrauss Transformation) and $\mathbf{x}[i]$ denotes the ith coordinate of the vector $\mathbf{x}$. Let $\bar{\mathbf{o}}$ be the nearest vector in the codebook of the data vector $\mathbf{o}$ (i.e., $\bar{\mathbf{o}}$ is the quantized vector of $\mathbf{o}$). Because the vector $\bar{\mathbf{o}}$ in the codebook corresponds to a bi-valued vector $\bar{\mathbf{x}}$, i.e., $\bar{\mathbf{o}} = P\bar{\mathbf{x}}$, the quantization code can be represented as a D-bit string, with 1 bit for each dimension.

During the query phase, it computes an estimator $\langle \bar{\mathbf{o}}, \mathbf{q} \rangle / \langle \bar{\mathbf{o}}, \mathbf{o} \rangle$ for unbiasedly estimating $\langle \mathbf{o}, \mathbf{q} \rangle$. It is noted that $\langle \bar{\mathbf{o}}, \mathbf{o} \rangle$ is independent of the query and can be pre-computed in the index phase. $\langle \bar{\mathbf{o}}, \mathbf{q} \rangle$ can be efficiently computed for a batch of quantization codes via FastScan [4, 5]. Specifically, the computation of $\langle \bar{\mathbf{o}}, \mathbf{q} \rangle$ is equivalent to that of the inner product between the two vectors after they are *reversely* rotated (i.e., via multiplying the vectors by P^{-1}). The following equation provides the deduction of the equivalence.

$$\langle \bar{\mathbf{o}}, \mathbf{q} \rangle = \langle P\bar{\mathbf{x}}, \mathbf{q} \rangle = \left\langle \bar{\mathbf{x}}, P^{-1}\mathbf{q} \right\rangle \tag{4}$$

FastScan [3] can be used to compute $\left\langle \bar{\mathbf{x}}, P^{-1}\mathbf{q} \right\rangle$ as follows. It splits the query vector $\mathbf{q}$ into $D/4$ sub-segments where each sub-segment has 4 dimensions. Since $\bar{\mathbf{x}}$ is a bi-valued vector, the result of inner product $\left\langle \bar{\mathbf{x}}, P^{-1}\mathbf{q} \right\rangle$ in each 4-dimensional sub-segment has up to 2^4 different values. It pre-computes the 2^4 values and forms a look-up-table (LUT). Thus, the inner product for a particular $\bar{\mathbf{x}}$ can be computed by looking up the LUTs for all the $D/4$ sub-segments. FastScan [5] proposes to host the LUTs in SIMD-based registers and pack every 32 quantization codes in a batch. In this way, it can estimate distances for 32 vectors simultaneously via a series of SIMD-based operations, which has been reported to be highly efficient by many libraries from industry [24, 47]. For more technical details and theoretical analysis about RaBitQ and FastScan, we refer readers to their original papers [5, 35].

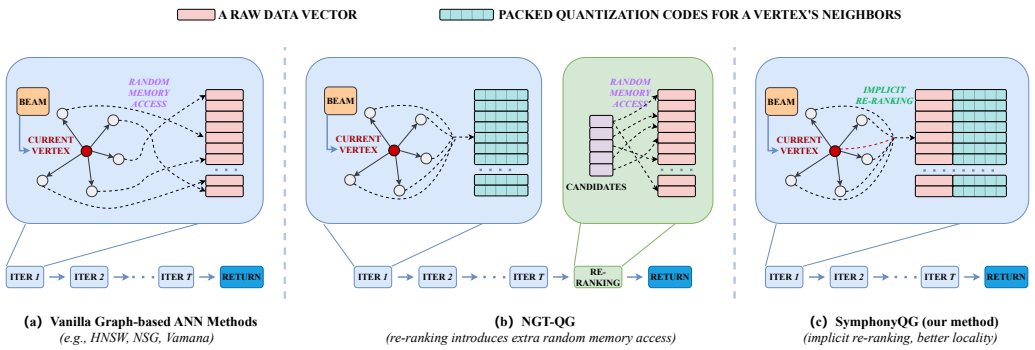


Fig. 2. Data layout and memory access pattern of vanilla graph-based methods, NGT-QG and our method. As illustrated, vanilla graph-based methods introduce plenty of random memory accesses when checking a vertex's neighbors in each iteration of searching the graph since the neighbors could be stored at different places. In NGT-QG, the neighbors of each vertex are stored sequentially at the vertex's side and they can be sequentially scanned for estimating their distances with FastScan, but it entails a re-ranking stage, which computes the exact distances of those candidates and incurs random memory accesses. Compared with NGT-QG, our method avoids the explicit re-ranking stage - instead it conducts implicit re-ranking (details will be introduced in Section 3.1.2) and thus it has no random memory accesses for re-ranking.

[3]Our implementation of FastScan is largely based on that of Faiss. For more details of the implementation, please refer to the Faiss wiki [28].

3 METHODOLOGY

In this section, we present our method SymphonyQG for ANN. SymphonyQG consists of a query phase and an index phase. During the query phase, it (1) searches on a graph-based index with FastScan and RaBitQ for boosting the search performance and (2) avoids explicit re-ranking for reducing random memory accesses. We present the query phase in Section 3.1. In particular, in Section 3.1.1, we incorporate RaBitQ to the scheme of searching on graph-based indices with FastScan. Then, in Section 3.1.2, we design a search algorithm together with a dedicated data layout to minimize random memory accesses. We summarize the memory access patterns of different methods including ours in Figure 2, which shows that (1) existing vanilla graph-based ANN methods suffer from having random memory accesses within each iteration of searching the graph (Figure 2(a)); (2) the existing NGT-QG method suffers from having random memory accesses during the re-ranking stage (Figure 2(b)); and (3) our method avoids random memory accesses both within an iteration and during re-ranking (Figure 2(c)).

During the index phase, it (1) leverages FastScan for efficient graph construction and (2) builds graph indices towards better alignment with FastScan. We present the index phase in Section 3.2. In particular, in Section 3.2.1, we propose to leverage our efficient search algorithm to accelerate the index construction (which relies on ANN queries for finding candidates/neighbors of a vertex for creating edges). Moreover, in Section 3.2.2, we propose to build a graph which has better alignment with FastScan. Specifically, we propose an adaptive pruning rule in graph construction to make sure that every vertex has its number of edges strictly being a multiple of the batch size.

We note that our method only supports ANN for Euclidean space and cosine similarity because the quantization method we use (i.e., RaBitQ [35]) is designed for these two metrics.

3.1 The Query Phase

3.1.1 Searching on Graphs with FastScan and RaBitQ for Boosting Search Performance. Suppose that for a given set of data vectors, a graph-based index has been built. To boost the search performance, we adopt the scheme of searching on the graph-based index with FastScan by following NGT-QG [47]. As is discussed in Section 1, searching on the graph-based indices with FastScan requires that for every vertex in the graph, a quantization code of every of its neighbors is stored on the side of the vertex (Figure 1). We further incorporate RaBitQ to this scheme.

We first specify the preparations needed for searching on graphs with FastScan and RaBitQ before querying. As discussed in Section 2.2, RaBitQ is originally used in combination with IVF. It normalizes the raw vectors with the centroids of the clusters in IVF (i.e., using the centroids as the center vector $\mathbf{c}$) and reduces the task of estimating distances between raw vectors to one of estimating the inner product between unit vectors. However, in graph-based indices, the centroids do not exist. Thus, the first question is how to normalize the vectors in graph-based indices. Recall that when searching graph-based indices, the algorithm iteratively visits a vertex and estimates distances for all its neighbors. A natural idea is to normalize the vectors of the neighbors with the vector of the vertex (i.e., using the vector of the vertex as the center vector $\mathbf{c}$). Then with the normalized data vectors, we can compute the quantization codes of RaBitQ [4] and store them on the side of the vertex. Note that the normalization process can be conducted before the query phase starts since it does not rely on a query vector, and thus its cost can be shared by all queries on the dataset. For example in Figure 1, a quantization code of the blue vertex is stored on the side of the

[4]It is worth noting that when a vector is a neighbor of two different vertices, its quantization codes stored on their sides are different from each other because the normalization is based on different vectors (e.g., in Figure 1, the red and green vertex both store quantization code of the blue vertex, but these codes are different.).

red vertex and the quantization code is computed by normalizing the blue vector with respect to the red vector.

However, this way of normalization introduces new issues in estimating distances during querying. In particular, recall that before efficiently estimating distances for a batch of vectors, FastScan needs to prepare LUTs (look-up tables) in prior. Specifically, based on normalization, the task of estimating the distances between the raw vectors has been reduced to that of computing the inner product $\langle \bar{\mathbf{x}}, P^{-1}\mathbf{q} \rangle$ (see Section 2.2) where $\bar{\mathbf{x}}$ is the bi-valued vector represented by the quantization code and $\mathbf{q}$ is the normalized query vector with respect to a center vector $\mathbf{c}$, i.e., $\mathbf{q} = \frac{\mathbf{q}_r - \mathbf{c}}{\|\mathbf{q}_r - \mathbf{c}\|}$. For the vector $\mathbf{q}$, FastScan needs to prepare LUTs such that it can compute $\langle \bar{\mathbf{x}}, P^{-1}\mathbf{q} \rangle$ for a batch of quantization codes all at once. Because $\mathbf{q}$ depends on the vector $\mathbf{c}$ for normalization, when different $\mathbf{c}$'s are used for normalization, different normalized query vectors $\mathbf{q}$ would be obtained, and consequently different LUTs should be prepared. In [35], RaBitQ is used in combination with IVF. The preparation of LUTs happens before probing a cluster and the time cost can be shared by all the vectors in the same cluster of IVF (e.g., hundreds or thousands of vectors). Thus, the amortized time cost is tiny. However, in our circumstance, we need to prepare LUTs every time we visit a vertex (because the quantization codes of its neighbors are produced by using the vertex's vector as the center vector for normalization), and the cost can only be shared by its neighbors (e.g., 32, 64 or 128 vectors). This is very costly and could cancel out the performance gain brought by FastScan.

