Towards the Distributed Large-scale *k*-NN Graph Construction by Graph Merge

1st Cheng Zhang
Xiamen University
Xiamen, China
zhangcheng@stu.xmu.edu.cn

2nd Wan-Lei Zhao*
Xiamen University
Xiamen, China
wlzhao@xmu.edu.cn

3rd Shihai Xiao
Huawei Technologies Ltd.
China
xiaoshihai@huawei.com

4th Jiajie Yao
Huawei Technologies Ltd.
China
yaojiajie1@huawei.com

5th Xuecang Zhang
Huawei Technologies Ltd.
China
zhangxuecang@huawei.com

***Abstract*—In order to support the real-time interaction with LLMs and the instant search or the instant recommendation on social media, it becomes an imminent problem to build a *k*-NN graph or an indexing graph for the massive number of vectorized multimedia data. In such scenarios, the scale of the data or the scale of the graph may exceed the processing capacity of a single machine. This paper aims to address the graph construction problem of such scale via efficient graph merge. For the graph construction on a single node, two generic and highly parallelizable algorithms, namely Two-way Merge and Multi-way Merge, are proposed to merge subgraphs into one. For the graph construction across multiple nodes, a multi-node procedure based on Two-way Merge is presented. The procedure makes it feasible to construct a large-scale *k*-NN graph/indexing graph on either a single node or multiple nodes when the data size exceeds the memory capacity of one node. Extensive experiments are conducted on both the large-scale *k*-NN graph and the indexing graph construction. For the *k*-NN graph construction, the large-scale and high-quality *k*-NN graphs are constructed by graph merge in parallel. Typically, a billion-scale *k*-NN graph can be built in approximately *17*h when only three nodes are employed. For the indexing graph construction, similar NN search performance as the original indexing graph is achieved with the merged indexing graphs while requiring much less time of construction.**

I. Introduction

The *k*-nearest neighbor graph (*k*-NN graph) is the fundamental data structure in various applications, such as information retrieval [1], [2], recommendation systems [3], [4], machine learning [5], [6], and data mining [7], [8]. Given a vector dataset $C = \{x_i \mid x_i \in \mathbb{R}^d\}, |C| = n$ and a predefined distance measure $metric(\cdot,\cdot)$, the *k*-NN graph *G* is a directed graph built on C. Each entry $G[i]$ is expected to keep *k* nearest neighbors of element x_i. Usually, the neighbors of each vertex in *G* are ranked in ascending order by their distances to the vertex[1]. The time complexity of brute-force *k*-NN graph construction is $O(d{\cdot}n^2)$. The dimensionality d could range from a few tens to several thousands, while the dataset size n may reach the scale of billions. Constructing a million-scale *k*-NN graph in brute-force may cost dozens of hours. The exhaustive *k*-NN graph construction becomes prohibitively expensive as n grows even bigger [9], [10].

The demand in building large-scale graphs also arises from another typical scenario, namely graph-based nearest neighbor search (NN search) [11], [12]. With the development of LLMs [13], their scale increases rapidly. In order to support the real-time interactions with LLMs [12]–[15], the underlying indexing graph that supports the retrieval augmented generation (RAG) [14] must also be scaled up. However, the size of the vectorized multimedia data may exceed the memory capacity of a single machine, let alone the memory cost for the indexing graph. Moreover, graph indexes may be constructed for different subsets of vector data on different contexts [16]. In order to allow the NN search to be carried out on the entire scope, merging multiple graph indexes becomes necessary. Although it is possible to rebuild the whole graph from scratch, the aggregated costs can be substantial since the rebuilding could be frequently required [16], [17].

Due to the high construction costs, recent studies no longer attempt to build *k*-NN graphs with *100%* quality. Several efforts [18]–[23] have proposed efficient solutions for approximate *k*-NN graph construction. Most of these methods follow a divide-and-conquer strategy [18]–[20], [22]. The dataset C is partitioned into small subsets, and a *k*-NN graph is constructed for each subset. Since the overlap between different subsets is introduced, these small *k*-NN graphs are then merged into one by a simple merge sort [18], [19]. However, these methods become inefficient when they are applied to high-dimensional, large-scale data. Typically, building an approximate *k*-NN graph for *1*-million SIFT data by a single thread still takes several tens of minutes [19]. Moreover, these methods lack of genericness due to the inductive biases about the data distribution or distance metrics. A similar strategy has been adopted to build large-scale indexing graph [12]. However, it is not suitable for *k*-NN graph construction, as the resulting *k*-NN graph quality would degrade since the elements from different subsets are not sufficiently cross-matched.

Apart from the above divide-and-conquer methods, the *k*-

Wan-Lei Zhao is the corresponding author.

[1]Without loss of generality, the smaller the distance, the closer the neighbor.

2375-026X/26/$31.00 ©2026 IEEE
DOI 10.1109/ICDE65706.2026.00015

NN graph can also be constructed by the iterative procedure NN-Descent [21]. The construction starts from a randomly initialized graph, which is in very low quality. The graph quality improves gradually as its entries are updated with the closer neighbors through a process called *Local-Join*. The iterative process is driven by the principle that "a neighbor of a neighbor is also likely to be a neighbor". NN-Descent remains as the most popular *k*-NN graph construction method in the literature due to its simplicity, efficiency, and genericness to various distance metrics. In the recent *k*-NN graph construction competition sponsored by SIGMOD[2], the methods from the *top-1* and the *top-3* winners were built upon NN-Descent.

NN-Descent is efficient and supports multi-threading. However, it is designed to run exclusively on a single machine, and it requires the entire dataset as well as the graph under construction to reside in memory throughout the process. The same limitations apply to other existing *k*-NN graph construction methods [18], [19], [22], [24]. Nevertheless, computing resources (e.g., the memory capacity) on a single node are limited regardless how powerful it is. When multiple nodes are available, the dataset can be partitioned into several blocks, and sub-*k*-NN graphs can be constructed in parallel across multiple nodes. The construction time is expected to decrease by multiple folds. Recent studies [9], [25] have attempted to deploy NN-Descent in a distributed setting. In [9], multiple nodes collaboratively run NN-Descent to construct a *k*-NN graph. The method in [25] divides a big dataset by the random division forest. NN-Descent is called to construct a sub-*k*-NN graph for each subset. The final *k*-NN graph is obtained by merging these subgraphs. Unfortunately, both of them require frequent data exchange between nodes. This problem is particularly pronounced with method in [9], where frequent data exchange becomes the processing bottleneck.

This paper aims to address the challenge of large-scale *k*-NN graph and graph index construction in both single-node and multi-node scenarios via graph merge. In particular, we provide efficient solution for the cases where the dataset or graph size exceeds the memory capacity of a single node. Our contributions are threefold.

- A two-way graph merge method is proposed. Based on a smart sampling scheme, it is two times faster than symmetric merge (S-Merge) [17] while maintaining the graph quality on the same level as S-Merge.
- A multi-way merge method is proposed to merge multiple *k*-NN graphs at once. Compared to the two-way merge or S-Merge [17], it shows much higher efficiency while only inducing minor graph quality degradation.
- With the two-way merge, a distributed multi-node graph construction procedure is proposed. It is extremely efficient when multiple nodes are available. Meanwhile, it is also highly flexible that it allows to build the large-scale approximate *k*-NN graph on a single node with the assistance of external storage when its memory capacity cannot hold the full *k*-NN graph or the raw data.

[2]https://2023.sigmod.org/sigmod_awards.shtml

II. Related Works

A. Approximate k-NN Graph Construction

Existing approximate k-NN graph construction methods can be roughly classified into two categories, namely the divide-and-conquer methods [18]–[20] and the neighborhood cross-matching methods [21], [24]. Although the methods based on the divide-and-conquer strategy are efficient, these methods are only applicable to l_p-spaces [19], [20]. In practice, it remains challenging to design a universal partitioning scheme that is effective across various metrics [17], [22].

Unlike the methods in the first category, NN-Descent [21] is feasible to various distance metrics. Owing to both its genericness and efficiency, it remains the most widely used method in the literature. NN-Descent builds the *k*-NN graph starting from a randomly initialized graph and iteratively refines it through the following two steps.

- **Step-1. Sampling** For each element x_i, only a small portion of its new/old neighbors and reverse neighbors are collected.
- **Step-2. Local-Join** With the collected neighbors, the distances of each old-new pairs and new-new pairs are calculated. New edges are subsequently generated and inserted into the corresponding entries of the graph if the distances are sufficiently small.

Intensive memory access is required in both the *Sampling* and *Local-Join* steps. Moreover, it accesses to the raw vectors in the memory in a random order because the neighbor of an element can be any other element in the dataset. Consequently, the costs in data exchange become very high when NN-Descent is performed directly across different nodes. As a result, the multi-node *k*-NN graph construction method proposed in [9] is inefficient. Another multi-node method [25] builds a large-scale *k*-NN graph following the map-reduce fashion. The subgraphs with overlaps are built by NN-Descent and reduced into a complete graph via simple merge sort. Due to the excessive subgraph construction and frequent data exchange across nodes, this method is inefficient either.

The KIFF method [26] constructs a *k*-NN graph by sequentially considering sampled pairs with high co-occurrence frequency. It is highly efficient for high-dimensional data. However, it is specifically designed for sparse datasets. In this paper, we focus on exploring construction solutions for dense and high-dimensional data.

