Dynamic NN-Descent: An Efficient *k*-NN Graph Construction Method

Jie-Feng Wang, Wan-Lei Zhao*, Shihai Xiao, Jiajie Yao, Xuecang Zhang

***Abstract*—As a classic *k*-NN graph construction method, NN-Descent has been adopted in various applications for its simplicity, genericness, and efficiency. However, its memory consumption is high due to the employment of two extra supporting graph structures. In this paper, a novel *k*-NN graph construction method is proposed. Similar to NN-Descent, the *k*-NN graph is constructed by doing cross-matching continuously on the sampled neighbors on each neighborhood. Whereas different from NN-Descent, the cross-matching is undertaken directly on the *k*-NN graph under construction. It makes the extra graph structures adopted to support the cross-matching no longer necessary. Moreover, no synchronization between different threads is needed within one iteration. The high-quality graph is constructed at the high-speed efficiency and considerably better memory efficiency over NN-Descent on both the multi-thread CPU and the GPU.**

***Index Terms*—*k*-NN Graph, Dynamic NN-Descent, NN-Descent, GPU**

I. Introduction

Given a dataset $C = \{x|x \in R^d\}$, the *k*-nearest neighbor (*k*-NN) graph refers to the data structure that keeps the top-*k* nearest neighbors for each sample from the dataset. Typically, given *k*-NN graph $\mathbf{G}$ built from C and sample $x_i \in C$, $\mathbf{G}[i]$ returns the indice of top-*k* nearest neighbors of sample x_i. It is the fundamental data structure for a wide variety of applications, such as vector database, multimedia information retrieval, recommendation system, deep metric learning, and classification [1]–[5]. When it is built in a brute-force way, the time complexity is $O(d{\cdot}n^2)$, where d is the dimension and n is the size of the dataset. In practice, n could reach as much as billion level or even bigger. As a result, the computation cost is prohibitively high when the graph is built exhaustively.

Intuitively, given an efficient NN search method is available, *i.e.* HNSW [6], the *k*-NN graph could be built by launching a query for each sample on the indexing structure built upon the whole set. Whereas, the time cost for constructing a HNSW graph is already much higher than several state-of-the-art *k*-NN graph construction methods [7], [8]. Conversely, the construction of *k*-NN graph is the prerequisite of the NN search in recent NN search methods [1], [9], [10]. Recently, an interesting attempt has been made in [11] to construct the *k*-NN graph incrementally by querying against the *k*-NN graph under construction. Although encouraging performance is achieved, its efficiency is still inferior to the offline construction method [7], particularly in the multi-thread context.

In face of the data in large-scale and high dimension, recent studies no longer attempt to build *k*-NN graph with *100%* quality. Instead, many efforts [2], [7], [12], [13] have been made to explore efficient methods to build an approximate *k*-NN graph. In general, there are two major categories. In the first type of methods, the *k*-NN graph is constructed by a divide-and-conquer strategy [2], [8], [12], [13]. Given dataset C, it is divided into small subsets. The *k*-NN graph is constructed for each small subset. Thereafter, such small *k*-NN graphs are unioned into one in one way or another. The most attractive advantage of this type of method comes from its efficiency. However, the inductive bias about the data distribution or the distance metrics defined over the data makes such methods no longer a generic solution for the *k*-NN graph construction problem.

Alternatively, the NN-Descent algorithm proposed in [7] builds the *k*-NN graph by an iterative procedure. No dataset division strategy is adopted. The construction starts from a random *k*-NN graph, which is of very low quality. The graph is refined iteratively by updating the graph with the closer neighbors produced by cross-matching, which is called "*local-join*". The *local-join* is performed within each intermediate *k*-NN neighborhood. The refining process is motivated by the observation "neighbor's neighbor is neighbor", which is largely true due to the low intrinsic dimensionality for many real-world data. For its simplicity, effectiveness, and genericness to the various distance metrics, it remains the most popular *k*-NN graph construction method. In the recent *k*-NN construction competition *2023* sponsored by SIGMOD[1], the methods from the *top-1* and the *top-3* winners are essentially built upon NN-Descent.

Although NN-Descent is already very efficient, its extra memory consumption during the iteration is quite high, which makes it hard to deploy to a large-scale graph construction task or in a context where the memory resources are precious. According to the method, in addition to the *k*-NN graph under construction, another two graph structures are maintained to hold the neighbors and the reverse neighbors for each sample. They are used to support the *local-join*. The size of these two graphs is on the same scale as the *k*-NN graph. For this reason, the extra memory consumption becomes significant given the data size n increases to the million level. Moreover, its refining procedure cannot run at full speed under the multi-

Manuscript received April 19, 2024; revised July 16, 2024.