We solve this problem by converting the problem of computing $\langle \bar{\mathbf{x}}, P^{-1}\mathbf{q} \rangle$ (which is dependent on a center vector $\mathbf{c}$) to one of computing $\langle \bar{\mathbf{x}}, P^{-1}\mathbf{q}_r \rangle$ (which is independent of the center vector $\mathbf{c}$). We have the following deductions.

$$\langle \bar{\mathbf{x}}, P^{-1}\mathbf{q} \rangle = \left\langle \bar{\mathbf{x}}, P^{-1}\frac{\mathbf{q}_r - \mathbf{c}}{\|\mathbf{q}_r - \mathbf{c}\|} \right\rangle \tag{5}$$

$$= \frac{1}{\|\mathbf{q}_r - \mathbf{c}\|} \cdot \left(\langle \bar{\mathbf{x}}, P^{-1}\mathbf{q}_r \rangle - \langle \bar{\mathbf{x}}, P^{-1}\mathbf{c} \rangle \right) \tag{6}$$

where Equation (5) is due to the definition of $\mathbf{q}$; and Equation (6) decomposes Equation (5) into two parts. **First**, $\langle \bar{\mathbf{x}}, P^{-1}\mathbf{c} \rangle$ is independent of the query. It can be pre-computed in the index phase. **Second**, $\langle \bar{\mathbf{x}}, P^{-1}\mathbf{q}_r \rangle$ is independent of the vector $\mathbf{c}$ for normalization. Thus, instead of preparing LUTs for computing $\langle \bar{\mathbf{x}}, P^{-1}\mathbf{q} \rangle$ as the original RaBitQ does, we can prepare LUTs for computing $\langle \bar{\mathbf{x}}, P^{-1}\mathbf{q}_r \rangle$. As the latter is independent of $\mathbf{c}$, the LUTs can be prepared when a query comes and shared by all vectors in a database.

In summary, the algorithm of searching a graph-based index with FastScan and RaBitQ works as follows. It uses the vector of a vertex for normalizing its neighbors before querying. Then, for every vertex, it computes the quantization codes of all its neighbors using RaBitQ and stores them compactly on the side of vertex. During querying, FastScan is used for estimating distances. As discussed in an existing study [20], the effectiveness of using an estimated to prune out objects relies on the similarity between the distribution of estimated distance and the distribution of the "true" distance. Our method uses estimated distances produced by RaBitQ to guide the search process during querying. Because RaBitQ provides an unbiased estimator with a theoretical error bound [35], the estimated distance used in our method has a similar distribution with the "true" distance. Therefore, it is effective for our method to prune out objects to improve the query efficiency, which is aligned with our experimental results.

3.1.2 Avoiding Explicit Re-Ranking for Reducing Random Memory Accesses. Recall that during the process of searching on graph-based indices, vanilla greedy beam search computes exact distances based on raw vectors, which requires random memory accesses within each iteration of

searching (see Section 2.1 and Figure 2(a)). When searching on graph-based indices with FastScan, the algorithm estimates distances of the neighbors of a vertex based on the quantization codes stored at its side, which avoids random memory accesses within each iteration (see Figure 2(b)). However, because the searching is based on the estimated distances, this entails a re-ranking step to achieve reasonable recall. Specifically, in NGT-QG, at the end, it accesses the vectors which have the smallest estimated distances to the query, computes their exact distances and finds the nearest neighbor accordingly. Because for computing exact distances, it needs to access raw vectors, this introduces random memory accesses. In order to eliminate these random accesses, we propose to conduct re-ranking implicitly during searching on graph-based indices, which we explain as follows.

Recall that as discussed in Section 3.1.1, our method uses the vector of a vertex for normalizing the vectors of all its neighbors. In order to estimate distances for all its neighbors based on Equation (2), this requires to compute the exact distance between the vector of the vertex and the query vector (i.e., $\|\mathbf{q}_r - \mathbf{c}\|$ in Equation (2), where $\mathbf{c}$ corresponds to the vector of the vertex). Therefore, we re-use this exact distance to maintain the NN that has been searched so far while using the estimated distances for searching graph-based indices. With a specialized data layout (which will be specified latter and illustrated in Figure 2(c)), this does not introduce extra random memory accesses. When the search process terminates, the vector with smallest exact distance maintained in the algorithm is returned as the result of the ANN query without an extra re-ranking step.

The above idea would help remove the random accesses of re-ranking, but it introduces accuracy loss. Specifically, in the above process, we update the NN only when we visit a vector and compute its exact distance. Note that a vector would be visited only when it has the smallest estimated distances during searching. If the distance of a true NN is over-estimated, the NN is likely to be missed. To resolve this issue, we propose to conduct greedy beam search using *multiple* estimated distances for every vector. Specifically, during the conventional greedy beam search with FastScan, in each iteration, the algorithm visits a vertex and estimates the distance for each of its neighbors. Each neighbor along with its estimated distance is appended into the beam set if the neighbor has not been in the set. In contrast, in our method, we always append the neighbor along with its estimated distance into the beam set unless it has been visited. As a result, the same vertex may be added to the beam set multiple times (along with different estimated distances). The rationale is as follows.

Recall that based on the theoretical guarantee of RaBitQ, every estimated distance is unbiased and error-bounded. Thus, when multiple estimated distances are used for a vector during searching, the search algorithm will have the following properties. (1) For a true NN, in each estimation, RaBitQ has 50% probability to produce an under-estimation of its distance. When multiple estimations are used, it is highly likely that there is at least one under-estimation. Note that the true NN has the smallest exact distance among all vectors. Therefore, an under-estimation of its distances is highly likely to be the smallest in the beam set in a certain iteration. This leads to a visit to the true NN and thus, the true NN is not missed. (2) For other vectors, RaBitQ guarantees that the estimated distances are error-bounded. When multiple estimations are used, it is guaranteed that all the estimated distances are within an error bound, i.e., all of the multiple estimated distances is unlikely to deviate significantly from the true distances due to the bound. Therefore, when the true distance of a vector is larger than that of the NN by a clear margin, it is less likely that it would be visited due to a severe under-estimation. On the other hand, when the true distance of a vector is slightly larger than that of the NN, the vector is very close to the query. Visiting the vector could help to find a true NN on the graph. Thus, based on this strategy, the accuracy is secured. The ablation study in Section 4.2.3 verifies the effectiveness of this technique.

Algorithm 1 Querying

Input: A query vector, the beam size n_b, a graph-based index $\mathbf{G}$, an entry point e
Output: The nearest neighbor of the query

1: Initialize the beam set $\mathcal{S}$ and NN with the entry point e
2: **while** $\exists$ an unvisited vertex in $\mathcal{S}$ **do**
3: Find the unvisited vertex p with the smallest estimated distance from $\mathcal{S}$ and mark p as visited
4: Compute the exact distance of p and update NN
5: Estimate distances for the neighbors of p in $\mathbf{G}$ via FastScan
6: Append all unvisited neighbors along with their estimated distances into $\mathcal{S}$ and cut off the size of $\mathcal{S}$ to n_b
7: **return** NN

3.1.3 Summary of Querying Algorithm. We summarize the querying algorithm in Algorithm 1. Given a query vector, a hyperparameter beam size n_b, a graph-based index $\mathbf{G}$ and an entry point e, the algorithm starts with initializing the beam set $\mathcal{S}$ and the nearest neighbor NN with the entry point (line 1). It then proceeds querying in iterations (line 2-6). In each iteration, it first finds the unvisited vertex with the smallest approximate distance p and computes its exact distance (line 3-4). Next, it estimates distances for its neighbors with FastScan and appends all the unvisited neighbors along with the estimated distances to the beam set (line 5-6). The algorithm terminates when all the vertices in $\mathcal{S}$ have been visited and returns the nearest neighbor.

Based on the querying algorithm above, we propose a dedicated data layout in order to minimize the random memory accesses, which is presented as follows. Overall, the index consists of n data objects. Each data object is assigned with a sequential block of memory which stores (1) its raw data vector, (2) the data for FastScan (including the quantization codes and the pre-computed factors needed by RaBitQ, e.g., $\|\mathbf{o} - \mathbf{c}\|$) and (3) the neighbors of the graph (see Figure 2(c) [5]). During querying, when visiting a certain vertex on the graph in an iteration, the algorithm accesses the corresponding block memory of the vertex. Clearly, the algorithm accesses a continuous segment in memory when visiting a vertex in each iteration, which leads to good cache locality. Based on this data layout, we summarize and compare the memory access pattern of the vanilla graph-based methods, NGT-QG and our method in Figure 2. We would like to highlight that compared with NGT-QG, our method eliminates the re-ranking at the end of the algorithm. This avoids the costly random memory accesses.