B. Graph index Construction

Owing to their excellent performance on high-dimensional data, graph-based NN search methods are increasingly popular in both academia and industry [27]–[30]. These algorithms perform NN search by leveraging navigational information provided by the graph index to traverse the data space, gradually approaching the query and locating its nearest neighbors. A *k*-NN graph can serve directly as a graph index [1], [2], [31]. However, it only reaches secondary search efficiency. Better performance can be achieved when the *k*-NN graph is further diversified [31]–[33]. In general, the diversification removes

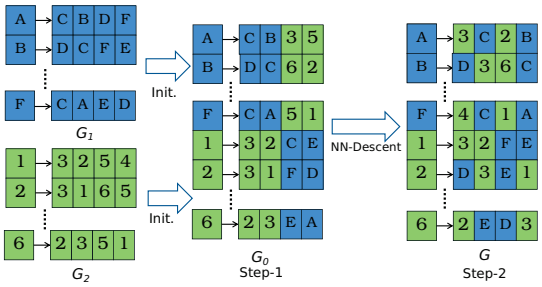


Fig. 1. The illustration of the general steps in S-Merge [17]. The letters and digits are used to differentiate the points from different subsets.

redundant edges to improve NN search efficiency. After the diversification, the resulting graph is no longer a *k*-NN graph but a *relative neighborhood graph* (RNG). Indexing graphs such as DPG [2], NSG [32], Vamana [12], HNSW [11], and FANNG [34] are the instances of RNG.

In general, graph indexes are constructed in two different ways. The first category derives the graph index from a pre-constructed approximate *k*-NN graph [2], [12], [32], where the diversification is applied as a post-processing step. The second category, exemplified by HNSW [11], incrementally builds the graph index by treating each element to be inserted as a query. The diversification is performed on the fly as the new elements are inserted. Although the diversification rules are different in details in methods [2], [11], [12], [32], most of them are largely built upon a *k*-NN graph.

The demand in building large-scale indexing graph out of multiple existing subgraph indexes arises when the graphs have been built for different subsets [17]. Similar to the case in relational database, a larger index is required when different sets of vector data are joined. To save the cost, it is more practical to merge the available graph indexes rather than rebuilding the entire structure from scratch.

C. Graph Merge Method

As pointed out in the above discussion, the demands in merging multiple subgraphs arise in the construction of both *k*-NN graph and graph index. Zhao *et al.* [17] addressed the problem of *k*-NN graph merging by symmetric merge (S-Merge). Given a dataset C, let C_1 and C_2 be disjoint subsets of C such that $C_1 \cap C_2 = C$, with graphs G_1 and G_2 constructed for these subsets. S-Merge first partitions each neighborhood in G_1 and G_2 into two halves. The second half of each neighborhood in G_1 is replaced with randomly selected elements from C_2, while the second half of each neighborhood in G_2 is replaced with randomly selected elements from C_1. The augmented two subgraphs are concatenated as the initial graph on C. This graph is then refined using the NN-Descent procedure [21]. Fig. 1 illustrates the procedure of S-Merge. Since S-Merge already possesses a portion of accurate neighbor information at the beginning, it converges faster than NN-Descent, which starts from a random graph. The task of merging multiple subgraphs can also be accomplished by calling S-Merge by a bottom-up hierarchy.

Unfortunately, neither S-Merge [17] nor hierarchical merging of multiple graphs is efficient. In S-Merge, the first half of the neighbors in a neighborhood are kept and joined in cross-matching during the merge iteration (*Step-2* in Fig. 1). These neighbors are repeatedly sampled by NN-Descent during each iteration, which is unnecessary and time-consuming. Such sampling also leads to numerous futile distance calculation between the old and new neighbors in a neighborhood, which delays the convergence. Moreover, similar to NN-Descent, it requires the memory capacity of a single machine to be large enough to hold the whole dataset as well as the full graph.

In the following sections, we present two efficient algorithms for single node graph construction via subgraph merge. Furthermore, built upon the merge algorithm, a highly flexible and distributed method is designed for large-scale *k*-NN graph/indexing graph construction. It remains effective in both single-node and multi-node scenarios where the memory capacity of one node is smaller than the scale of a dataset.

TABLE I
NOTATIONS USED IN SECTION III AND SECTION IV

Notation	Definition
C	A set of vectors
C_i	The i-th subset of C
G	k-NN graph built on C, $G[i]$ is sorted in an ascending order
$\overline{G}$	The reverse graph of G
G_i	An approximate k-NN graph built on subset C_i
G_i^j	Graph keeps nearest neighbors from C_j for elements in C_i
k	The size of a k-NN graph neighborhood
λ	The number of elements sampled from a neighborhood ($\lambda \leq k$)
$metric(\cdot,\cdot)$	Calculates the distance between two vectors
N_i	The i-th node in the distributed system
$\Omega(\dots)$	Concatenates multiple graphs directly into one
$SoF(x)$	Returns the index of subset that contains the element x
S	Graph keeps sampled neighbors for each element in a set
S_i	Graph keeps sampled neighbors for each element in subset C_i

III. GRAPH CONSTRUCTION VIA MERGE ON A SINGLE NODE

Problem Definition: given $\{C_1, \cdots, C_i, \cdots, C_j, \cdots, C_m\}$ are the subsets, $m \geq 2$, and $\forall i \neq j$, $C_i \cap C_j = \varnothing$[3], the corresponding subgraphs $\{G_1, \cdots, G_m\}$ are built by the same method, such as NN-Descent or HNSW. Each subgraph has a neighborhood size of at most k. The task of graph merge is to build the complete k-NN graph or graph index for $C = \cup_{i=1}^{m} C_i$ based on the subgraphs $\{G_1, \cdots, G_m\}$.

Intuitively, one could perform NN search for each element against all the remaining subgraphs. However, as revealed in [2], [16], this method is inefficient because it is essentially a single-path hill-climbing procedure [16], [21]. Alternatively, one can merge subgraphs by merge sort, when a certain degree of overlap between subsets is allowed [12]. Nevertheless, as m grows, elements from different subsets cannot be sufficiently cross-matched, leading to degraded graph quality. Although one could call NN-Descent to undertake the cross-matching

[3]Merging intersecting elements can be easily handled by merge sort.

directly on the concatenated subgraphs [17], too many futile distance calculations are induced.

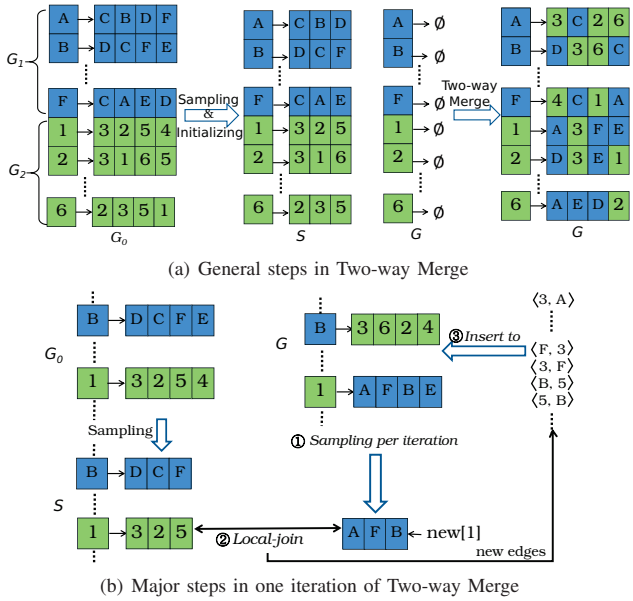
It is easy to see that elements within the same subset require no cross-matching, as they have already been sufficiently connected in the constructed subgraph. The key challenge is how to perform cross-matching efficiently for the elements from different subsets, so that they can discover neighbors from one another. In the following, we introduce two smart cross-matching strategies that fully capitalize on the constructed neighborhood relations within the subgraphs. These two strategies are designed to merge two subgraphs and merge multiple subgraphs at once, respectively.

To facilitate the discussion in the following two sections, several variables and operations are defined. Let $G_0 = \Omega(G_1, \cdots, G_m)$, where $\Omega(\cdot)$ represents the direct concatenation of subgraphs $G_1, \cdots, G_m$. The size of G_0 is n. Obviously, $G_0[i]$ only keeps the nearest neighbors from the subset which the element x_i belongs to. $\overline{G_0}$ is the reverse graph of G_0, in which $\overline{G_0}[i]$ keeps the indices of x_i's reverse neighbors [21]. $\overline{G_0}$ can be derived from G_0. Additionally, an operator $SoF(i)$ is introduced. It returns the index of subset that the element x_i belongs to (notations are summarized in Tab. I).

A. Two-way Merge

Let's consider the graph merge problem with two subgraphs. Specifically, given subgraphs $\{G_1, G_2\}$ constructed on subsets $\{C_1, C_2\}$, let $G_0 = \Omega(G_1, G_2)$ and $C = C_1 \cup C_2$. For $\forall x_i \in C$, its neighbors come from two subsets. The first group of neighbors comes from $SoF(i)$, which have been found during the construction of subgraphs and have been kept in $G_0[i]$. The second group consists of neighbors from the other subset, i.e., $C \backslash SoF(i)$, which the merge process needs to identify. Herein, we introduce graph G, in which $G[i]$ is assumed to keep the neighbors of x_i so far discovered from $C \backslash SoF(i)$. Following the principle that "a neighbor of a neighbor is also likely to be a neighbor" [21], the elements in $G_0[i]$ and $G[i]$ are most likely to be neighbors of each other, with x_i acting as the bridge between them. Cross-matching (*Local-Join*) between $G_0[i]$ and $G[i]$ therefore produces even shorter edges for G. As the merge process progresses, $G[i]$ is continuously updated until it eventually retains the nearest neighbors of x_i from $C \backslash SoF(i)$. This is actually inline with the iteration principle in NN-Descent [21].