Jie-Feng Wang and Wan-Lei Zhao are with the Department of Computer Science and Technology, Xiamen University Xiamen 361005, China. Wan-Lei Zhao is the corresponding author, email: wlzhao@xmu.edu.cn.

Shihai Xiao, Jiajie Yao, and Xuecang Zhang are with Huawei Technologies Co., Ltd, Hangzhou, 310051, China.

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

thread context due to the required synchronization between different threads within one iteration.

In this paper, a dynamic NN-Descent algorithm for efficient *k*-NN graph construction is proposed. Similar to NN-Descent, the graph construction starts from a random *k*-NN graph and has been refined by the *local-join* on the *k*-NN neighborhood of each sample. However, different from NN-Descent, the extra two graph structures to support the *local-join* have been reduced to an incomplete graph. This graph structure only keeps part of the reverse neighbors of each sample. This leads to more than *80%* of extra memory save-up while without any impairment on the graph quality. The contributions of this paper are twofold.

- A dynamic NN-Descent algorithm for *k*-NN graph construction is presented. Compared to the classic NN-Descent algorithm, the way that we collect the neighbors for the *local-join* in the next round is essentially different. Instead of allocating the extra graph to hold the old and new neighbors, the neighbors are collected right before the *local-join* for each node. This leads to considerably lower memory consumption as well as the high-speed efficiency when it runs in multiple-thread.
- The GPU-based dynamic NN-Descent is also presented. It outperforms the most efficient GPU-based *k*-NN graph construction methods in the literature while requiring much less extra GPU memory. Typically, the high quality *k*-NN graph for a high dimensional *10*-million dataset can be constructed in *20* seconds.

The remainder of this paper is organized as follows. In Section II, the representative *k*-NN graph construction methods in the literature, particularly NN-Descent are reviewed. Section III presents the proposed Dynamic NN-Descent. In Section IV, a comprehensive study about the performance of Dynamic NN-Descent on both single-thread and multiple-thread contexts, as well as on GPU is conducted. Section V concludes the paper.

II. Related Work

The *k*-NN graph construction methods in the literature can be broadly categorized into two groups. The first type of method follows the divide-and-conquer strategy [2], [8], [12], [13]. In general, three steps are involved in this type of method. In the first step, the dataset is divided into small subsets. In the second step, the sub *k*-NN graphs are constructed for each subset. Finally, the sub-*k*-NN graphs are merged into one. Usually, a post-processing step is adopted to further refine the graph quality. For the second type of methods [7], [11], they essentially perform NN search on the *k*-NN graph under construction in an NN-Descent manner. In the remainder of this section, brief reviews about these two types *k*-NN graph construction are presented.

A. Methods based on Divide-and-Conquer

Different division strategies have been adopted by divide-and-conquer methods. In [8], [12], the dataset is recursively divided into small subsets with equal size. To facilitate the afterward sub-graph merging, overlapping between the bi-sected subsets is allowed in [12]. The complete graph is the union of sub-graphs constructed for the small subsets. An NN-Descent [7]-like procedure is adopted to refine the graph further. While in [2], [13], the dataset is divided into small subsets for several rounds. In one round, the overlapping between different subsets is not allowed. For each round, the sub-graph is built for each small subset. As a result, several "base" approximate neighborhoods [2] are produced for one sample. The merge of these neighborhoods for one sample results in a neighborhood for each sample of higher quality. Similar to [8], [12], an NN-Descent [7]-like procedure is also adopted to enhance the graph quality after the merging of sub-graphs. The pipelines in [13] and [2] are similar, the major difference lies in their division strategies. Method in [2] divides the dataset by a random partition tree, while method [13] partitions the dataset by random projection of the hash codes, which are derived from the raw vectors. These methods are very efficient according to the performance reported in [8]. Nevertheless, the assumption (or restriction) about the distance metrics is necessary to design an appropriate partition strategy. For this reason, one cannot expect that they are still generic solutions for the *k*-NN graph construction problem.

B. NN-Descent and its Variants

Different from the first type of method, no dataset division is adopted in the second type of method. The graph construction fully relies upon an NN-Descent procedure. Namely, neighbors find each other by comparing with samples living in the same intermediate neighborhood. In this way, samples are descent to their true neighbors gradually.