3.1.4 Implementation Techniques. Besides the aforementioned techniques in terms of algorithms, we would also like to mention the following techniques in our implementation.

Memory Prefectching. Although the random memory accesses within an iteration have almost been eliminated, there could still be cache miss between iterations, i.e., the sequential piece of data for the vertex to visit in the next iteration might not be cached. In order to further relieve the issue, we prefetch the data of the vector which has the smallest distance when updating the beam set (line 6 in Algorithm 1).

More SIMD Acceleration. In [35], RaBitQ only uses SIMD (i.e., FastScan) for computing $\langle \bar{\mathbf{x}}, P^{-1}\mathbf{q} \rangle$ and supports only till AVX2. We conduct more extensive optimizations with SIMD in the whole

[5]Here, we omit the data of neighbors in Figure 2 to provide a clearer comparison among different methods. In practice, the neighbors of each vertex are stored right after the data for FastScan in our method for better cache locality.

process of the algorithm, e.g., using SIMD to accelerate the computation of Equation (2), and provide the support of AVX512.

Fast Johnson-Lindenstrauss Transformation. For efficiently implementing the random rotation (Johnson-Lindenstrauss Transformation [50], JLT) for RaBitQ, we apply an algorithm of Fast JLT based on Fast Hadamard Transformation [2, 3, 29]. This improves the time complexity of random rotation from $O(D^2)$ to $O(D \log D)$, where D denotes the number of dimensions of a vector.

3.2 The Index Phase

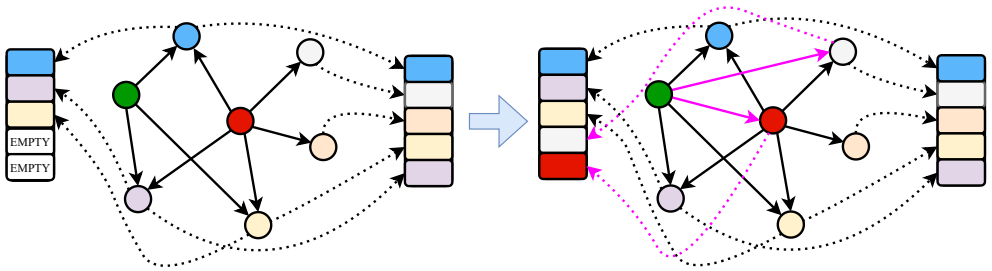
3.2.1 Leveraging FastScan for Building Graph-based Indices Efficiently. In many graph-based ANN methods [30, 32, 46, 48, 57, 58, 67, 81], constructing edges for each vertex can be roughly divided into two stages. In stage one, the algorithm finds several vertices as the candidates for a vertex's neighbors. In stage two, a pruning rule is applied on the candidates. Finally, an edge between each candidate that has not be pruned and the vertex will be created. For example, NSG [32] constructs its graph in the scheme of initializing a graph and adjusting it. Specifically, for every vector, it conducts ANN query on the initialized graph. Then, the results of ANN query are taken as the candidates. A new graph is then built after applying a pruning rule on the candidates.

We notice that our efficient search algorithm can be easily applied to accelerate the above scheme of graph construction. Specifically, the above construction algorithm conducts ANN query for every vector to generate candidates, during which, our efficient search algorithm can be applied to accelerate the ANN query. The detailed process of our index construction is presented as follows. We consider randomly initializing a graph and adjusting it iteratively. In each iteration, for the graph produced in the last iteration, we first prepare the data for FastScan (e.g., quantization codes). Then, we apply our search algorithm for finding candidates for every vertex. For each vertex, we use the pruning rule of NSG [32] to select at most R vertices as the vertex's new neighbors (the neighbors in the graph of the last iteration will not be retained in the new graph). After all vertices have their new neighbors, we adjust edges in the graph and start the next iteration. Note that the graph index is adjusted at the end of each iteration and is unchanged within an iteration. This indicates that the candidate generation and pruning of different vertices are independent to each other. Thus, the indexing algorithm in each iteration is highly parallelable. According to our experimental results in Section 4.2.3, compared with the original algorithm of constructing NSG, the above indexing algorithm brings at least 2.5x speed-up in indexing without harming the search performance of the constructed graphs. It is also worth emphasizing that our method is at least 8x faster than NGT-QG in indexing.

3.2.2 Refining Graph-based indices towards Better Alignment with FastScan. Based on the above algorithm, we have built a graph based on an existing pruning rule of NSG. However, as discussed in Section 1, FastScan requires to estimate distances in a batch for 32 vectors simultaneously. Without a specialized design, it is likely that the out-degree of a vertex in the constructed graph is not a multiple of the batch size and thus, leads to a waste of computation. We note that the graph cannot have the desired number of edges because the pruning rule is sometimes too restrictive, i.e., after pruning, the number of edges has already been smaller than a target which we denote by R.

According to the experimental results in Section 4.2.3, based on the pruning rule of NSG, the average out-degree of the graph is less than R on all tested datasets (e.g., the average out-degree of graph on the MSong dataset is 19.8 when $R = 32$). To solve this issue, we propose to further refine the graph by supplementing edges in the graph such that the number of edges of each vertex is strictly aligned with the batch size(see an example in Figure 3).

Specifically, we first build edges based on the original pruning rule of NSG [32]. For a vertex, if the number of its edges is smaller than R, we re-consider the candidates that have been pruned and



(a) Before Refinement **(b) After Refinement**

Fig. 3. Refining a graph towards better alignment with FastScan by supplementing edges. Here, we assume R and the batch size are 5 for the simplicity of illustration. The out-degree of the green vertex is only 3 in the original graph because the original pruning rule is too restrictive. After supplementing edges with an adaptive pruning rule, its out-degree is guaranteed to be 5. For the red vertex, its out-degree is already aligned with the batch size, so we do not need to supplement edges for this vertex.

adaptively relax the pruning rule on them such that every vertex has exactly R edges. In particular, recall that according to the analysis in NSG [32], its pruning rule ensures the diversity of the edges of a vertex, i.e., the angle between any two out-edges of a vertex in the high-dimensional space is at least 60°. Given the pruned candidates, we apply an *angle-based* pruning rule, i.e., for a candidate of the vertex, if there exists another candidate whose distance to the vertex is smaller and the angle between the edges of the candidates is smaller than a given threshold, then we prune the candidate. It is clear that the larger the threshold is, it is more likely that a candidate is pruned and subsequently, the smaller the candidate set is after pruning. As the size of the candidate set after pruning is *monotonic* with respect to the threshold, we apply *binary search* on the threshold. It adaptively finds the most restrictive pruning rule (i.e., the largest threshold for pruning) such that a vertex can have exactly R neighbors. Note that the pruning rule is adaptive in the sense that different vertices may have different thresholds for pruning. Thus, it is ensured that every vertex in the graph strictly has a preset number of out-edges [6]. This further ensures that the degree of every vertex is aligned with FastScan.

Note that because in FastScan, the distances are estimated batch by batch, supplementing edges for a graph to fill a batch does not increase the time cost of distance estimations. On the other hand, the more edges a graph has, the less likely the true NN is missed during graph-based searching. Thus, the algorithm achieves better time-accuracy trade-off. According to the ablation study in Section 4.2.3,our graph refinement strategy brings consistent improvement in search performance on all the tested datasets.

We summarize the indexing of our method in Algorithm 2. The graph is randomly initialized to a graph, where every vertex has R neighbors. Then the graph is adjusted iteratively (line 2-8). In each iteration, the algorithm prepares the data for FastScan based on $\mathbf{G}$ (line 3-4). Then it uses our querying algorithm to find the candidates of neighbors for all vertices and applies NSG's pruning rule on these candidates (line 5-7). At the end of each iteration, it adjusts the graph with new neighbors for each vertex (line 8). As discussed, the candidate generation and pruning are highly parallelable across different vertices. After t iterations ($t = 3$ or 4 in practice), it finds vertices whose

[6]In the extreme case that the number of candidates is smaller than the maximum out-degree, our method will use randomly selected neighbors to make sure every vertex's out-degree is a multiple of 32.

Algorithm 2 Indexing

Input: A dataset $\mathcal{D}$, the out-degree R, the # of iterations t
Output: A graph-based index **G** with the data for FastScan

```
1: Randomly initialize a graph G
2: for 1 ≤ i ≤ t do
3:     for all vertices do
4:         Prepare data for FastScan based on G
5:     for all vertices do
6:         Search on G to find candidates
7:         Prune candidates to obtain new neighbors
8:     Adjust G
9: Supplement edges for vertices whose out-degree < R
10: Update the data for FastScan based on G
11: return G with the data for FastScan
```

out-degree is less than R and supplement edges for them (line 9). At the end of indexing, it updates the data used for FastScan based on the final graph and returns the index (line 10-11).

3.3 The Summary

In summary, SymphonyQG handles ANN query by using RaBitQ and FastScan to search on graph-based indices. During querying, we incorporate RaBitQ to graph-based indices by resolving issues in normalization and efficient distance estimation. To reduce the cost of random memory access, we design the implicit re-ranking strategy and eliminate the explicit re-ranking step of NGT-QG. To handle the issue of possible accuracy loss caused by implicit re-ranking, we propose a search strategy based on multiple estimated distances. As for indexing, we use our efficient search algorithm to accelerate the candidate generation process of index construction. Moreover, considering the in-batch computation of FastScan, we design a graph refinement strategy with an adaptive pruning rule, which makes sure the out-degree of every vertex is aligned with the batch size. Experimental studies in Section 4 shows that based on all the proposed techniques, SymphonyQG establishes the new state-of-the-art in terms of the time-accuracy trade-off on all tested real-world datasets. For example, at 95% recall ($K = 10$), our method achieves 1.5x-4.5x QPS compared with the most competitive baselines (which differ from dataset to dataset) and achieves 3.5x-17x compared with the classical library HNSWlib on all tested datasets. At the same time, the indexing of our method is at least 2.5x faster than NSG and is at least 8x faster than NGT-QG.