From the above analysis, we learn that the quality of graph G can be steadily improved by performing *Local-Join* between $G_0[i]$ and $G[i]$ for all $x_i \in C$. To achieve high graph quality, reverse neighbors from $\overline{G_0}[i]$ and $\overline{G}[i]$ should also participate in *Local-Join*. Moreover, the neighbors that are closer to x_i should have higher priority to participate, since closer neighbors are more likely to be neighbors of each other as well. Therefore, we sample a portion of x_i's neighbors and reverse neighbors for *Local-Join*. For newly found neighbors from $G[i]$ and $\overline{G}[i]$, a cache $new[i]$ is introduced to store the sampled neighbors. Since $G[i]$ is continuously updated, $new[i]$ needs to be re-sampled before each round of *Local-Join*. For neighbors from $G_0[i]$ and $\overline{G_0}[i]$, a supporting graph



(a) General steps in Two-way Merge

(b) Major steps in one iteration of Two-way Merge

Fig. 2. The illustration of Two-way Merge. In contrast to S-Merge [17] and NN-Descent [21], the sampling on the concatenated subgraphs is only undertaken once. The sampled elements for each x_i are kept in S. The sampling on $G[i]$ during the iteration only takes place on the new elements belonging to $C \backslash SoF(i)$ for x_i. The sampled elements are kept in $new[i]$. The *local-join* is undertaken between $S[i]$ and $new[i]$. In the 3rd step of figure (b), we try to insert the edges produced by *local-join* to G. For clarity, the sampling on the reverse graph is not illustrated in the figure.

S is introduced to keep the sampled neighbors from $G_0[i]$ and $\overline{G_0}[i]$. Since G_0 is already built, S is kept unchanged during the iteration (the generation of graph S is illustrated in Fig. 2). In summary, *Local-Join* is carried out between $S[i]$ and $new[i]$ for all $x_i \in C$. The cache $new[i]$ is iteratively refreshed with new elements from the updated $G[i]$ and $\overline{G}[i]$. Moreover, since some neighbors in $G[i]$ may have already been sampled and participated in *Local-Join*, they should be excluded from subsequent sampling to avoid redundant computations. To this end, a flag is introduced for each neighbor in $G[i]$. Newly inserted neighbors are marked as *true*. Once they are sampled, this flag is set to *false* to prevent them from being resampled into $new[i]$.

There are two major operations in our merge algorithm, namely *sampling* and *Local-Join*. Since it is specifically designed for merging two subgraphs, we refer to it as *Two-way Merge*. The complete procedure of Two-way Merge is summarized in Alg. 1. The input parameter λ specifies the maximum number of neighbors in sampling. Graph G is initialized as empty, while each entry of the supporting graph $S[i]$ is initialized with neighbors and reverse neighbors from $G_0[i]$ and $\overline{G_0}[i]$ (*Lines 3–7*). In the first iteration, since $G[i]$ is empty, $new[i]$ is filled with λ randomly selected elements from $C \backslash SoF(i)$ (*Lines 10–11*). Consequently, *Local-Join* is conducted between a group of random elements and $S[i]$ in the first round. In the subsequent iterations, $new[i]$ collects neighbors that are marked as *true* in $G[i]$ (*Line 13*).

Algorithm 1: Two-way Merge in Parallel

Data: subsets C_1, C_2, $metric(\cdot,\cdot)$, k, λ, subgraphs G_1 and G_2
Result: graph of newly found *k*-NN G

1 **begin**
2 $G_0 \leftarrow \Omega(G_1, G_2)$; $C \leftarrow C_1 \cup C_2$;
3 Initialize G;
4 parallel **for** $x_i \in C$ **do**
5 $S[i] \leftarrow$ max λ items in $G_0[i]$;
6 $S[i] \leftarrow S[i] \cup$ max λ items in $\overline{G_0}[i]$;
7 **end**
8 **repeat**
9 parallel **for** $x_i \in C$ **do**
10 **if** *First Iteration* **then**
11 $new[i] \leftarrow \lambda$ random samples in $C \backslash SoF(i)$;
12 **else**
13 $new[i] \leftarrow$ max λ nearest items in $G[i]$ with $true$ flag ;
14 **for** $u \in new[i]$ **do**
15 **if** $R[u].size < \lambda$ **then**
16 $R[u] \leftarrow R[u] \cup x_i$;
17 **end**
18 **end**
19 Mark sampled items as $false$ in $G[i]$;
20 **end**
21 **end**
22 parallel **for** $x_i \in C$ **do**
23 $new[i] \leftarrow new[i] \cup R[i]$;
24 $R[i] \leftarrow \varnothing$
25 **end**
26 parallel **for** $x_i \in C$ **do**
27 **for** $v \in new[i]$, $u \in S[i]$ **do**
28 $dist \leftarrow metric(u, v)$;
29 try insert $\langle u, dist \rangle$ into $G[v]$ with $true$ flag;
30 try insert $\langle v, dist \rangle$ into $G[u]$ with $true$ flag;
31 **end**
32 **end**
33 **until** *Convergence*;
34 $G \leftarrow$ `MergeSort`(G, G_0);
35 **return** G;
36 **end**

Meanwhile, element x_i is collected into $R[u]$ as a reverse neighbor of each its sampled neighbor u (*Lines 14–18*), so that reverse neighbors of x_i are also collected in $R[i]$. Finally, neighbors in $R[i]$ are integrated into $new[i]$ (*Lines 22–25*). At this stage, $new[i]$ contains neighbors and reverse neighbors of x_i from $C \backslash SoF(i)$. Then, through *Local-Join*, elements in $S[i]$ and $new[i]$ find potential neighbors and attempt to update G (*Lines 26–32*). One round of the Two-way Merge is illustrated in Fig. 2(b). This process repeats until convergence. The output of Two-way Merge is the graph G, in which $G[i]$ keeps the approximate nearest neighbors of element x_i from $C \backslash SoF(i)$ at the end. The complete *k*-NN graph on C can be obtained by a simple merge sort between G and G_0.

Discussions In Two-way Merge, the potential nearest neighbors of x_i from $SoF(i)$ and $C \backslash SoF(i)$ are kept in $G_0[i]$ and $G[i]$, respectively. This facilitates the separate sampling on the neighbors from different subsets. The sampling on neighbors from $SoF(i)$ is performed only once and the sampled neighbors are kept in $S[i]$, while the sampling on neighbors from $C \backslash SoF(i)$ only involves the newly inserted ones. In contrast, S-Merge [17] samples all the neighbors repeatedly regardless which subset they are from. Additionally, unlike NN-Descent or S-Merge, the neighbors that have participated in *Local-Join* are excluded from sampling in the following rounds. On the one hand, since such "old" neighbors are no longer sampled, Two-way Merge prevents the prolonged slow improvement in the graph quality when the iteration is close to convergence, which is observed in both NN-Descent [21] and S-Merge [17]. On the other hand, it decreases the number of sampled neighbors. This in turn speeds up the *Local-Join* and allows it to focus on the elements that are most likely to be neighbors of each other.

Besides the smart sampling strategy, Two-way Merge also shows better memory efficiency over S-Merge [17] and NN-Descent. As seen in Alg. 1 *Line 24*, the reverse neighbors of x_i, namely $R[i]$ is cleared right before the *Local-Join* is performed between $S[i]$ and $new[i]$. $R[i]$ will be filled with new reverse neighbors of x_i that are collected in the next iteration. As a result, unlike S-Merge [17] and NN-Descent [21], Two-way Merge will not maintain the complete reverse neighbors of x_i. Instead, it collects them on the fly and releases them immediately after the *Local-Join*. Through the whole procedure, only a small set of reverse neighbors is maintained in $R[i]$. As will be revealed in the experiment, such innovation does not impair the quality of the final graph.

Moreover, since graph entry $G[i]$ only keeps the neighbors collected from $C \backslash SoF(i)$ during the iteration. Two-way Merge is friendly to the multi-node scenario. Given two subgraphs residing on two nodes *A* and *B*, the merge process on *A* can send the supporting graph S to the merge process on *B*. After the Two-way Merge on *B*, the resulting graph G is sent back to *A*. This becomes a distributed graph construction process on two nodes, which will be detailed in Section IV.

B. Post-processing after Merging Graph Indexes

Besides the *k*-NN graph, the proposed Two-way Merge also supports merging two indexing graphs, such as HNSW [11] and Vamana [12] since they can be considered as the derivatives of *k*-NN graphs. Compared to *k*-NN graphs, some edges are removed from each neighborhood in the graph to improve the NN search efficiency [2], [11], [31], [32]. According to [11], given x_a and x_b are two neighbors of x_i, x_b is removed

from x_i's neighborhood if the following inequalities hold:

$$\begin{cases} metric(x_i, x_a) < metric(x_i, x_b) \\ \alpha \cdot metric(x_a, x_b) < metric(x_i, x_b), \end{cases} \quad (1)$$

where $\alpha \geq 1.0$ is a tunable parameter. Since Two-way Merge collects the neighbors purely by distance and keeping neighbors from different subsets, the resulting graph may contain edges that violate the expected structure of the graph index. Thus, diversification becomes essential to enforce the desired neighborhood topology.

The diversification scheme varies for different indexing graph construction methods. In our implementation, the same diversification scheme as the original indexing graph construction method [11], [12] is adopted during the post-processing. In particular, since an HNSW index is a multi-layer graph structure, merging the HNSW indexes is performed on the different layers separately.