In the algorithm NN-Descent [7], *k*-NN graph construction starts from a randomly generated *k*-NN graph. Its iterative procedure refines this *k*-NN graph gradually. On one iteration, neighbors in a neighborhood $\mathbf{G}[i]$ are compared to each other, which is called the *local-join* step. This operation is motivated by the observation that "a neighbor's neighbor is likely to be a neighbor". New edges are produced after the *local-join*. The new edges are in turn used to update the graph. As only shorter edges than the current ones are inserted into the graph, it is guaranteed that the graph quality monotonically increases. Obviously, this basic procedure can be optimized further. For instance, there is no need to compare the old neighbors within the old neighborhood, which are referred to the neighbors already in the neighborhood before the previous iteration. Moreover, there is no need to perform pair-wise comparisons between all the neighbors. Another step called "sampling" is introduced in one iteration. The iterative procedure of NN-Descent can be summarized as follows.

- **Step-1. Sampling** For each node *i*, only a small portion of new/old close neighbors from *k*-NN graph $\mathbf{G}[i]$ and its reverse neighbors are collected. Since "a neighbor's neighbor is likely to be a neighbor", the afterward *local-join* between these sampled close neighbors will produce shorter edges, which are used to update graph $\mathbf{G}$.
- **Step-2. Local-join** With the sampled neighbors, *local-join* calculates the distance between old-new pairs and

new-new pairs. The produced new edges are inserted into the corresponding neighborhood in graph **G** when they are shorter than the kept edges.

It is possible to sample all the neighbors from $\mathbf{G}[i]$ to undertake the *local-join*. Due to the fixed small size of a neighborhood, the produced long edges by the *local-join* will be most likely squeezed out by the shorter edges which are produced by *local-join* between the close neighbors. For efficiency, only the close neighbors and the close reverse neighbors are sampled in the first step.

The convergence of NN-Descent has been proved in [14]. For the sake of efficiency, it can be terminated early when there are only very few updates on the graph [7]. In [11], each sample is treated as an incoming query to query against the *k*-NN graph under construction. This method is good for datasets of growing size. However, it cannot be as efficient as NN-Descent in particular under the multi-threads environment. NN-Descent is able to build a high-quality approximate *k*-NN graph in only a few iterations. Its implementation is available on Github[2]. Recently, the GPU-based NN-Descent is also proposed [15]. It is *10* times faster than the CPU version due to the high parallelization of GPU-based computing. Unfortunately, its memory consumption remains high.

In this paper, a novel way to undertake the *local-join* on the neighborhood of each sample is proposed. Instead of collecting the neighbors and reverse neighbors after one round of iteration, the neighbors are collected directly from the *k*-NN graph **G** that is under construction. No extra graph structure is needed to hold the collected neighbors of each sample. As a result, an incomplete list of reverse neighbors of each sample will be kept. For this reason, compared to the classic NN-Descent, *local-join* with the reverse neighbors in our algorithm is fulfilled in consecutive rounds. Such kind of strategy breaks the boundary between different rounds. It, therefore, makes the synchronization between different threads on one round iteration unnecessary, leading to a considerably higher speed in a multi-thread context. Moreover, such an advantage in speed efficiency over the classic NN-Descent becomes more apparent when more threads are used.

III. Dynamic NN-Descent

In NN-Descent, there are two separate steps in each iteration. Namely, the first step collects the new/old neighbors from the neighborhood of each node. The second step performs *local-join* on old-new and new-new pairs of each node. To support the iteration, it is necessary to maintain the extra graph structure to hold the old and new neighbors for each node. Let's call this graph structure as **H**. In the meantime, the reverse old and new neighbors of each node should be held on a similar structure as well. Let's call this graph structure as **R**. Figure 1(b) shows the major data structures that are allocated for the graph in Figure 1(a). As shown in the figure, another two graph structures **H** and **R** are allocated in addition to **G** that is under construction. Since the size of these two graph structures is on the same level as the graph **G**, the extra memory consumption grows linearly with the size of the data.

[2]https://github.com/aaalgo/kgraph

In this section, we explore a novel way to undertake the NN-Descent iteration to reduce its extra memory consumption.