Overhead of Memory Consumption. Despite the promising performance of SymphonyQG in terms of the accuracy-efficiency trade-off, we would like to note that the method brings some overhead of memory consumption, whose details are presented as follows. Let D and R be the dimensionality and the out-degree of every vector respectively. First of all, like vanilla graph-based methods, for every vector, SymphonyQG uses $32 \times D$ bits (D floating-point numbers) to store the raw vectors and $32 \times R$ bits (R integers) to store a vector's neighbors. In addition, recall that by following NGT-QG [45], for every vector, SymphonyQG creates quantization codes for all its R neighbors. Each quantization code takes D bits [35], and thus, the extra space for storing the codes is $D \times R$ bits. Therefore, the total space cost of our index is $n \times (32D + 32R + DR)$ bits, where n is the number of vertices. Under the typical settings of the out-degree ($R = 32$ and 64), the overhead for storing the codes is 1x and 2x of that for storing the raw vectors respectively.

4 EXPERIMENTS

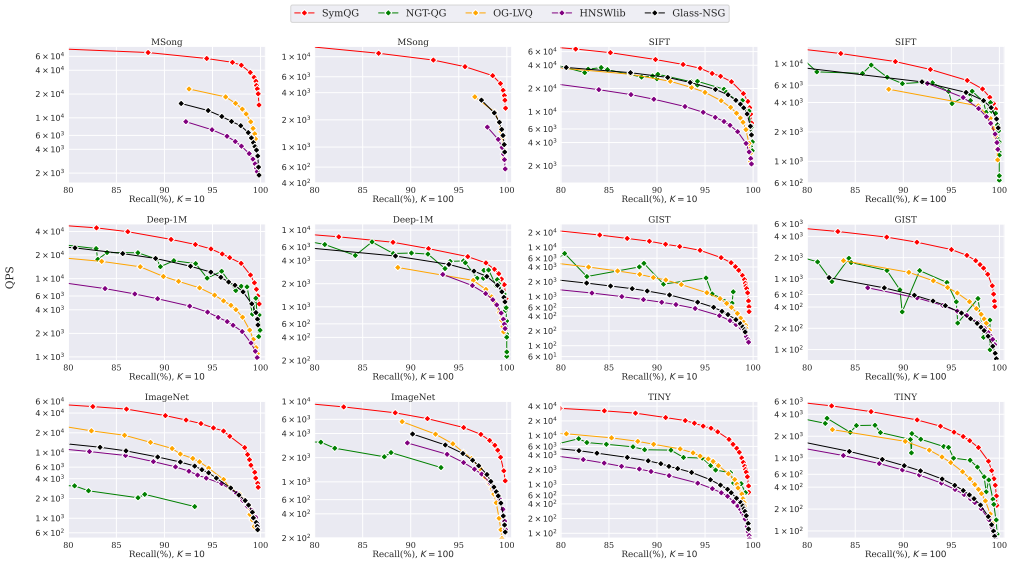


Fig. 4. Comparison of all methods on 6 real-world datasets (QPS-Recall). The missing curve of a certain method indicates that it fails to achieve at least 80% recall on the dataset.

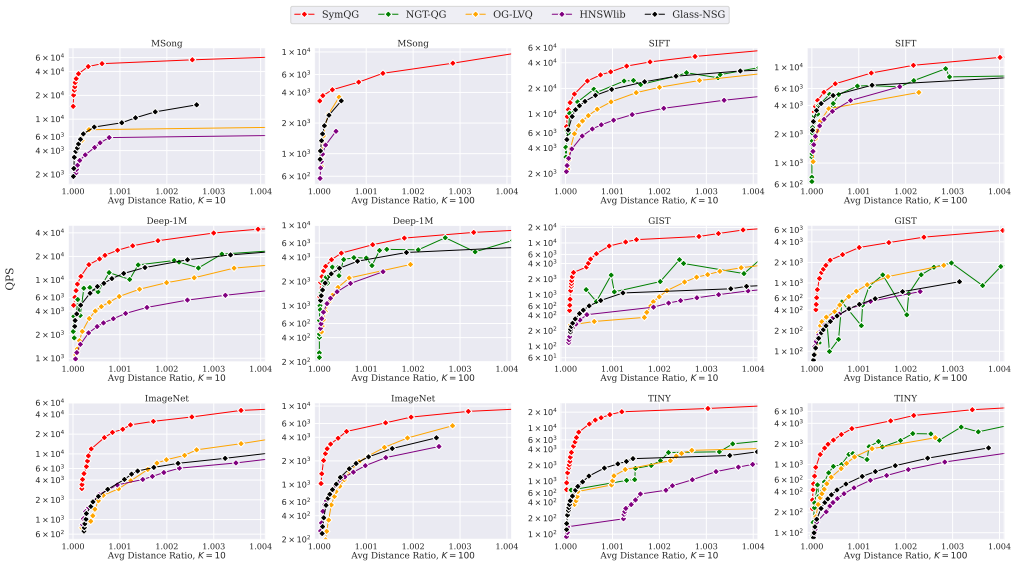


Fig. 5. Comparison of all methods on 6 real-world datasets (QPS-Ave Distance Ratio). The missing curve of a certain method indicates that it fails to achieve at least 1.004 average distance ratio on the dataset.

We provide the experimental setup in Section 4.1 and the experimental results which involve four folds in Section 4.2. (1) In Section 4.2.1, we evaluate the search performance of our method

by comparing it with the state-of-the-art open-source libraries on real-world datasets. We also report the indexing time and memory footprint of every method. (2) In Section 4.2.2, we test the scalability of our method on a dataset with 100 million data vectors. (3) In Section 4.2.3, we verify the effectiveness of the proposed techniques via an ablation study. (4) In Section 4.2.4, we conduct the parameter study for our newly introduced parameter.

4.1 Experimental Setup

Datasets. To evaluate the search performance of different methods, we adopt 6 moderate-scale (million-scale) datasets including MSong, SIFT, Deep-1M, GIST, ImageNet and TINY. We also test the scalability of our method on two large-scale datasets Deep-100M and MSTuring-100M [7]. These datasets cover diverse dimensionality, sizes and similarity metrics, and they have been widely used in existing studies [1, 8, 23, 32, 55, 58, 62, 63] to benchmark ANN algorithms and libraries. All these datasets provide both the data vectors and a set of query vectors for evaluation. The detailed statistics of the datasets are presented in Table 1.

Machine Configuration. We run all experiments on a server with two Intel(R) Xeon(R) Gold 6418H@4.0GHz CPUs and 1TB DDR5 memory (@4800MT/s) under Ubuntu 22.04 LTS with GCC 11.4.0 compiler. For all methods, by following most of the existing studies and benchmarks [8, 23, 32, 48, 58, 67, 79], the search performance is evaluated in a single thread and the indexing time is measured using multiple threads (96 threads, 48 cores with our server).

Metrics of Query Performance. Following existing studies and benchmarks [8, 23, 32, 48, 58, 65, 66], we measure the query efficiency by the number of queries responded per second (QPS), and we measure the query accuracy by recall and average distance ratio. The recall is defined as $\frac{|G \cap S|}{K}$, where G denotes groundtruth of K nearest neighbors, and S denotes search results of an ANN algorithm. The average distance ratio (ADR) is the average of the distance ratios of the ANN query results wrt the groundtruth of K nearest neighbors. To show the trade-off between accuracy and efficiency of different methods, we draw QPS-recall curve and QPS-ADR curves.

Table 1. Statistics of the datasets. N and N_q denote the number of data vectors and the number of queries, respectively. D denotes the dimensionality of vectors

Dataset	N	N_q	D	Size (GiB)	Distance
MSong	~1M	200	420	1.6	Euclidean
SIFT	1M	10,000	128	0.5	Euclidean
Deep-1M	1M	1,000	256	0.9	Cosine
GIST	1M	1,000	960	3.6	Euclidean
ImageNet	~2.3M	200	150	1.4	Euclidean
TINY	5M	200	384	7.2	Euclidean
Deep-100M	100M	10,000	96	37.8	Cosine
MSTuring-100M	100M	10,000	100	40.0	Cosine

Methods and Parameters. We compare our method with four well-optimized graph-based open-source libraries including NGT-QG (Yahoo Japan) [47], OG-LVQ (Intel) [1], Glass-NSG (Zilliz) [89] and HNSWlib [58, 85]. For all methods, we conduct parameter search to find the optimal setting of the parameters for indexing [8]. Then, we fix the optimal parameters for indexing and plot the QPS-recall curve by varying the parameters for querying. The details of the optimal parameter

[7]We adopted the first 10K query vectors from the query set of MSTuring-100M.