C. Multi-way Merge

In addition to the scenario of two subgraphs, there would be more than two subgraphs to be merged on a single node. Intuitively, multiple rounds of Two-way Merge can be undertaken to hierarchically merge individual subgraphs into one, as shown in Fig. 3(a). However, this method is not the most efficient solution in such a case. Following the idea of Two-way Merge, we attempt to merge all subgraphs at once. In this scenario, $G_0 = \Omega(G_1, \cdots, G_m)$ are constructed on subsets $\{C_1, \cdots, C_m\}$, and $C = \cup_{j=1}^{m} C_j$, where $m > 2$.

Following the idea of Two-way Merge, we introduce graphs S and G. Similar to Two-way Merge, $S[i]$ stores the neighbors and reverse neighbors of element x_i sampled from $SoF(i)$. The difference lies in $G[i]$. The newly discovered neighbors kept in $G[i]$ may come from multiple subsets outside $SoF(i)$, i.e., from $C \backslash SoF(i)$. *Local-Join* should be performed between $S[i]$ and the neighbors in $G[i]$ as in Two-way Merge. Moreover, when the neighbors kept in $G[i]$ come from different subsets, they are new to each other. Since they reside in the same neighborhood $G[i]$, they are likely to be neighbors of each other as well. Therefore, additional cross-matching between elements from different subsets in $G[i]$ is necessary. Based on the above analysis, we propose *Multi-way Merge*, which employs a different process from Two-way Merge to merge all subgraphs at once.

To accomplish the aforementioned additional cross-matching, for each x_i, two caches $new[i]$ and $old[i]$ are introduced to keep the sampled neighbors. The cache $new[i]$ collects neighbors newly inserted into $G[i]$ in the previous round, while $old[i]$ collects neighbors that were already inserted in the earlier rounds. Reverse neighbors are collected in $R[i].new$ and $R[i].old$ through an operation similar to Two-way Merge (Alg. 1 *Lines 14–18*) and are subsequently integrated into $new[i]$ and $old[i]$, respectively. Unlike Two-way Merge, *Local-Join* in Multi-way Merge is not only performed between $new[i]$ and $S[i]$, but also performed within $new[i]$, as well as between $new[i]$ and $old[i]$. Note that distance calculations within the same subset are excluded during *Local-Join*.

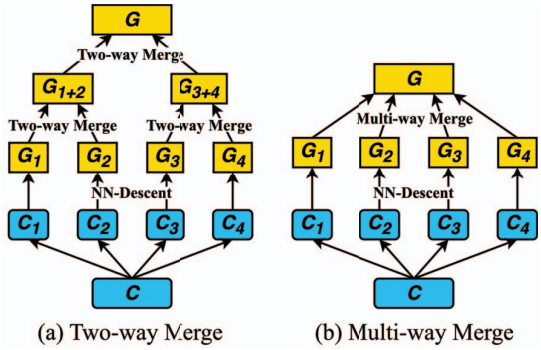


Fig. 3. Two-way Merge by a bottom-up heirarchy versus Multi-way Merge.

Discussions To this end, there are two algorithms available to construct a graph out of multiple subgraphs. Consider the example illustrated in Fig. 3, the full dataset C is divided into four subsets, and four subgraphs have been constructed on these subsets. We want to merge these subgraphs into a single graph on C. The merging task can be accomplished by calling Two-way Merge in a bottom-up hierarchy (as shown in Fig. 3(a)). It calls the Two-way Merge *3* times. For *m* subgraphs, it requires $m-1$ number of calls on Two-way merge. Alternatively, Multi-way Merge can merge all subgraphs (as shown in Fig. 3(b)) at once. Compared to Two-way Merge, Multi-way Merge involves more distance calculations in one call. The question is which algorithm is more efficient when merging multiple subgraphs.

Time Complexities In both Two-way Merge and Multi-way Merge, distance calculations in *Local-Join* are the most computationally intensive operations. During the *Local-Join* in Two-way Merge, elements in $new[i]$ are cross-matched with at most 2λ elements from another subset. Since each $x_i \in C$ participates in $\log_2 m$ merging, each element in $new[i]$ is cross-matched with $2\lambda \cdot t \cdot \log_2 m$ elements in total, where *t* denotes the number of iterations. In Multi-way Merge, *Local-Join* for x_i in each iteration consists of three parts of calculations. These occur between $new[i]$ and $S[i]$, $new[i]$ and $old[i]$, and within $new[i]$. Therefore, each element from $new[i]$ is cross-matched with $3 \cdot 2\lambda \cdot t$ elements at most. Since $new[i]$ contains at most 2λ elements, the time complexity of Two-way Merge and Multi-way Merge is $O(4\lambda^2 \cdot t \cdot n \cdot \log_2 m)$ and $O(3 \cdot 4\lambda^2 \cdot t \cdot n)$, respectively. Clearly, Multi-way Merge is favored when the number of subgraphs (*m*) exceeds *8*. Since the actual number of sampled neighbors and reverse neighbors is typically smaller than the sampling upper bound threshold λ, Multi-way Merge achieves higher efficiency in practice even when $m < 8$ (as shown later in Fig. 9).

IV. Distributed Graph Construction on Multi-Node

Two-way Merge and Multi-way Merge solve the *k*-NN graph construction problem when all subgraphs reside on a

Algorithm 2: Distributed Peer-to-peer Multi-Node Graph Construction on Node $\boldsymbol{N}_i$

Data: Dataset C, k, λ, m, Task i
Result: graph G_i with neighbors from C

1 **begin**
2 $G_i \leftarrow \texttt{NNDescent}(k, C_i)$;
3 $S_i \leftarrow 2\lambda$ neighbors and reverse neighbors for each element sampled from G_i;
4 $iter \leftarrow 1$;
5 **while** $iter \leq \lceil \frac{m-1}{2} \rceil$ **do**
6 $t \leftarrow (i + iter)\%m$;
7 $j \leftarrow (i - iter + m)\%m$;
8 send S_i to Node $\boldsymbol{N}_t$;
9 wait to receive S_j from Node $\boldsymbol{N}_j$;
10 $G_i^j, G_j^i \leftarrow \texttt{TwoWayMerge}(k, C_i, C_j, S_i, S_j)$;
11 $G_i \leftarrow \texttt{MergeSort}(G_i, G_i^j)$;
12 send G_j^i to $\boldsymbol{N}_j$;
13 wait to reclaim G_i^t from $\boldsymbol{N}_t$;
14 $G_i \leftarrow \texttt{MergeSort}(G_i, G_i^t)$;
15 $iter \leftarrow iter + 1$
16 **end**
17 **return** G_i
18 **end**

single node. However, the available resources on a single node are limited, regardless how powerful it is. It is possible that the dataset to be processed is too large to fit into the memory of a single node. Therefore, multi-node execution is necessary for constructing large-scale *k*-NN graphs. Nevertheless, Two-way Merge, Multi-way Merge, and NN-Descent cannot be directly applied in multi-node scenarios, as they all involve intensive memory access. Deploying these algorithms naively across multiple nodes would incur long delays due to the data exchange [9]. An alternative way is to construct subgraphs in parallel across multiple nodes and then merge them either all at once using Multi-way Merge or gradually in a bottom-up hierarchy using Two-way Merge (as shown in Fig. 3). However, such an intuitive solution is infeasible or suboptimal. On the one hand, merging graphs on a node induces significant memory consumption. On the other hand, fewer nodes are involved in the subsequent merges, leading to imbalanced workloads. These challenges underscore the necessity of a parallel graph construction method that can efficiently operate across multiple nodes.

Let's reconsider Two-way Merge from the perspective of a subset. Given C_i and C_j are two disjoint subsets of the dataset C, and the corresponding subgraphs G_i and G_j are constructed for them, respectively. When we perform graph merge between subgraphs G_i and G_j with Alg. 1, the subgraphs do not directly participate in the merge. Instead, an supporting graph S which is composed by two parts, S_i and S_j is employed. S_i keeps the sampled neighbors and reverse neighbors from G_i, and S_j keeps the sampled neighbors from G_j. For node N_i, graph S_i is produced with subgraph G_i. At this point, N_i

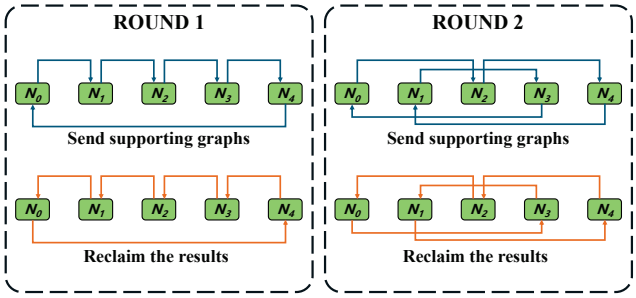


Fig. 4. The illustration of the data flow between different nodes.

only needs to claim S_j from node N_j to complete the merge between G_i and G_j. Due to the property of Alg. 1, the graph (G) after merge can be divided into two parts, denoted as G_i^j and G_j^i. G_i^j is the graph keeps neighbors found from C_j for elements in C_i. G_j^i is in the similar case for subset C_j. Once N_i completes the merge, graph G_i can be updated with G_i^j by merge sort, so that it is joined with the neighbors from C_j for elements in C_i. Meanwhile, G_j^i is sent back to node N_j to update G_j as well. By this way, we realize the subgraph merge between subsets C_i and C_j. In this way, both subgraph construction and Two-way Merge can be executed locally on a single node, avoiding the delays caused by intensive data exchange. Moreover, such way is extensible to the scenario with more than two nodes.