A. k-NN Graph Construction

Intuitively, the graph structure **H** appears unnecessary since the list of old and new neighbors of each node is already maintained by **G**. However, **H** differs from **G** due to the iteration process of NN-Descent. Graph **G** is dynamically updated with the new edges produced by the *local-join*, while graph **H** is static, being renewed with the resulting **G** after every round of NN-Descent iteration. Both **H** and **G** maintain n neighborhoods, guiding the iteration on which group of samples the *local-join* should be performed. If we sample neighbors from **G**[i] instead of using **H**[i], the new closer neighbors will join in the *local-join* right after they have been inserted. This is actually beneficial to the graph quality, according to the principle "a neighbor's neighbor is likely to be a neighbor". It is therefore feasible to perform the *local-join* directly on the neighborhoods of graph **G**. Moreover, the neighborhood of **G**[i] is sampled right before a thread is going to perform *local-join* on **G**[i]. Only a cache is needed to keep the sampled neighbors for one thread. Therefore, the memory consumption induced by the introduction of **H** is saved.

Nevertheless, due to the absence of **H**, not all the reverse neighbors for one node are available. It is therefore impossible to sample from a complete reverse neighbor list for node *i*. Compared to the original NN-Descent, some of the reverse neighbors will not cross-match with neighbors being squeezed out in this round. Instead, they will join in the next round *local-join*. As a result, sufficient cross-matches between close neighbors are still performed in each neighborhood, which guarantees the high graph quality. In the following, we show how the procedure is undertaken when the sampling is performed directly on the dynamically changing graph **G**.

Given an arbitrary node *i*, its new/old neighbors are sampled from its current neighborhood $\mathbf{G}[i]$, namely $\mathcal{O} \leftarrow \mathbf{G}[i].old$, and $\mathcal{N} \leftarrow \mathbf{G}[i].new$. Given $a \in \mathcal{O}$, the entry for a in the reverse graph is updated by $\mathbf{R}[a].old \leftarrow \mathbf{R}[a].old \cup i$. Similarly, $\mathbf{R}[a].new \leftarrow \mathbf{R}[a].new \cup i$ when a is in $\mathcal{N}$. In addition, the corresponding reverse old and new neighbors are joined into $\mathcal{O}$ and $\mathcal{N}$ respectively (shown in Eqn. 1).

$$\begin{aligned} \mathcal{O} &\leftarrow \mathcal{O} \cup \mathbf{R}[i].old \\ \mathcal{N} &\leftarrow \mathcal{N} \cup \mathbf{R}[i].new \end{aligned} \tag{1}$$

The *local-join* is performed between samples within $\mathcal{N}$ and between $\mathcal{O}$ and $\mathcal{N}$. After the *local-join* is fulfilled on $\mathbf{G}[i]$, the reverse neighbor entry, namely $\mathbf{R}[i].old$ and $\mathbf{R}[i].new$ are set to empty. Compared to NN-Descent, it is clear to see both **G** and **R** are under update consistently.

Given i is the current node, only samples that have been visited before i are collected in its reverse neighborhood. These reverse neighbors will take part in the *local-join* of i in this round. After the *local-join* on $\mathbf{G}[i]$, $\mathbf{R}[i]$ is set to empty. As the iteration continues, other samples, *e.g.* b are joined into $\mathbf{R}[i]$ when *i* is collected during the sampling on b's neighborhood. The right side of Figure 1(c) illustrates the **R** in our method. Given *node-3* in the figure is the visiting node,