[8]We build multiple indices and test their QPS-recall trade-off. Based on the results, we adopt the parameter for indexing which achieves the highest QPS at 95% recall.

settings for each dataset are provided in our technical report [37] due to the limitation of space. (1) SymphonyQG (SymQG for short) is our method. There are three parameters during indexing of SymphonyQG, the maximum out-degree of the graph R, a parameter that controls the number of candidates during graph construction named EF and the number of iteration for graph construction t (more details can be found in Section 3.2). During querying, the beam size n_b is varied to control the time-accuracy trade-off. We fix EF to 400 and vary R among 32, 64, 128. We set t as 4 for MSTuring-100M and 3 for other datasets. (2) NGT-QG [47] is a method published in the open-source library NGT by Yahoo Japan. It is the originator of the QG framework, which pioneers in searching on graph-based indices with FastScan and has been reported to be one of the state-of-the-art libraries in the well-acknowledged ANN-Benchmark [8]. The indexing and querying of NGT-QG involves many parameters. We refer readers to the library [47] for detailed information. We conduct parameter search among its suggested parameter settings in ANN-Benchmark [8]. (3) OG-LVQ [1] is a method published in the open-source library SVS by Intel. It is based on graph-based indices and scalar quantization. The indexing parameters include a) the parameters for graph-based indices including R and EF like our method anda parameter α which controls the pruning of its graph-based index Vamana [48] and b) the number of bits for scalar quantization. We follow [1] by fixing $\alpha = 1.2$ and decide R and EF in the same way of our method. As for the number of bits for scalar quantization, we search parameters among its provided settings including LVQ-8, LVQ-4x8, and LVQ-8x8. During querying, similar to our method, the beam size n_b is varied to control the time-accuracy trade-off. (4) Glass [89] is an open-source graph-based ANN library published by Zilliz. It has been reported to be one of the state-of-the-arts in ANN-Benchmarks [8]. In our experiment, we use NSG [32] provided in Glass [9]. The indexing parameters include R and EF like our method. Besides, it involves a parameter which indicates the optimization level. We observe that a high optimization level may cause disastrous failure. Thus, for each dataset, we evaluate the search performance based on all optimization levels and report the best results. R and EF are decided in the same way of our method. The time-accuracy trade-off in the query phase is controlled by the beam size n_b. (5) HNSWlib [85] offers an efficient implementation of the classical graph-based ANN method HNSW [58]. The indexing parameters of HNSW include EF and maximum out-degree M similar to our method, and they are decided in the same way of our method. The time-accuracy trade-off in the query phase is controlled by the beam size n_b. As all the libraries provide convenient Python interfaces, we conduct all experiments based on the Python bindings. We note that in recent years, there are also many scientific studies which explore the opportunities of optimizing different aspects of ANN algorithms [17, 31, 34, 67, 87]. However, these scientific studies usually focus on a certain component in an algorithm while spending less effort on engineering optimizations on the whole library. This makes their empirical performance less competitive when compared with well-optimized libraries such as NGT [47] (Yahoo Japan), SVS [1] (Intel) and Glass [89] (Zilliz), which are mostly from industry and have been included for comparison in this paper. We, thus, exclude these studies from comparison.

4.2 Experimental Results

4.2.1 Performance Study. In this section, we evaluate the search performance as well as the indexing time and memory footprint of each method on real-world datasets.

Search Performance. We first evaluate the search performance for all methods on moderate-scale datasets. As discussed, we build indices with the optimal parameter setting for each dataset and vary parameters in the query phase (e.g., the beam size n_b) to plot the QPS-recall curve and QPS-ADR curve. In particular, the search performance is evaluated under two different settings including

[9]We fix several bugs for this library so that it can run normally on the large-scale dataset Deep-100M.

Table 2. Indexing time (minutes). The results of NGT-QG are not shown on two large-scale datasets since it fails to build an index with 1TB memory within 1 day.

	SymQG	NGT-QG	OG-LVQ	Glass-NSG	HNSWlib
MSong	0.8	97.3	2.2	2.1	1.0
SIFT	0.4	6.5	1.1	1.3	0.4
Deep-1M	0.8	8.1	2.3	2.1	0.8
GIST	2.6	27.3	6.5	6.1	3.3
ImageNet	1.6	14.3	3.3	3.4	1.2
TINY	7.6	142.2	19.9	19.5	9.7
Deep-100M	51.2	-	182.5	290.6	47.6
MSTuring-100M	92.6	-	165.8	304.2	59.8

Table 3. Memory footprint (GiB). The results of NGT-QG are not shown on two large-scale datasets since it fails to build an index with 1TB memory within 1 day.

	SymQG	NGT-QG	OG-LVQ	Glass-NSG	HNSWlib
MSong	4.0	12.0	1.0	1.7	1.9
SIFT	1.5	5.2	0.5	0.7	0.7
Deep-1M	3.9	10.8	0.6	1.2	1.3
GIST	12.5	22.9	1.7	3.8	4.2
ImageNet	8.1	16.0	1.2	1.9	2.1
TINY	32.1	84.2	4.1	8.4	8.9
Deep-100M	140.3	-	30.9	51.6	59.2
MSTuring-100M	140.3	-	33.5	53.2	59.4

$K = 10$ and $K = 100$. Figure 4 plots QPS-recall curves (upper-right is better), and Figure 5 plots QPS-ADR curves (upper-left is better) on six moderate-scale datasets. We have the following key observations. (1) In general, the search performance of SymQG surpasses all baseline methods. Compared to other methods, SymQG achieves a better QPS-recall trade-off and better QPS-ADR trade-off on all tested datasets. When $K = 10$, our method surpasses the most competitive baselines (which differ from dataset to dataset) by 1.5x-4.5x in QPS at 95% recall on all the tested datasets. Moreover, SymQG outperforms the classical baseline HNSWlib by 3.5x-17x in QPS at 95% recall on all tested datasets. When $K = 100$, our method still shows the best search performance on tested datasets. As for QPS-ADR curves presented in Figure 5, our method achieves a higher QPS than all baselines at the same ratio for both $K = 10$ and $K = 100$ for all tested datasets, which is consistent with the QPS-recall curves. (2) As discussed in Section 1, our method can handle ANN query of datasets on which NGT-QG fails. As illustrated in Figure 4, we observe NGT-QG cannot achieve at least 80% recall on the MSong dataset. This can be explained by the disastrous failure of PQ on the dataset as has been reported in [35]. It is worth noting that on ImageNet, NGT-QG also incurs severe performance drop. In contrast, our method has more robust performance across different datasets. (3) The curves of NGT-QG are not as smooth as those of other methods. This is because during searching, it involves two parameters, one for searching and the other for re-ranking, and it needs to tune these parameters simultaneously. This introduces new difficulties of parameter tuning when using NGT-QG in practice. We refer readers NGT-QG's library [47] and ANN-Benchmark [8] for more details about the searching parameters. On the other hand, SymQG adopts implicit re-ranking (see Section 3.1.2), which eliminates the re-ranking parameter. Therefore, our method has smooth QPS-recall curves.

Indexing Time and Memory Footprint. We measure the indexing time and memory footprint for all the methods on all the datasets. The indexing time is reported in Table 2. As illustrated, the indexing time of our method is comparable with HNSWlib and is much shorter than other

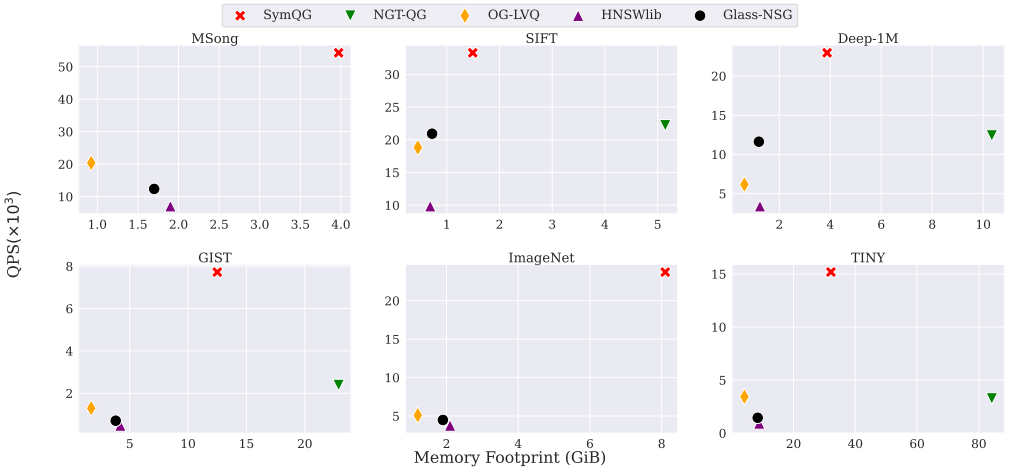


Fig. 6. Trade-off between QPS and memory footprint for different methods (results with recall at 95% and $K = 10$). The missing results of a certain method indicate that it fails to achieve at least 95% recall on the dataset.

methods. The indexing time of NGT-QG is significantly higher than other methods. This is because it adopts an extra process to optimize a constructed graph [46]. The memory footprint is reported in Table 3. As demonstrated, the memory footprint of our method and NGT-QG is higher than OG-LVQ, Glass-NSG and HNSWlib. This is because our method and NGT-QG search on graph-based indices with FastScan, which requires to store multiple quantization codes for a single vertex (see Figure 1 and Section 1). This is likely to be unavoidable if we target to mitigate random memory accesses during searching in pursuit of search performance. In addition, compared with NGT-QG, the memory footprint of our method is much smaller, partly because the quantization codes used by our method (which are produced by RaBitQ) are shorter than those used by NGT-QG (which are produced by PQ). In Figure 6, we show time-space trade-off for different methods. As illustrated, SymphonyQG achieves a significantly higher QPS compared with all baselines. Although the space consumption of our method is higher than vanilla graph-based methods (i.e., HNSW and NSG), it is still smaller than that of NGT-QG. In summary, our method establishes the new state-of-the-art query performance and achieves a better time-space trade-off than NGT-QG.