Based on the above analysis, we propose a peer-to-peer *k*-NN graph construction framework built upon graph merge. Alg. 2 illustrates the entire procedure of the proposed method from the perspective of node N_i. Each node retains a copy of the dataset C in advance. At the beginning, subgraph G_i and supporting graph S_i are prepared on node N_i (*Lines 2–3*). In one round of construction, N_i sends its S_i over the network to another node N_t ($t \neq i$). Since each node in the merge plays an equal role, it also receives S_j from N_j (*Lines 6–9*). Then, N_i performs a Two-way Merge using S_i and S_j, producing G_i^j and G_j^i (*Line 10*). Here, G_i^j is used to update G_i, while G_j^i is sent back to N_j (*Lines 11–12*). Similarly, N_t will return G_i^t, which is also used to update G_i (*Lines 13–14*). Clearly, in each round, G_i is updated twice with new neighbors discovered from subsets C_j and C_t (*Lines 11, 14*). Therefore, the entire construction process requires $\lceil \frac{m-1}{2} \rceil$ rounds to complete. Fig. 4 illustrates an example where the construction is carried out using *5* nodes.

In this construction method, the maximum number of distance calculations in the merging procedure is $\frac{m(m-1)}{2} \cdot \lambda^2 \cdot t \cdot n$. In contrast, this number will be only $\lambda^2 \cdot t \cdot n \cdot \log_2 m$ using the Two-way Merge in a bottom-up hierarchical manner (as shown in Fig. 3(a)). However, this multi-node graph construction is favored for at least two reasons. Firstly, due to the high degree of parallelization across multiple nodes, this method is more efficient as it ensures optimal workload balancing. Secondly, memory consumption on each node remains stable. It only requires the memory to hold the subset of vectors, the

TABLE II
OVERVIEW OF DATASETS

Name	d	LID [35]	metric($\cdot$,$\cdot$)	Type	Data Size
SIFT1M [36]	128	15.6	L2	SIFT [37]	10^6
DEEP1M [38]	96	15.9	L2	DEEP	10^6
SPACEV1M [39]	100	23.2	L2	Text	10^6
GIST1M [40]	960	25.9	L2	GIST [40]	10^6
SIFT100M	128	15.6	L2	SIFT	10^8
DEEP100M	96	15.9	L2	DEEP	10^8
SIFT1B	128	15.6	L2	SIFT	10^9

corresponding subgraph and the supporting subgraphs. It is, therefore, particularly useful when the scale of the graph or the dataset exceeds the memory capacity of a single node.

Alg. 2 can also run on a single node to build a large-scale *k*-NN graph when there is no sufficient memory to hold the dataset or the graph. On the first hand, the dataset is divided into subsets whose size fits into the memory capacity. The subgraph for each subset is constructed one by one. The graph merge can be undertaken between two subgraphs in each round. Other subgraphs and their corresponding vectors are kept in the external storage. The resulting subgraphs G_i and G_j in each round are saved back to the external storage. Another two subgraphs with their vectors are swapped in for the next round of merge. Since the size of the subgraphs to be merged is fixed, it becomes achievable to build the large graph on a single node by following the same flow of pairwise merges shown in Alg. 2.

In the real scenario of graph construction on multiple nodes, it is possible that one node cannot fulfill the assigned task due to its limited memory capacity. In this case, the assigned subset on each node can be further divided into smaller subsets whose size fits into the memory capacity. An instance of Alg. 2 will work alone on each node with the assistance of external storage to build and merge the smaller subgraphs on the first hand. With the constructed subgraphs on each node, another instance of Alg. 2 runs across the multiple nodes to build the graph for the entire dataset. To this end, it is clear to see Alg. 2 is highly flexible to the available computing resources. Apparently, it is increasingly efficient when more computing resources are available.

V. EXPERIMENTS

This section evaluates the performance of Two-way Merge, Multi-way Merge, and multi-node graph construction on large-scale *k*-NN graph and indexing graph construction tasks. Seven real-world datasets are used in the evaluation, including four *1*-million datasets, two *100*-million datasets, and one *1*-billion dataset. A summary of these datasets is provided in Tab. II. All datasets are dense and high-dimensional. We use *Local Intrinsic Dimensionality* (LID) [35] (the *3rd* column in Tab. II) to measure the difficulty of a dataset. Datasets with high LID are more challenging for graph construction and NN search.

A. Evaluation Protocol

For the *k*-NN graph construction task, the performance is evaluated by studying the curves of construction time versus the quality of the *k*-NN graph. NN-Descent [21] is used to construct subgraphs for our merge methods. At the same time, it is also treated as the comparison baseline. The quality of the *k*-NN graph is assessed using top-*10* recall (*Recall@10*) and top-*100* recall (*Recall@100*). Given $R(i,k)$ is the number of true-positive neighbors in top-*k* nearest neighbor list of element x_i, the top-*k* recall for the entire dataset is given as

$$Recall@k = \frac{\sum_{i=1}^{n} R(i,k)}{n \times k}.$$

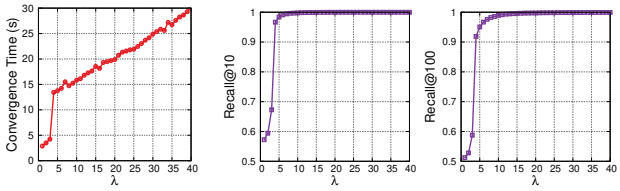
For the indexing graph construction task, the graph quality cannot be reflected by the top-*k* recall. Instead, we evaluate the NN search performance on the indexing graphs that are constructed by our merge methods. We study the curve of top-*10* and top-*100* recall of NN search versus the number of queries processed in one second (*Queries Per Second*). In the experiments, the performance of the indexing graphs built by our merge method is compared to those are directly built by HNSW [11] or Vamana [12]. Both of them are popularly adopted in various large-scale NN search tasks.

The experimental evaluation covers two scenarios, namely single node and multi-node. The experiments about the graph construction on a single node are carried out on a machine equipped with dual *40*-core *3.0*GHz CPUs and *1280*GB DDR4 memory. All codes are compiled with standard C++ compiler on Ubuntu *22.04*. For graph merge and construction experiments, all CPU cores are utilized with OpenMP support, whereas NN search experiments are conducted on a single core. For multi-node graph construction experiments, another eight machines are employed, namely seven with dual *40*-core *3.0*GHz CPUs and *256*GB DDR4 memory, and another with dual *24*-core *3.2*GHz CPUs and *256*GB DDR4 memory. Data exchange between machines is supported by OpenMPI. For all the following experiments, each run of graph merge algorithms uses two subgraphs with the same size and maximum number of edges. The merged graphs retain the same maximum number of edges as their corresponding subgraphs.

Before we present the performance of our methods on both single-node and distributed graph construction tasks, we validate the parameter settings in our methods.

B. Study on the Parameter Settings

1) Parameter Selection: This section studies the impact of parameter settings on the performance of the Two-way Merge algorithm. Two major parameters are involved in Two-way Merge. The first is the neighborhood size k, which is determined based on practical user requirements. A typical range for k is $[40,\ 200]$. Another parameter is the maximum number of samples taken from each element's neighborhood λ. For this reason, λ is no bigger than the neighborhood size k, i.e., $\lambda \leq k$. A larger λ leads to the higher graph quality, while it also makes the algorithm take longer time to converge. Given k is fixed, varying λ leads to the graphs in different qualities and different computation time costs. This section focuses on analyzing the trend of graph quality and merge time under varying λ, with k is fixed at *100* in the experiments.



(a) Time cost vs. λ. (b) Recall@10 vs. λ.

Fig. 5. The impact of λ on the performance of Two-way Merge. $k = 100$

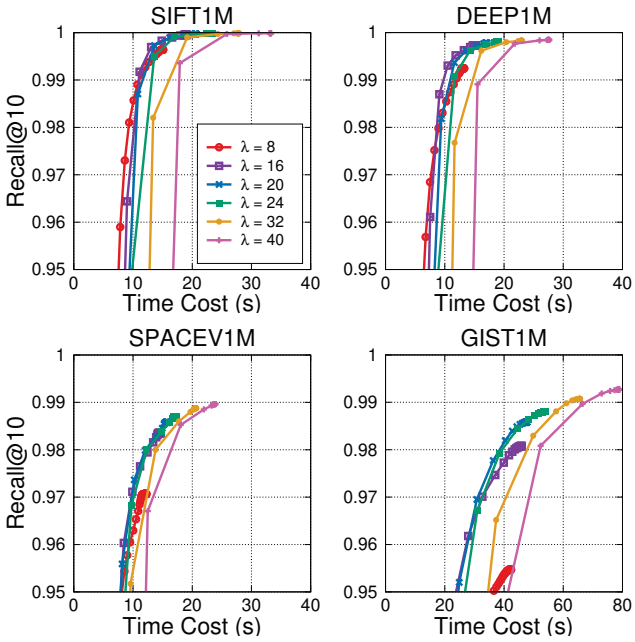


Fig. 6. Recall versus time cost with different λ. $k = 100$

Fig. 5 presents the performance of Two-way Merge with different λ on the SIFT1M dataset. In the figures, time and recall are presented at the point of full convergence. It is clear to see that both convergence time and graph quality increase as λ grows. Both the time cost and graph quality remain low when $\lambda < 4$. A significant improvement in graph quality is observed, accompanied by substantially higher computation costs when $\lambda \geq 4$. When $\lambda > 4$, graph quality improves slowly, whereas time cost increases linearly.