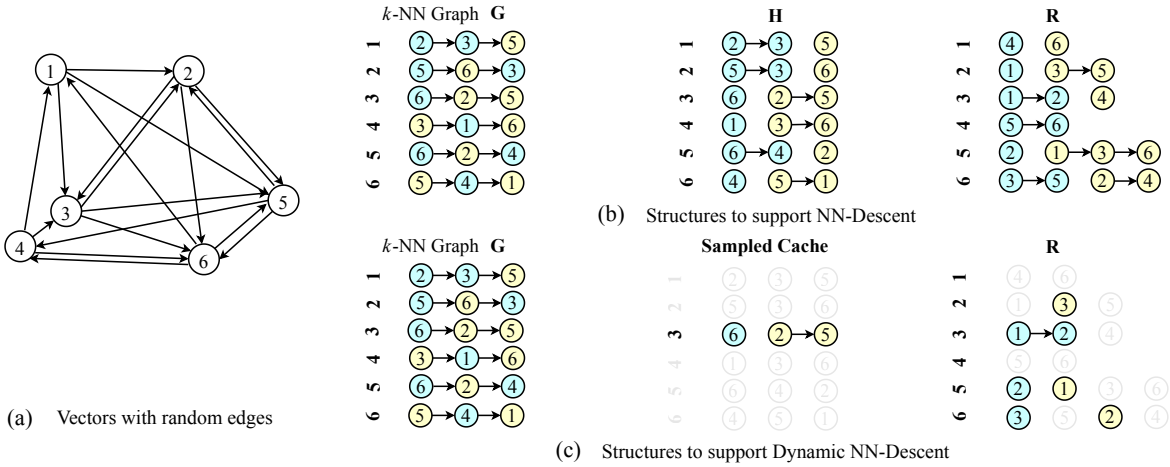

Fig. 1. The major data structures that are maintained for *k*-NN graph construction in NN-Descent and Dynamic NN-Descent. New and old neighbors are colored in yellow and blue, respectively. Figure (a) shows a *3*-NN graph with six nodes. Figure (b) shows three major structures in NN-Descent. Namely, they are *k*-NN graph **G** under construction, graph **H** that keeps the sampled old/new neighbors for each node, and the reverse graph **R** for **H**. Figure (c) shows the data structures adopted in Dynamic NN-Descent. In contrast to NN-Descent, graph **H** is reduced to a dynamic cache that only maintains the sampled neighbors for the node under visit. Moreover, only an incomplete reverse graph **R** is maintained.

caches $\mathcal{O}$ and $\mathcal{N}$ keep the sampled old and new neighbors from **G**[3] respectively. Compared to **R** in Figure 1(b), *node-4* is not in the list of **R**[3] because *4* is not yet visited at this moment. So it will not join in the *local-join* of **R**[3] of this round. *Node-4* will be added to **R**[3] as the reverse neighbor of *node-3* after **G**[4] has been visited. It will join in the *local-join* of **R**[3] in the next round. Compared to NN-Descent, only an incomplete reverse graph **R** is maintained. Moreover, graph **H** is replaced by two caches $\mathcal{O}$ and $\mathcal{N}$. As a result, significant memory consumption is saved. As verified in the later experiment, no significant graph quality degradation is observed when the *local-join* is performed on the samples from the dynamic neighborhood. Additionally, because the number of samples that take part in the *local-join* is similar to the original NN-Descent, our method shows similar efficiency as the original NN-Descent.

Since the *local-join* is performed on the dynamic neighborhoods of **G** and **R**, this revised algorithm is called *Dynamic NN-Descent*. The full algorithm that runs in multiple threads is presented in Algorithm 1. As seen from the algorithm, given a sample *i*, we sample its old neighbors and new neighbors from $\mathbf{G}[i]$ and push them to $\mathcal{O}$ and $\mathcal{N}$ respectively (*Lines 6–12*). Current i's reverse neighbors $\mathbf{R}[i].old$ and $\mathbf{R}[i].new$ are concatenated with corresponding new neighbors in $\mathcal{O}$ and $\mathcal{N}$ shown in *Lines 14–15*. Thereafter, $\mathbf{R}[i].old$ and $\mathbf{R}[i].new$ are set to be empty and ready to collect *i*'s reverse neighbors in the next iteration processes (*Lines 16–17*). The *local join* and the aftermath graph update are performed in *Lines 18–22*.

Discussion There are three major differences from the classic NN-Descent. Firstly, Dynamic NN-Descent performs sampling directly on the graph under construction, whose NN lists are updated dynamically. As a result, the sampling on such a dynamic neighborhood reflects the real neighborhood structure of one node at the moment of the *local-join*. In contrast, the neighborhood that is maintained in $\mathbf{H}[i]$ is static until the update in the next round of iteration. As the neighbors are collected on-the-fly, there is no need to maintain graph structure **H**. The memory consumption in keeping the sampled neighbors becomes minor. Secondly, Dynamic NN-Descent samples on an incomplete reverse neighbor graph $\mathbf{R}[i]$. As a result, the *local-join* with the reverse neighbors are fulfilled in consecutive rounds instead of one. Another benefit is that the expected memory consumption by **R** is around *50%* less than that of NN-Descent. Furthermore, there is no egg-chicken loop between *k*-NN graph **G** and the two supporting graphs **H** and **R**. In contrast to two nested loops in NN-Descent, there is only one loop in Algorithm 1. As a result, no synchronization between different *local-joins* within one round is needed, which turns out to be more friendly to the algorithm parallelization.

Space Complexity and Time Complexity The space complexity of the Dynamic NN-Descent and NN-Descent algorithms is $O((k+2\cdot smpN)\cdot n)$ and $O((k+4\cdot smpN)\cdot n)$ respectively, where k is the size of the neighborhood and *smpN* is the sampling size. The space complexity of the two algorithms is linear to the size of the dataset n. However, the Dynamic NN-Descent is lower than that of NN-Descent by a factor $2\cdot smpN$. Unlike most of the algorithms, the time complexity of NN-Descent and Dynamic NN-Descent varies from one dataset to another. It is closely related to the intrinsic dimension of the data. For this reason, an empirical time complexity analysis of the two algorithms is presented in Section IV-E.