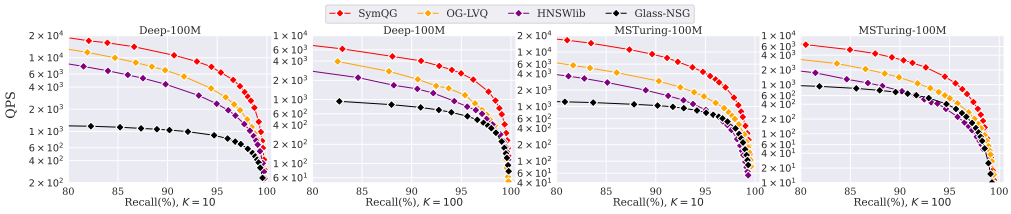


Fig. 7. Evaluation of scalability.

4.2.2 Scalability Study. We verify the scalability of our method on Deep-100M and MSTuring-100M datasets. The indexing parameters are set as $R = 32$ and $EF = 400$ for SymQG, OG-LVQ, Glass-NSG and HNSWlib. We exclude NGT-QG in the evaluation in the scalability study since NGT-QG fails to build an index with 1TB memory within 1 day (as a reference, our method can finish indexing

within 2 hours). We notice that there is another method named QBG, which is provided in the NGT library for handling large-scale datasets. However, as has been reported [1], the method cannot achieve reasonable recall (e.g., >85%). Thus, we do not include QBG in the study either. The QPS-recall curves are presented in Figure 7, and indexing time and size are shown in Table 2 and Table 3, respectively. The result shows that our method has good scalability and is still the most efficient on Deep-100M and MSTuring-100M. Specifically, our method achieves at least 2.5x QPS compared with the most competitive baseline OG-LVQ at 95% recall. On the other hand, we note that the memory footprint of our method is larger than other methods by a clear margin. This is a limitation of our method caused by storing the data for FastScan (e.g., the quantization codes). We leave it as a future work to further optimize its memory consumption.

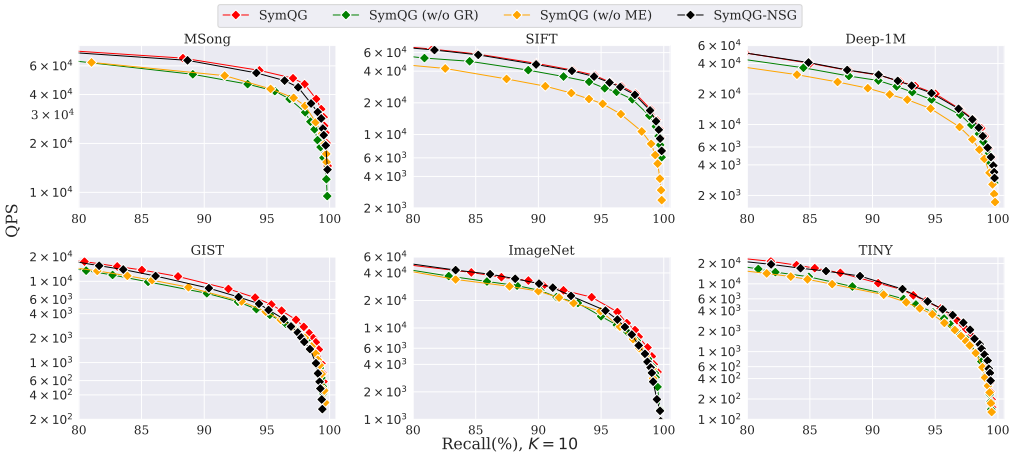


Fig. 8. Ablation study on six moderate-scale datasets.

4.2.3 Ablation Study. In this section, we evaluate the effectiveness of the components in our method with an ablation study. In particular, we study the effects of ablating a certain component from our method on the QPS-recall trade-off. The study is conducted under the parameters of $R = 32$, $EF = 400$ and $K = 10$ [10]. The ablation study involves the following three folds.

Evaluating the Impact of Using Multiple Estimated Distances on Query Performance. To show the effectiveness of using multiple estimated distances (Section 3.1.2), we keep all other components while using only a single estimated distance for each vector during searching, i.e., we use the first estimated distance. We denote the method without using multiple estimated distances as SymQG (w/o ME). According to Figure 8, SymQG surpasses SymQG (w/o ME) on all tested datasets, which illustrates the effectiveness of using multiple estimated distances.

Evaluating the Impact of Our Efficient Indexing Algorithm on Index and Query Performance. We verify our index algorithm in both aspects of indexing and querying. (1) We use the original algorithm to build an NSG [32] ($R = 32$, $EF = 400$), and we apply all our other proposed techniques on it (including the graph refinement). We prepare the data for FastScan on the constructed graph index. This method is denoted as SymQG-NSG. We report the index time of SymQG-NSG and compare it with that of our method in Table 4. It shows that SymQG is at least 2.5x faster than SymQG-NSG in index construction. (2) To test the quality (i.e., the search performance) of the graph

[10]Due to the space limitation, we present more ablation study with different settings of R, EF and K in the technical report [37] .

constructed by our indexing algorithm (see Section 3.2.1), we test SymQG and SymQG-NSG on six moderate-scale datasets. Figure 8 shows that SymQG can always provide competitive (or even slightly better) query performance on all tested datasets compared to SymQG-NSG. These results together indicate that our efficient indexing algorithm speeds up index construction significantly without harming the graph's quality in searching.

Table 4. Indexing time (minutes) for SymQG and SymQG-NSG. For all datasets, $R = 32$ and $EF = 400$ during indexing.

	MSong	SIFT	Deep-1M	GIST	ImageNet	TINY
SymQG	0.8	0.3	0.6	1.7	1.1	4.7
SymQG-NSG	3.0	1.7	2.3	6.2	3.7	22.5

Evaluating the Impact of the Graph Refinement Strategy on Query Performance. To test the effectiveness of the graph refinement strategy (Section 3.2.2), we build indices under same parameters while removing graph refinement process. This is denoted as SymQG (w/o GR). As presented in Figure 8, removing the component causes performance drop on all tested datasets. We also report the average out-degree of the graph SymQG (w/o GR) in Table 5. It shows that without the graph refinement strategy, the number of edges is indeed unaligned with the batch size of FastScan and thus, FastScan would waste much computation as the batch is not full. This helps to explain the performance drop of ablating the graph refinement component.

Table 5. Average out-degree of the graph without graph refinement. With graph refinement, each vertex's out-degree is guaranteed to be 32.

	MSong	SIFT	Deep-1M	GIST	ImageNet	TINY
Degree	19.8	25.9	29.2	18.1	17.9	19.7

4.2.4 Parameter Study. In our algorithm, we introduce a new parameter in indexing, i.e., the number of iterations of Algorithm 2. To illustrate its influence on the quality of the graph for ANN queries, we build indices with different iterations and test their query performance. As presented in Figure 9, when the number of iterations is 3 and larger, the search performance keeps stable on two different datasets. Therefore, we would like to suggest to set the number of iterations in graph construction to be 3 or 4 for our method by default.

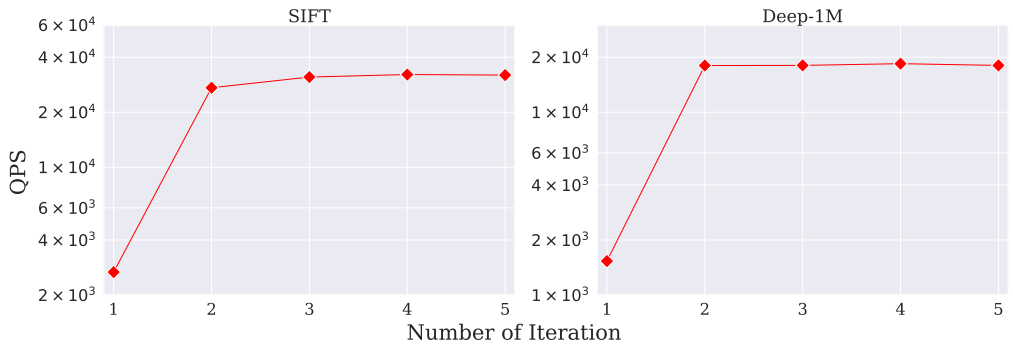


Fig. 9. QPS at 95% recall ($K = 10$) for different number of iterations during indexing of SymQG.