Fig. 6 further shows the curves of graph quality versus graph construction time under different λ settings on two *1*-million datasets. On the one hand, different λ values result in significantly different graph qualities when $\lambda \leq 20$, while there are only minor differences on time consumption to achieve the same recall. On the other hand, higher λ only leads to minor graph quality improvement, while the time consumption increases significantly when $\lambda \geq 20$. The setting of λ is also closely related to local intrinsic dimensionality (LID) [35], where $LID \leq d$. For datasets with low LID, such as SIFT, a small λ is sufficient to achieve high graph quality. However, for datasets with high LID, such as GIST, a larger λ is required

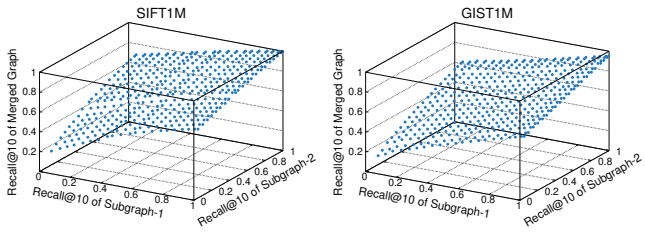


Fig. 7. Correlation between the quality of subgraphs and quality of the merged graph.

to ensure the high graph quality. Typically, a reasonable range for λ is $[16,\ 24]$, providing a good balance between time costs and graph quality.

2) Impact of Subgraph Quality: This section studies the impact of the subgraph quality on the quality of the merged graph using Two-way Merge algorithm. On the one hand, the neighbors supplied by each subgraph are potentially the final neighbors in the merged graph. The quality of the subgraph has a direct impact on the quality of the merged graph. On the other hand, the high-quality neighbors in one subgraph are also helpful for the elements from another subgraph to find their true neighbors. Our study about the relationship between the subgraph quality and merged graph quality is carried out on two *1*-million datasets SIFT1M and GIST1M. The parameters k and λ are fixed to 100 and 20, respectively. Fig. 7 shows the relationship between the quality of the merged graph and the quality of the two subgraphs. As shown from the figures, the quality of the merged graph is positively correlated with the quality of the two subgraphs. When both subgraphs have sufficiently high quality, the *Recall@10* of the merged graph approximates the average *Recall@10*s of the two subgraphs. In terms of the time costs in graph merge, it remains around *17*s and *28*s respectively on SIFT1M and GIST1M. No noticeable correlation with subgraph quality is observed.

C. Performance on k-NN Graph Construction

In this section, the effectiveness of merge algorithms is studied in comparison with S-Merge[4] on four *1*-million datasets and two *100*-million datasets. The neighborhood size k is fixed at *100* for comparative experiments. Parameter λ is fixed at *20* for all four methods. For each dataset, two subgraphs are constructed by NN-Descent[5] in advance with the above parameter settings. It takes around *10*s (*45*s for GIST1M) to construct a subgraph for *1*-million datasets, and around *2500*s for a subgraph of a *100*-million datasets. Two-way Merge and S-Merge are called to construct the k-NN graph based on these subgraphs. The curves of time cost versus *Recall@10* are shown in Fig. 8. The results from NN-Descent are also presented when it is called to construct k-NN graph directly.

As shown in Fig. 8, Two-way Merge is at least *2* times faster than S-Merge to achieve the same recall. Compared to the graph constructed by NN-Descent from scratch, Two-way

[4]https://github.com/wlzhao22/nn-merge
[5]https://github.com/aaalgo/kgraph

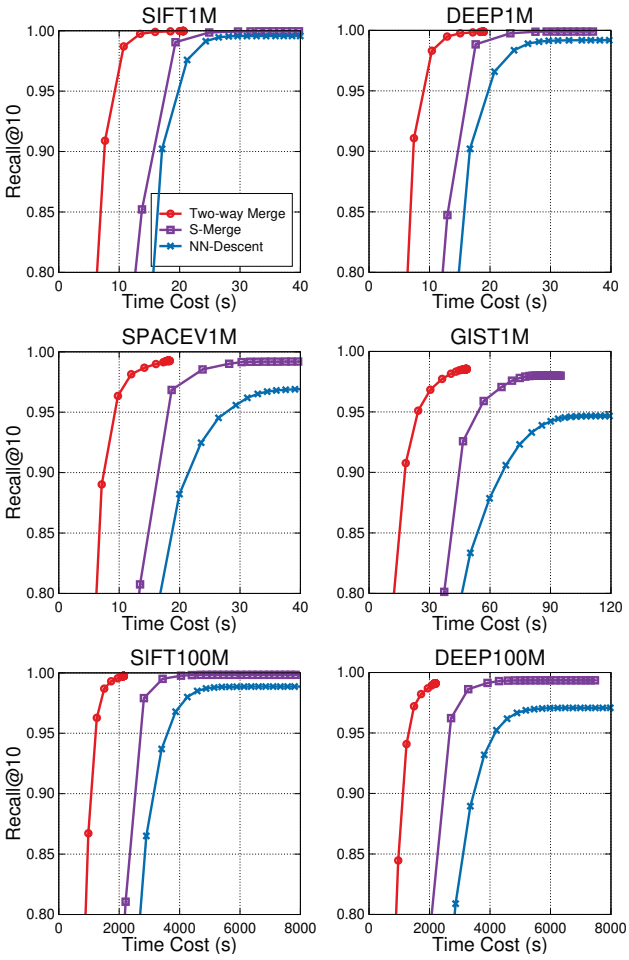


Fig. 8. *Recall@10* versus time cost by Two-way Merge, S-Merge and NN-Descent on six *1*-million datasets.

Merge requires only about *1/3* of the time while achieving significantly higher recall. As observed from the top tail of the curves, it takes more time for both S-Merge and NN-Descent to make minor improvements in the graph quality. This is largely attributed to the sampling strategy in Two-way Merge. The supporting graph S, whose entries keep the sampled elements from G_0, remains fixed throughout the process. The unnecessary samplings and cross-matchings are largely avoided when the procedure is close to convergence.

Moreover, the behaviors of Two-way Merge and Multi-way Merge are studied when they are called to construct k-NN graph out of different number of subgraphs. The experiments are conducted on SIFT1M and DEEP1M. The datasets are divided into $2, 4, \cdots, 64$ subsets, respectively. Each group of the subsets has an equal size. The subgraphs were constructed by NN-Descent in advance. Thereafter, Two-way Merge and Multi-way Merge are called respectively to construct the k-NN graph of *1*-million scale out of each group of the subsets. For Two-way Merge, it builds the graph by a bottom-up hierarchy (shown in Fig. 3(a)). For Multi-way Merge, it merges all the

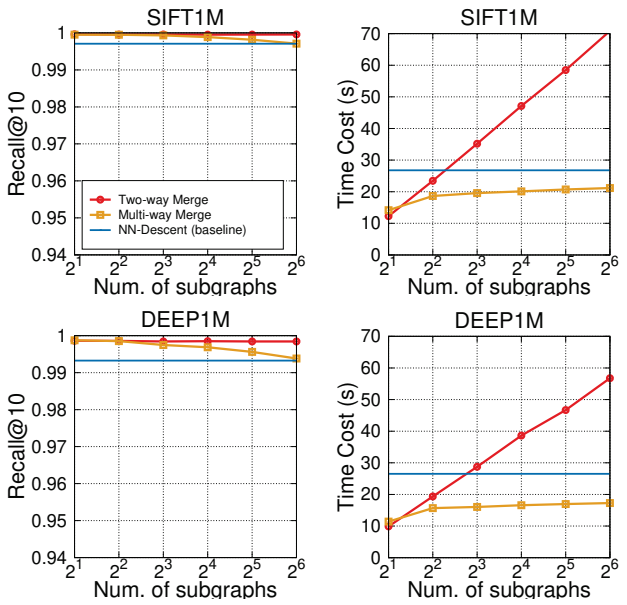


Fig. 9. The trend of *Recall@10* (left) and time cost (right) as the number of subgraphs increases.

subgraphs at once. The quality of the constructed graph and the corresponding time costs are shown in Fig. 9.

As seen from Fig. 9, the graph quality of Two-way Merge remains stable as the number of subsets increases. In contrast, the graph quality from Multi-way Merge drops gradually. This is reasonable because significantly fewer cross-matchings are performed in Multi-way Merge. This leads to the miss of true neighbors. Nevertheless, the graph quality drops slowly (only $0.002 \sim 0.003$). On the one hand, it always takes more time for both Two-way Merge and Multi-way Merge to merge more number of subgraphs when the dataset size is fixed. On the other hand, to merge the same number of subgraphs, the time costs from Multi-way Merge is increasingly lower than Two-way Merge when the number of subgraphs increases. In summary, Multi-way Merge is favored over Two-way Merge when we merge multiple graphs on the one node.