B. GPU-based Dynamic NN-Descent

We also implemented the GPU-based Dynamic NN-Descent. Different from [15], Algorithm 1 is written into one kernel in CUDA C++. The GPU version largely follows the steps presented in Algorithm 1. The major difference lies in *Line 18* of Algorithm 1. The *local joins* between $\mathcal{O}$ and $\mathcal{N}$ and within $\mathcal{N}$ are realized by matrix-multiplication-like distance calculation, which is very efficient on GPU.

Algorithm 1 Dynamic NN-Descent

Require: dataset C, distance metric m, k-NN list size K, num. of sampled number $smpN$, num. of iterations $iterN$
Ensure: k-NN graph **G**

$iter \leftarrow 0$
Generate a random k-NN graph
for $iter < (|C| \cdot iterN)$ **do**
  $i \leftarrow iter \ \% \ |C|$
  /* Sampling */
  $\mathcal{O} \leftarrow$ sample $3 \cdot smpN$ items in $\mathbf{G}[i]$ with a false flag
  $\mathcal{N} \leftarrow$ sample $smpN$ items in $\mathbf{G}[i]$ with a true flag
  if $u \in \mathcal{O}$ && $\mathbf{R}[u].old.size() < smpN$ **then**
    $\mathbf{R}[u].old \leftarrow \mathbf{R}[u].old \cup i$
  end if
  if $u \in \mathcal{N}$ && $\mathbf{R}[u].new.size() < smpN$ **then**
    $\mathbf{R}[u].new \leftarrow \mathbf{R}[u].new \cup i$
  end if
  Mark sampled new items in $\mathbf{G}[i]$ as false
  $\mathcal{O} \leftarrow \mathcal{O} \cup \mathbf{R}[i].old$
  $\mathcal{N} \leftarrow \mathcal{N} \cup \mathbf{R}[i].new$
  $\mathbf{R}[i].old \leftarrow \phi$ /*clear the list*/
  $\mathbf{R}[i].new \leftarrow \phi$
  /*Local-join*/
  for $u_1, u_2 \in \mathcal{N}, u_1 < u_2$ or $u_1 \in \mathcal{N}, u_2 \in \mathcal{O}$ **do**
    dist $\leftarrow m(u_1, u_2)$
    Try to insert $\langle u_2, \text{dist}, \text{true} \rangle$ to $\mathbf{G}[u_1]$
    Try to insert $\langle u_1, \text{dist}, \text{true} \rangle$ to $\mathbf{G}[u_2]$
  end for
end for
return **G**

Due to the limited space of shared memory on the Streaming Multiprocessor (SM), *i.e.* only *100*KB for each SM on RTX4090, the matrix-multiplication-like distance calculation is organized in a tiled way. Only a small portion of vectors are loaded to the shared memory each time. The complete distances are accumulated after several phases of calculation. Figure 2 illustrates a mini example, where the sizes of $\mathcal{O}$ and $\mathcal{N}$ are *4*, the data dimension $d = 4$, and the tiled size is 2×2. In each tiled block, the calculation is split into *2* phases. Each phase fetches a 2×2 small block from A and B, then each thread calculates one dimension's square of their difference. The final distance matrix of a tiled block is accumulated through all phases.

IV. Experiments

A. Experiment Setup

In this section, the performance of Dynamic NN-Descent is studied on both CPU and GPU, in comparison mainly to NN-Descent and its variants. In the empirical study, we mainly consider the *k*-NN graph quality, construction time, and memory consumption. For CPU-based methods, our method is compared with classic NN-Descent and EFANNA [8]. For GPU-based methods, we compare with GNND[3] from [15], GGNN [16] as well as the package RAFT[4] recently released by NVIDIA.

[3]https://github.com/RayWang96/GPU_KNNG

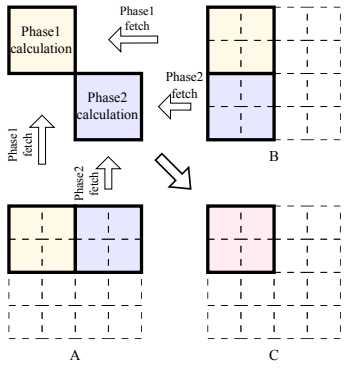

Fig. 2. The tiled distance calculation in the shared memory.