5 RELATED WORK

Existing algorithms for ANN search in high-dimensional vectors can be roughly classified into four categories: the graph-based [10, 18, 19, 30–32, 41, 44, 48, 57, 58, 67, 76], the quantization-based [4, 11, 12, 36, 39, 49, 77], the hashing-based [22, 33, 42, 43, 74, 75] and the tree-based [6, 13, 21, 38]. Graph-based methods offer the state-of-the-art time-accuracy trade-off on real-world datasets [8, 62, 63]. Quantization-based methods compress high-dimensional vectors into short quantization codes [11, 12, 36, 39, 49] and use the compressed codes to estimate distances. FastScan [4, 5] significantly accelerates the distance estimation of these methods, making them an important component in many in-memory ANN libraries [24, 39, 47]. Hashing-based methods promise a theoretical guarantee on the probability of finding c-approximate nearest neighbors. However, they often struggle to achieve high recall with competitive efficiency in practice. Tree-based methods are powerful on low-dimensional datasets but their search performance suffers from the curse of dimensionality in the high-dimensional scenarios. For more comprehensive understanding of different ANN methods, we refer readers to recent tutorials and surveys [9, 23, 27, 56, 64, 66, 68, 72, 79, 81].

Moreover, since graph-based methods offer the state-of-the-art query performance, some existing studies target to optimize certain aspects of indexing and querying for graph-based methods. For example, τ-MNG [67], NSSG [31] and ONNG [46] try to fine-tune the graph structure to obtain a better query performance. FINGER [17] and ADSAMPLING [34] target to reduce the cost of distance computation. Vamana [48] and OG-LVQ [1] focus on decreasing the memory footprint of graph-based indices on large-scale datasets. ELPIS [10] aims at reducing the construction time of graph-based indices. In this paper, we aim to integrate quantization and graph-based indices more symphoniously and leave it as future work to apply and/or adapt these optimization techniques to boost our method.

Besides the ANN query in the Euclidean space, we note that there is a vast literature which concerns the similarity search question for other data types and similarity metrics [25, 26, 60, 65, 66, 88]. We refer readers to surveys and reference books on the similarity search question in more generic metric spaces [66, 69, 86]. Our library focus on the ANN query in the Euclidean space (including the metrics of Euclidean distances and cosine similarity) while its design might provide some insights for other similarity search questions. For example, graph-based indices have also been used for maximum inner product search [60, 88]. It is also possible to enhance these methods with similar ideas of this paper, e.g., conducting re-ranking implicitly and using multiple estimated distances during graph-based searching. However, we note that developing a library for other similarity search questions involves highly non-trivial challenges and workloads, which is out of the scope of this paper. We notice that recently, a thread of studies show the potential of applying the techniques in the similarity search question of data series to that of high-dimensional vectors [25, 26]. It remains to be an interesting question whether a well-optimized library based on these techniques can outperform the libraries based on graph-based indices.

6 CONCLUSION AND DISCUSSION

In conclusion, in this paper, we propose a new method named SymphonyQG which targets to fully unleash the potential of integrating quantization and graph-based indices in a symphonious way. For querying, we incorporate RaBitQ [35] into graph-based indices for its superiority over the conventional PQ method. Moreover, we design a novel search algorithm, by which our method can eliminate the random memory accesses caused by an explicit re-ranking step. In this search algorithm, we propose to use multiple estimated distances in graph-based searching to boost the accuracy. For indexing, we use our efficient search algorithm to accelerate graph index construction. We further propose to refine the graph structure of SymphonyQG such that it is better aligned

with the in-batch computations of FastScan. Based on all the proposed techniques, SymphonyQG establishes the new state-of-the-art in terms of time-accuracy trade-off as well as significantly speeding up the indexing construction.

On the other hand, our method still has certain limitations. Specifically, as discussed in Section 1 and Section 4, SymphonyQG stores duplicated quantization codes for a single vector, which increases the space consumption. Therefore, our method may be unsuitable for memory-limited scenarios (e.g., cloud functions [73]). Also, SymphonyQG can handle ANN query with the metrics of Euclidean distance and cosine similarity only since the quantization method we use (i.e., RaBitQ) is designed for these metrics. More efforts are needed to extend (some of) our techniques for other methods. For future work, we would like to discuss the following possible directions. (1) SymphonyQG has promising performance in indexing. This implies its potential of supporting efficient dynamic updates. For example, for deleting a vertex from the graph, it suffices to re-build the edges of the vertices which have an edge towards the vertex. This can be achieved by running the indexing algorithm for these vertices. (2) The relatively large index size of SymphonyQG limits its application scenarios. Therefore, compression algorithms (e.g., dimension reduction) can be used to reduce its memory consumption.

ACKNOWLEDGEMENTS

This research is supported by the Ministry of Education, Singapore, under its Academic Research Fund (Tier 2 Award MOE-T2EP20221-0013, Tier 2 Award MOE-T2EP20220-0011, and Tier 1 Award (RG20/24)). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore. We would like to thank the anonymous reviewers for providing constructive feedback and valuable suggestions. Additionally, we would like to thank Masajiro Iwasaki, a researcher from Yahoo Japan, for patiently explaining the details of NGT-QG to us.

REFERENCES

[1] Cecilia Aguerrebere, Ishwar Bhati, Mark Hildebrand, Mariano Tepper, and Ted Willke. 2023. Similarity search in the blink of an eye with compressed indices. *arXiv preprint arXiv:2304.04759* (2023).

[2] Nir Ailon and Bernard Chazelle. 2009. The Fast Johnson–Lindenstrauss Transform and Approximate Nearest Neighbors. *SIAM J. Comput.* 39, 1 (2009), 302–322. https://doi.org/10.1137/060673096 arXiv:https://doi.org/10.1137/060673096

[3] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and Optimal LSH for Angular Distance. arXiv:1509.02897 [cs.DS]

[4] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2016. Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan. In *42nd International Conference on Very Large Data Bases*, Vol. 9. 12.

[5] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2017. Accelerated nearest neighbor search with quick adc. In *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval*. 159–166.

[6] Sunil Arya and David M Mount. 1993. Approximate nearest neighbor queries in fixed dimensions.. In *SODA*, Vol. 93. Citeseer, 271–280.

[7] Akari Asai, Sewon Min, Zexuan Zhong, and Danqi Chen. 2023. Retrieval-based language models and applications. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 6: Tutorial Abstracts)*. 41–46.

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

[9] Martin Aumüller and Matteo Ceccarello. 2023. Recent Approaches and Trends in Approximate Nearest Neighbor Search, with Remarks on Benchmarking. *Data Engineering* (2023), 89.

[10] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. Elpis: Graph-based similarity search for scalable data science. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1548–1559.

[11] Artem Babenko and Victor Lempitsky. 2014. Additive quantization for extreme vector compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 931–938.

[12] Artem Babenko and Victor Lempitsky. 2014. The inverted multi-index. *IEEE transactions on pattern analysis and machine intelligence* 37, 6 (2014), 1247–1260.

[13] Alina Beygelzimer, Sham Kakade, and John Langford. 2006. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*. 97–104.

[14] Wei-Cheng Chang, Felix X Yu, Yin-Wen Chang, Yiming Yang, and Sanjiv Kumar. 2020. Pre-training tasks for embedding-based large-scale retrieval. *arXiv preprint arXiv:2002.03932* (2020).

[15] 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. *Proc. VLDB Endow.* 17, 12 (Nov. 2024), 3772–3785. https://doi.org/10.14778/3685800.3685805

[16] Meng Chen, Kai Zhang, Zhenying He, Yinan Jing, and X. Sean Wang. 2024. RoarGraph: A Projected Bipartite Graph for Efficient Cross-Modal Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 17, 11 (Aug. 2024), 2735–2749. https://doi.org/10.14778/3681954.3681959

[17] Patrick Chen, Wei-Cheng Chang, Jyun-Yu Jiang, Hsiang-Fu Yu, Inderjit Dhillon, and Cho-Jui Hsieh. 2023. Finger: Fast inference for graph-based approximate nearest neighbor search. In *Proceedings of the ACM Web Conference 2023*. 3225–3235.

[18] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. 2018. *SPTAG: A library for fast approximate nearest neighbor search.* https://github.com/Microsoft/SPTAG

[19] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. *Advances in Neural Information Processing Systems* 34 (2021), 5199–5212.

[20] Paolo Ciaccia and Marco Patella. 2002. Searching in metric spaces with user-defined and approximate distances. *ACM Transactions on Database Systems (TODS)* 27, 4 (2002), 398–437.

[21] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 426–435.

[22] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*. 253–262.

[23] Magdalen Dobson, Zheqi Shen, Guy E Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2023. Scaling Graph-Based ANNS Algorithms to Billion-Size Datasets: A Comparative Analysis. *arXiv preprint arXiv:2305.04359* (2023).

[24] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. arXiv:2401.08281 [cs.LG]

[25] Karima Echihabi, Panagiota Fatourou, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2022. Hercules against data series similarity search. *arXiv preprint arXiv:2212.13297* (2022).

[26] Karima Echihabi, Theophanis Tsandilas, Anna Gogolou, Anastasia Bezerianos, and Themis Palpanas. 2023. ProS: data series progressive k-NN similarity search and classification with probabilistic quality guarantees. *The VLDB Journal* 32, 4 (2023), 763–789.

[27] Karima Echihabi, Kostas Zoumpatianos, and Themis Palpanas. 2021. New Trends in High-D Vector Similarity Search: Al-Driven, Progressive, and Distributed. *Proc. VLDB Endow.* 14, 12 (jul 2021), 3198–3201. https://doi.org/10.14778/3476311.3476407

[28] faiss. 2022. Fast accumulation of PQ and AQ codes (FastScan). https://github.com/facebookresearch/faiss/wiki/Fast-accumulation-of-PQ-and-AQ-codes-(FastScan). Accessed: 17 Apr, 2024.

[29] FALCONN-LIB. 2015. Library for Fast Fast Hadamard Transform. https://github.com/FALCONN-LIB/FFHT. Accessed: 17 Apr, 2024.