D. Performance on Indexing Graph Construction

In this section, we show the feasibility of building large indexing graphs by graph merge. Two representative indexing graphs are considered in the experiments, namely HNSW[6] and Vamana[7]. The NN search performance achieved on the merged indexing graphs is compared with the graphs constructed by the original methods. Two *100*-million datasets SIFT100M and DEEP100M are adopted in the experiment. Each dataset is divided into *2*, *4*, and *8* subsets. HNSW and Vamana are adopted to construct the subgraphs for each group of subsets. The parameter settings are loyal to the original papers. For HNSW, the parameters are set to $M = 32$ and $EF = 512$. For

[6]https://github.com/nmslib/hnswlib
[7]https://github.com/microsoft/DiskANN

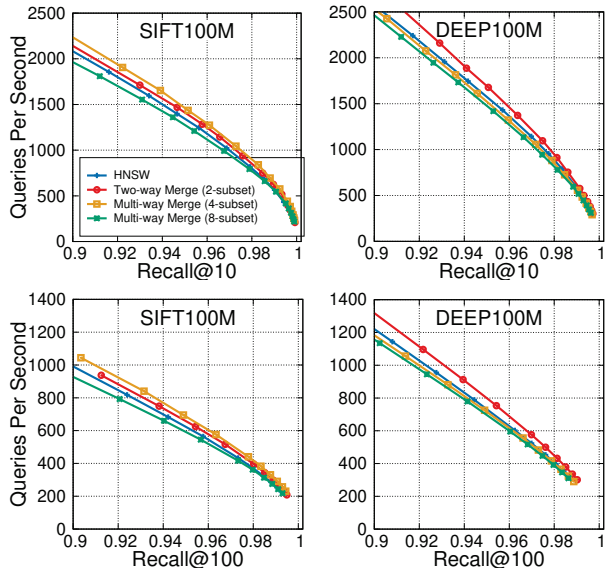


Fig. 10. NN search performance from merged indexing graphs and the graphs built directly from scratch by HNSW.

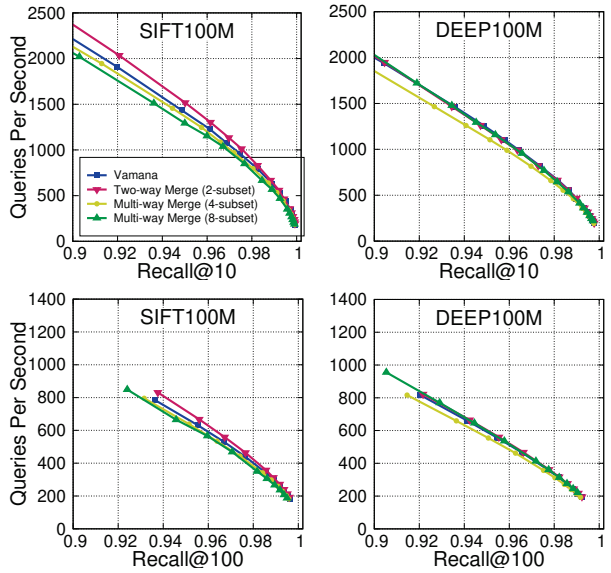


Fig. 11. NN search performance from merged indexing graphs and the graphs built directly from scratch by Vamana.

Vamana, the parameters are set to $R = 64$ and $L = 256$. Under these parameter settings, both methods construct indexing graphs with a maximum of *64* edges in each neighborhood. For both HNSW and Vamana, the time costs of building a graph in half-size is roughly from *1/3* to *1/2* of time costs in building a full one. The parameter k for graph merge algorithms is generally set to the maximum neighborhood size in the subgraphs. Therefore, it is fixed at *64*. Two-way Merge and Multi-way Merge are called to build the full indexing graph based on the produced subgraphs.

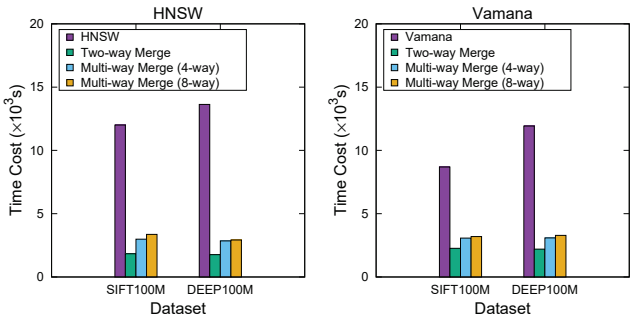


Fig. 12. Time costs in merging and construction of HNSW (left) and Vamana (right) graphs

Fig. 10 and Fig. 11 show the NN search performance on two datasets. The performance from HNSW and Vamana are treated as the baselines. As shown in the figures, the search performance of the merged graphs is comparable to that of the original graphs. In particular, the search performance on the graphs constructed by Two-way Merge is *1*$\sim$*2*% better due to their higher graph quality. For the graphs constructed by Multi-way Merge, the performance fluctuates slightly compared to those built by the original algorithms. Nevertheless, the difference is less than *5*%. Fig. 12 further compares the time costs in graph merge to that of building the graphs from scratch. As shown in the figure, the time required for graph merge is significantly lower. Therefore, it is favored to construct the indexing graph by merge over building it from scratch when the subgraphs are ready.

E. Performance on Distributed Graph Construction

In this section, the proposed distributed graph construction method on multiple nodes is tested on the large-scale k-NN graph construction tasks. Namely, *3* $\sim$ *9* servers connected by *1000*Mbps network are employed to construct k-NN graphs for SIFT100M, DEEP100M, and SIFT1B. The performance from NN-Descent is treated as the baseline. The results from GNND [41] and Faiss[8] IVF-PQ [10], both of which run on an *NVIDIA RTX 3090* GPU, are also included. In addition, the k-NN graph construction results based on the strategy proposed in DiskANN [12] are also reported.

In order to build k-NN graph of such scale, Alg. 2 runs both on single-node mode and multi-node mode. On the single mode, the subset assigned to each node is further divided into smaller *4* subsets each of which fits to the capacity (only *256*GB available) of a single node. The smaller subgraphs are constructed on one node with the assistance of external storage (an SSD with maximum sequential read/write speed of *7450/6900* MB/s). This involves *4* times of subgraphs construction and *6* rounds of two-way merge, which is carried out by an instance of Alg. 2 on each node. Once the subgraphs on each node are ready, another instance of Alg. 2 runs on multi-node mode. It performs several rounds of Two-way

[8]https://github.com/facebookresearch/faiss

TABLE III
TIME COSTS AND THE GRAPH QUALITY IN THE CONSTRUCTION OF LARGE-SCALE k-NN GRAPH ON THREE NODES. NN-DESCENT IS PULLED OUT ON THE FIRST SERVER AS THE EXPERIMENTS BEFORE

	SIFT100M		DEEP100M		SIFT1B	
Dataset	↓Time	↑*Recall@10*	↓Time	↑*Recall@10*	↓Time	↑*Recall@10*
Multi-node Cons.	**2,145**s	**0.991**	**2,127**s	**0.975**	**17.2**h	**0.991**
NN-Descent [21]	5,143s	0.988	5,461s	0.970	-	-
GNND [41]	3,033s	0.966	2,888s	0.956	77.0h	0.955
IVF-PQ [10]	4,469s	0.730	4,262s	0.770	-	-

Merge between the subgraphs from multiple nodes to build the k-NN graph for the entire dataset.

The construction performance on three nodes are presented in Tab. III. For our multi-node graph construction method, time costs in data exchange and storage access are all considered. As shown in the table, for the two *100*-million datasets, the multi-node graph construction method requires around *2/5* of the total time costs of NN-Descent, and much less time than GNND and Faiss IVF-PQ, while achieving higher graph quality. For the SIFT1B dataset, the multi-node graph construction method is *4* times faster than GNND and achieves significantly higher graph quality. Subjected to the memory capacity of a single node, NN-Descent is unable to construct the k-NN graph for SIFT1B. In contrast, the superior performance is achieved by the multi-node merge in terms of both efficiency and graph quality on all three datasets. It shows its unique advantage in constructing large-scale k-NN graph. Compared to the results presented in [9], our method is considerably more effective in the sense that it shows lower time costs in building a k-NN graph under considerably lower hardware configurations. While the neighborhood size of our graph is *5* times bigger.

We also tested the feasibility of building large-scale k-NN graph by the indexing graph merge strategy used in DiskANN [12], which is originally designed for large-scale indexing graph construction. The dataset is partitioned into overlapping subsets by k-means with multiple assignments. Then NN-Descent is adopted to build k-NN graphs for the subsets on different nodes in parallel. The complete k-NN graph on the entire dataset is constructed by reducing multiple neighbor lists of each element from multiple subgraphs. Typically, when SIFT100M is divided into *21* overlapping subsets, the k-NN graph quality only reaches *Recall@10* of *0.855*. While for DEEP100M, the *Recall@10* reaches *0.827* only. The low graph quality is mainly due to the insufficient cross-matching between elements from different subsets.

Moreover, we also study the efficiency of Alg. 2 when different number of nodes are employed in the construction (shown in Fig. 13). Correspondingly, the percentages of the key operations accounting for the total time costs are shown in Fig. 14. For the *1*-billion dataset, due to the limited memory, the subset assigned to each node is further divided into the smaller subsets. Therefore, the multi-node graph construction algorithm (Alg. 2) is undertaken both on the node with the assistance of external storage and across different nodes. As seen from Fig. 13, the time costs drop steadily when more nodes are employed. Nevertheless, the gained efficiency by

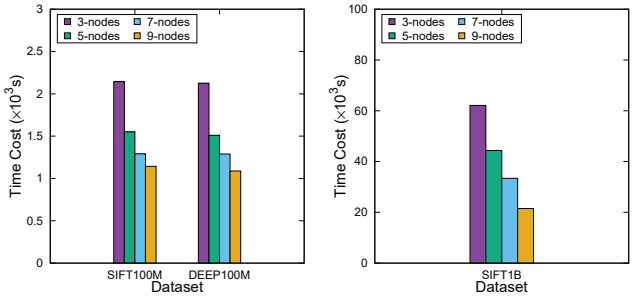


Fig. 13. The time efficiency of distributed graph construction for three datasets when different number of nodes are employed.