TABLE I
Summary on Datasets used for Evaluation

Name	d	LID	m(·,·)	Data Size
SIFT	128	15.6	L2	$10^4 \sim 10^8$
DEEP	96	15.9	L2	$10^4 \sim 10^8$
T2I	200	20.9/15.5	IP	10^6
SPACEV	100	23.2	L2	10^6
GIST	960	25.9	L2	10^6
TURING	100	28.5	L2	10^6

Six real-world datasets[5] are adopted in the evaluation. The data scale ranges from *1*-million level to *100*-million level. The brief information about these datasets is summarized in Table I. As shown in the table, our evaluation covers traditional image feature [17], [18], deep image features [19], text features [20], and cross-model features (namely T2I). They are all dense and in high dimensions. Different distance measures are adopted for different datasets. We use local intrinsic dimensionality [21] (LID shown in the *3rd* column) to measure the difficulty of a dataset. Generally, it is more challenging to build a high-quality *k*-NN graph on datasets with high intrinsic dimensionality. The top-*10* recall (*Recall@10*) on each dataset is studied under different metrics such as *L2* and *Inner Product*. Given a function $R(i, k)$ returns the number of true-positive neighbors at top-*k* NN list of sample i, the recall at top-k on the whole set is given as $Recall@k = \frac{\sum_{i=1}^{n} R(i,k)}{n \times k}$.

For CPU-based methods, all the experiments are carried out on a machine with two Intel(R) Xeon(R) Gold *6133* CPU (*2.50*GHz) and *256* GB of memory. All the codes of different methods considered in this study are compiled by GCC *11.4.0* with *-Ofast* compile option.

B. Memory Efficiency and k-NN Graph Quality

In this experiment, the memory efficiency and the *k*-NN graph quality of Dynamic NN-Descent are studied in comparison to NN-Descent. The *iterN* is fixed as *10*. In order to produce *k*-NN Graphs with different qualities, the parameter

[4]https://github.com/rapidsai/raft
[5]Datasets except GIST are available at https://big-ann-benchmarks.com

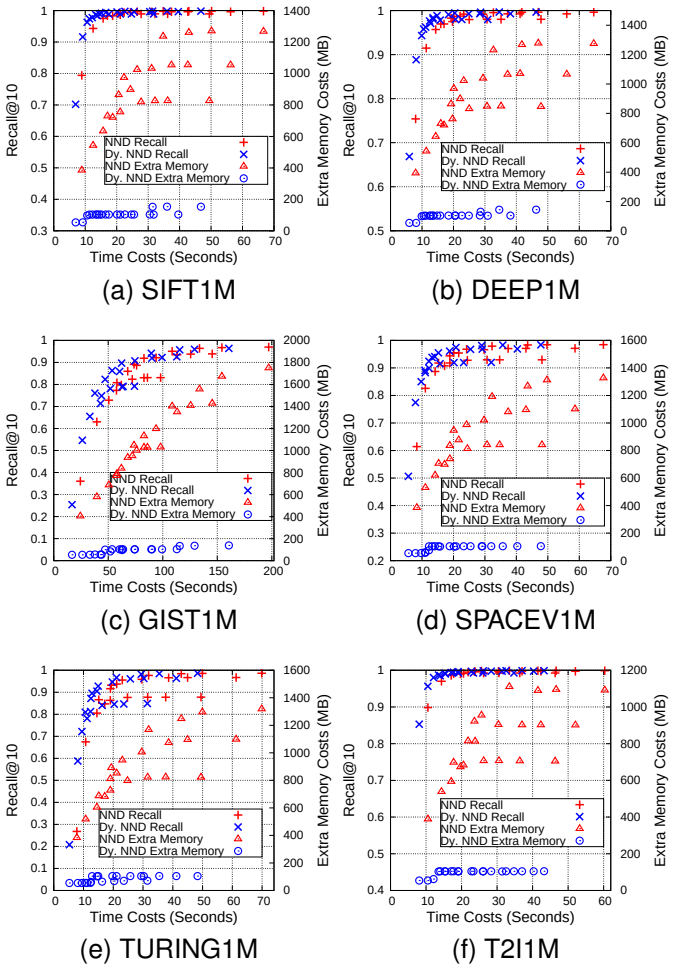

Fig. 3. *Recall@10* versus graph construction time, and *Recall@10* versus extra memory consumption by Dynamic NN-Descent and NN-Descent on six *1*-million datasets. #Threads=*80*.