[30] Cong Fu and Deng Cai. 2016. Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph. *arXiv preprint arXiv:1609.07228* (2016).

[31] Cong Fu, Changxu Wang, and Deng Cai. 2022. High Dimensional Similarity Search With Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 8 (2022), 4139–4150. https://doi.org/10.1109/TPAMI.2021.3067706

[32] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2017. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143* (2017).

[33] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 541–552.

[34] Jianyang Gao and Cheng Long. 2023. High-dimensional approximate nearest neighbor search: with reliable and efficient distance comparison operations. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[35] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proc. ACM Manag. Data* 2, 3, Article 167 (may 2024), 27 pages. https://doi.org/10.1145/3654970

[36] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized product quantization. *IEEE transactions on pattern analysis and machine intelligence* 36, 4 (2013), 744–755.

[37] Yutong Gou, Jianyang Gao, Yuexuan Xu, and Cheng Long. 2024. SymphonyQG: Towards Symphonious Integration of Quantization and Graph for Approximate Nearest Neighbor Search. arXiv:2411.12229 [cs.DB] https://arxiv.org/abs/2411.12229

[38] Yan Gu, Zachary Napier, Yihan Sun, and Letong Wang. 2022. Parallel cover trees and their applications. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 259–272.

[39] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*. PMLR, 3887–3896.

[40] Changhun Han, Suji Kim, and Ha-Myung Park. 2024. Efficient Proximity Search in Time-accumulating High-dimensional Data using Multi-level Block Indexing.. In *EDBT*. 547–558.

[41] Ben Harwood and Tom Drummond. 2016. Fanng: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5713–5722.

[42] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment* 9, 1 (2015), 1–12.

[43] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 604–613.

[44] Masajiro Iwasaki. 2016. Pruned bi-directed k-nearest neighbor graph for proximity search. In *International Conference on Similarity Search and Applications*. Springer, 20–33.

[45] Masajiro Iwasaki. 2023. Fusion of graph-based indexing and product quantization for ANN search. https://medium.com/@masajiro.iwasaki/fusion-of-graph-based-indexing-and-product-quantization-for-ann-search-7d1f0336d0d0. Accessed: 17 Apr, 2024.

[46] Masajiro Iwasaki and Daisuke Miyazaki. 2018. Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data. arXiv:1810.07355 [cs.DB]

[47] Yahoo Japan. 2018. Neighborhood Graph and Tree for Indexing High-dimensional Data. https://github.com/yahoojapan/NGT. Accessed: 17 Apr, 2024.

[48] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems* 32 (2019).

[49] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.

[50] William B Johnson and Joram Lindenstrauss. 1984. Extensions of Lipschitz mappings into a Hilbert space 26. *Contemporary mathematics* 26 (1984), 28.

[51] Omar Khattab and Matei Zaharia. 2020. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 39–48.

[52] Yubin Kim. 2022. Applications and future of dense retrieval in industry. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 3373–3374.

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

[54] Huayang Li, Yixuan Su, Deng Cai, Yan Wang, and Lemao Liu. 2022. A survey on retrieval-augmented text generation. *arXiv preprint arXiv:2202.01110* (2022).

[55] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2016. NNS Benchmark: Evaluating Approximate Nearest Neighbor Search Algorithms in High Dimensional Euclidean Space. https://github.com/DBAIWangGroup/nns_benchmark. Accessed: 17 Apr, 2024.

[56] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2019. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (2019), 1475–1488.

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

[58] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[59] Jason Mohoney, Anil Pacaci, Shihabur Rahman Chowdhury, Ali Mousavi, Ihab F Ilyas, Umar Farooq Minhas, Jeffrey Pound, and Theodoros Rekatsinas. 2023. High-throughput vector similarity search in knowledge graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[60] Stanislav Morozov and Artem Babenko. 2018. Non-metric similarity graphs for maximum inner product search. *Advances in Neural Information Processing Systems* 31 (2018).

[61] Javier Vargas Munoz, Marcos A Gonçalves, Zanoni Dias, and Ricardo da S Torres. 2019. Hierarchical clustering-based graphs for large scale approximate nearest neighbor search. *Pattern Recognition* 96 (2019), 106970.

[62] NeurIPS'21. 2021. Billion-Scale Approximate Nearest Neighbor Search Challenge: NeurIPS'21 competition track. https://big-ann-benchmarks.com/neurips21.html Accessed: 17 Apr, 2024.

[63] NeurIPS'23. 2023. NeurIPS'23 Competition Track: Big-ANN. https://big-ann-benchmarks.com/neurips23.html Accessed: 17 Apr, 2024.

[64] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector Database Management Techniques and Systems. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 597–604. https://doi.org/10.1145/3626246.3654691

[65] Marco Patella and Paolo Ciaccia. 2008. The Many Facets of Approximate Similarity Search. In *First International Workshop on Similarity Search and Applications (sisap 2008)*. 10–21. https://doi.org/10.1109/SISAP.2008.18

[66] Marco Patella and Paolo Ciaccia. 2009. Approximate Similarity Search: A Multi-Faceted Problem. *J. of Discrete Algorithms* 7, 1 (mar 2009), 36–48. https://doi.org/10.1016/j.jda.2008.09.014

[67] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient approximate nearest neighbor search in multi-dimensional databases. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[68] Jianbin Qin, Wei Wang, Chuan Xiao, Ying Zhang, and Yaoshu Wang. 2021. High-Dimensional Similarity Query Processing for Data Science. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Virtual Event, Singapore) *(KDD '21)*. Association for Computing Machinery, New York, NY, USA, 4062–4063. https://doi.org/10.1145/3447548.3470811

[69] Hanan Samet. 2005. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[70] Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. 2021. Colbertv2: Effective and efficient retrieval via lightweight late interaction. *arXiv preprint arXiv:2112.01488* (2021).

[71] J Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. 2007. Collaborative filtering recommender systems. In *The adaptive web: methods and strategies of web personalization*. Springer, 291–324.

[72] Larissa C Shimomura, Rafael Seidi Oyamada, Marcos R Vieira, and Daniel S Kaster. 2021. A survey on graph-based methods for similarity searches in metric spaces. *Information Systems* 95 (2021), 101507.

[73] Yongye Su, Yinqi Sun, Minjia Zhang, and Jianguo Wang. [n. d.]. Vexless: A Serverless Vector Data Management System Using Cloud Functions. 2, 3 ([n. d.]), 1–26. https://doi.org/10.1145/3654990

[74] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *Proceedings of the VLDB Endowment* (2014).

[75] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2010. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Transactions on Database Systems (TODS)* 35, 3 (2010), 1–46.

[76] Godfried T Toussaint. 1980. The relative neighbourhood graph of a finite planar set. *Pattern recognition* 12, 4 (1980), 261–268.

[77] Ertem Tuncel, Hakan Ferhatosmanoglu, and Kenneth Rose. 2002. VQ-index: an index structure for similarity searching in multimedia databases. In *Proceedings of the Tenth ACM International Conference on Multimedia* (Juan-les-Pins, France) *(MULTIMEDIA '02)*. Association for Computing Machinery, New York, NY, USA, 543–552. https://doi.org/10.1145/641007.641117

[78] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.

[79] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *Proc. VLDB Endow.* 14, 11 (2021), 1964–1978. http://www.vldb.org/pvldb/vol14/p1964-wang.pdf

[80] Wenping Wang, Yunxi Guo, Chiyao Shen, Shuai Ding, Guangdeng Liao, Hao Fu, and Pramodh Karanth Prabhakar. 2023. Integrity and junkiness failure handling for embedding-based retrieval: A case study in social network search. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 3250–3254.

[81] Zeyu Wang, Peng Wang, Themis Palpanas, and Wei Wang. 2023. Graph-and Tree-based Indexes for High-dimensional Vector Similarity Search: Analyses, Comparisons, and Future Directions. *Data Engineering* (2023), 3–21.

[82] Zeyu Wang, Haoran Xiong, Zhenying He, Peng Wang, and Wei wang. 2024. Distance Comparison Operators for Approximate Nearest Neighbor Search: Exploration and Benchmark. arXiv:2403.13491 [cs.DB] https://arxiv.org/abs/2403.13491

[83] 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. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3152–3165.

[84] Mingyu Yang, Jiabao Jin, Xiangyu Wang, Zhitao Shen, Wei Jia, Wentao Li, and Wei Wang. 2024. Bridging Speed and Accuracy to Approximate K-Nearest Neighbor Search. arXiv:2404.16322 [cs.DB] https://arxiv.org/abs/2404.16322

[85] Leonid Boytsov Yury Malkov. 2018. Hnswlib - fast approximate nearest neighbor search. https://github.com/nmslib/hnswlib. Accessed: 17 Apr, 2024.

[86] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. 2010. *Similarity Search: The Metric Space Approach* (1st ed.). Springer Publishing Company, Incorporated.

[87] 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. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1979–1991.

[88] Zhixin Zhou, Shulong Tan, Zhaozhuo Xu, and Ping Li. 2019. Möbius transformation for fast inner product search on graph. *Advances in Neural Information Processing Systems* 32 (2019).

[89] Zilliz. 2023. Pyglass - Graph Library for Approximate Similarity Search. https://github.com/zilliztech/pyglass. Accessed: 17 Apr, 2024.

Received July 2024; revised September 2024; accepted November 2024