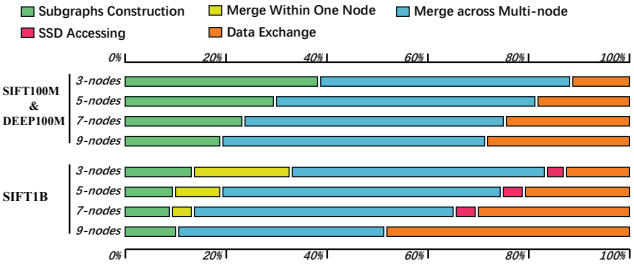


Fig. 14. The time cost percentage from each type of operation accounting for the total cost in multi-node graph construction.

employing more nodes drops due to the increasing time costs in data exchange between nodes. As seen from Fig. 14, the time costs in data exchange account for nearly *50%* when *9* nodes are employed in the construction.

VI. CONCLUSION

In this paper, we have thoroughly studied the problem of building k-NN graphs/indexing graphs out of multiple subgraphs. The efficient graph construction is achieved by the parallel graph merge in both single node and multiple-node scenarios. In the single node scenario, Two-way Merge and Multi-way Merge are built upon a smart *Local-Join* strategy that performs the cross-matching between the elements from different subgraphs. Compared to the existing graph merge method, both algorithms show considerably higher efficiency while maintaining the graph quality, which has been confirmed in the large-scale k-NN graph construction, indexing graph construction, as well as the NN search tasks.

More importantly, based on the Two-way Merge, a highly flexible and distributed graph construction procedure is proposed. On the one hand, it allows to build large-scale k-NN graph on a single-node when its memory is insufficient to hold the entire dataset. With the assistance of the external storage, the complete graph can be built by multiple rounds of subgraph merge that can be fit into the memory capacity of one node. On the other hand, it makes the parallel k-NN graph construction on the multi-node become an easy task to undertake. As illustrated in the billion-scale experiments, it demonstrates significantly higher efficiency and superior graph quality than the state-of-the-art methods.

AI-Generated Content Disclosure

Acknowledgement

No AI generated contents are used in this paper.

References

[1] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang, "Fast approximate nearest-neighbor search with k-nearest neighbor graph," in *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, 2011, pp. 1312–1317.

[2] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin, "Approximate nearest neighbor search on high dimensional data experiments, analyses, and improvement," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 8, pp. 1475–1488, 2020.

[3] Y. Wang, "Improving collaborative filtering recommendation via graph learning," 2023. [Online]. Available: https://arxiv.org/abs/2311.03316

[4] L. Boytsov, D. Novak, Y. Malkov, and E. Nyberg, "Off the beaten path: Let's replace term-based retrieval with k-NN search," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, 2016, pp. 1099–1108.

[5] Z. Abu-Aisheh, R. Raveaux, and J.-Y. Ramel, "Efficient k-nearest neighbors search in graph space," *Pattern Recognition Letters*, vol. 134, pp. 77–86, 2020.

[6] O. Boiman, E. Shechtman, and M. Irani, "In defense of nearest-neighbor based image classification," in *Proceedings of the 2008 IEEE Conference on Computer Vision and Pattern Recognition*, 2008, pp. 1–8.

[7] D. A. Adeniyi, Z. Wei, and Y. Yongquan, "Automated web usage data mining and recommendation system using k-nearest neighbor (KNN) classification method," *Applied Computing and Informatics*, vol. 12, no. 1, pp. 90–108, 2016.

[8] M. R. Brito, E. L. Chávez, and et al., "Connectivity of the mutual k-nearest-neighbor graph in clustering and outlier detection," *Statistics & Probability Letters*, vol. 35, no. 1, pp. 33–42, 1997.

[9] K. Iwabuchi, T. Steil, B. Priest, and et al., "Towards a massive-scale distributed neighborhood graph construction," in *Proceedings of the Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 730–738.

[10] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535 – 547, 2021.

[11] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, 2020.

[12] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "DiskANN: Fast accurate billion-point nearest neighbor search on a single node," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 2019, pp. 13 766–13 776.

[13] S. Minaee, T. Mikolov, N. Nikzad, M. Chenaghlu, R. Socher, X. Amatriain, and J. Gao, "Large language models: A survey," 2024. [Online]. Available: https://arxiv.org/abs/2402.06196

[14] W. Fan, Y. Ding, L. Ning, S. Wang, H. Li, D. Yin, T.-S. Chua, and Q. Li, "A survey on RAG meeting LLMs: Towards retrieval-augmented large language models," in *Proceedings of the 30th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining*, 2024, pp. 6491–6501.

[15] J. Sun, G. Li, J. Pan, J. Wang, Y. Xie, R. Liu, and W. Nie, "GaussDB-Vector: a large-scale persistent real-time vector database for LLM applications," in *Proceedings of VLDB Endowment*, vol. 18, no. 12, 2025, pp. 4951 – 4963.

[16] W.-L. Zhao, H. Wang, and C.-W. Ngo, "Approximate k-NN graph construction: A generic online approach," *IEEE Transactions on Multimedia*, vol. 24, pp. 1909–1921, 2022.

[17] W.-L. Zhao, H. Wang, and et al., "On the merge of k-NN graph," *IEEE Transactions on Big Data*, vol. 8, no. 6, pp. 1496–1510, 2022.

[18] J. Chen, H. ren Fang, and Y. Saad, "Fast approximate kNN graph construction for high dimensional data via recursive lanczos bisection," *Journal of Machine Learning Research*, vol. 10, no. 69, pp. 1989–2012, 2009.

[19] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li, "Scalable k-NN graph construction for visual descriptors," in *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 1106–1113.

[20] Y.-M. Zhang, K. Huang, G. Geng, and C.-L. Liu, "Fast kNN graph construction with locality sensitive hashing," in *Proceedings of the 2013 European Conference on Machine Learning and Principles and Practice of Knowledge Discovery*, 2013, pp. 660–674.

[21] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," in *Proceedings of the 20th International Conference on World Wide Web*, 2011, pp. 577–586.

[22] C. Fu and D. Cai, "EFANNA: An extremely fast approximate nearest neighbor search algorithm based on kNN graph," 2016. [Online]. Available: https://arxiv.org/abs/1609.07228

[23] M. Connor and P. Kumar, "Fast construction of k-nearest neighbor graphs for point clouds," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 4, pp. 599–608, 2010.

[24] J.-F. Wang, W.-L. Zhao, S. Xiao, J. Yao, and X. Zhang, "Dynamic NN-Descent: An efficient k-NN graph construction method," *IEEE Transactions on Big Data*, vol. 11, no. 2, pp. 879–886, 2025.

[25] S.-H. Kim and H.-M. Park, "Efficient distributed approximate k-nearest neighbor graph construction by multiway random division forest," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 1097–1106.

[26] A. Boutet, A.-M. Kermarrec, N. Mittal, and F. Taiani, "Being prepared in a sparse world: The case of KNN graph construction," in *Proceedings of the IEEE 32nd International Conference on Data Engineering*, 2016, pp. 241–252.

[27] J. Vargas Muoz, M. A. Gonalves, Z. Dias, and R. da S. Torres, "Hierarchical clustering-based graphs for large scale approximate nearest neighbor search," *Pattern Recognition*, vol. 96, p. 106970, 2019.

[28] C. Feng, D. Lian, X. Wang, Z. Liu, X. Xie, and E. Chen, "Reinforcement routing on proximity graph for efficient recommendation," *ACM Transactions on Information System*, vol. 41, no. 1, pp. 1–27, 2023.

[29] X. Wan and J. Xiao, "Exploiting neighborhood knowledge for single document summarization and keyphrase extraction," *ACM Transactions on Information System*, vol. 28, no. 2, pp. 1–34, 2010.

[30] Y. Zhang, F. Sun, X. Yang, C. Xu, W. Ou, and Y. Zhang, "Graph-based regularization on embedding layers for recommendation," *ACM Transactions on Information System*, vol. 39, no. 1, pp. 1–27, 2020.

[31] M. Wang, X. Xu, Q. Yue, and Y. Wang, "A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search," *Proceedings of VLDB Endowment*, vol. 14, no. 11, pp. 1964–1978, 2021.

[32] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," *Proceedings of VLDB Endowment*, vol. 12, no. 5, pp. 461–474, 2019.

[33] K. Aoyama, K. Saito, H. Sawada, and N. Ueda, "Fast approximate similarity search based on degree-reduced neighborhood graphs," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2011, pp. 1055–1063.

[34] B. Harwood and T. Drummond, "FANNG: Fast approximate nearest neighbour graphs," in *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 5713–5722.

[35] L. Amsaleg, O. Chelly, M. E. Houle, K.-i. Kawarabayashi, M. Radovanović, and W. Treeratanajaru, "Intrinsic dimensionality estimation within tight localities," in *Proceedings of the 2019 SIAM international conference on data mining*, 2019, pp. 181–189.

[36] H. Jgou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 1, pp. 117–128, 2011.

[37] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, pp. 91–110, 2004.

[38] A. B. Yandex and V. Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," in *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2055–2063.

[39] Q. Chen, H. Wang, M. Li, G. Ren, S. Li, J. Zhu, J. Li, C. Liu, L. Zhang, and J. Wang, "SPTAG: A library for fast approximate nearest neighbor search," 2018. [Online]. Available: https://github.com/Microsoft/SPTAG

[40] M. Douze, H. Jégou, H. Sandhawalia, L. Amsaleg, and C. Schmid, "Evaluation of GIST descriptors for web-scale image search," in *Proceedings of the 2009 ACM International Conference on Image and Video Retrieval*, 2009, pp. 1–8.

[41] H. Wang, W.-L. Zhao, X. Zeng, and J. Yang, "Fast k-NN graph construction by GPU based NN-Descent," in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, 2021, pp. 1929–1938.