K varies from *16* to *200*, and the $smpN$ varies from *16* to *32* for both Dynamic NN-Descent and NN-Descent. Typically, a larger K or $smpN$ builds a better *k*-NN graph, while taking more memory and construction time. Under the same parameter settings, Dynamic NN-Descent and NN-Descent build *k*-NN graphs of almost the same quality. The number of threads is fixed to *80* for both methods. For each dataset, the variations of *graph quality* (given as *Recall@10*) and the corresponding *Construction time* are plotted in Figure 3 for two methods. Accordingly, the variations in *Extra Memory Consumption*[6] are plotted in the figure as well.

As shown from Figure 3, the memory consumption of NN-Descent increases steadily when we build *k*-NN graph in higher quality. For a million-level dataset, the extra memory consumption is as much as more than *1.2*G Bytes. In contrast, the extra memory consumption from Dynamic NN-Descent remains very low across different configurations. In general, *6.7* times extra memories and $1.2 \sim 1.7$ times total memories are required for NN-Descent to build the *k*-NN graph at the similar quality and time costs as Dynamic NN-Descent. We also compared Dynamic NN-Descent with EFANNA [8].

[6]The memories to hold the raw data and the *k*-NN graph are not counted.

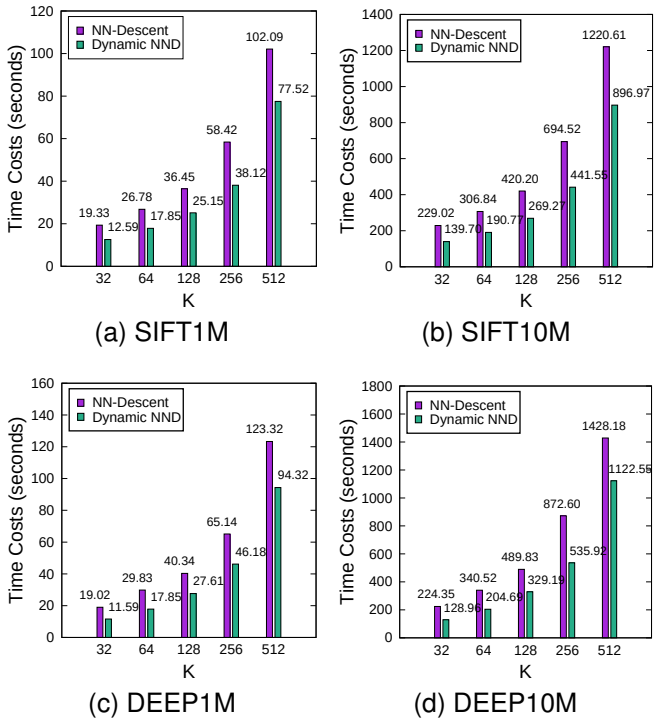

Fig. 4. Time efficiency under different Ks, *iterN=10*, #Thread=*80*. The *k*-NN graph quality (Recall@10) of SIFT1M, SIFT10M, DEEP1M, and DEEP10M are maintained at *0.99*, *0.98*, *0.99*, and *0.99* respectively for both algorithms.

The graph quality for our method and EFANNA is fixed to *Recall@10 = 0.99* for six million-scale datasets. Compared to our method, EFANNA takes $2.6 \sim 4.3$ times more time and $1.9 \sim 3.7$ times more memory.

C. Scalability Tests

In this section, the scalability of Dynamic NN-Descent is studied when we vary the size of *k*-NN graph, the number of threads, and the data size. The performance from NN-Descent is treated as the comparison baseline.

In the first experiment, *k*-NN graphs in different neighborhood sizes are built by Dynamic NN-Descent and NN-Descent. *K* varies from *32* to *512*. The other parameters are fixed to the same for them. Such that the graph quality from the two methods is similar. Figure 4 shows the time costs of two methods on SIFT1M, SIFT10M, DEEP1M, and DEEP10M.

As shown from the figure, Dynamic NN-Descent is considerably faster than NN-Descent across different *K*s. The efficiency is more significant when K is small. In practice, K is in the range $[40, 200]$ [1], [9], [10], [22]. Our method is *30%* faster than NN-Descent in the range.

In the second test, the performance trend of Dynamic NN-Descent is studied when it runs on a different number of threads. In the test, the number of threads varies from 1 to 64, while the other parameters are fixed. The construction time costs on four datasets are shown in Figure 5. As seen from the figure, Dynamic NN-Descent shows much higher efficiency over NN-Descent as more threads are used. This is largely attributed to its highly parallelizable procedure. Compared to NN-Descent, the synchronization on the inner-loop iteration

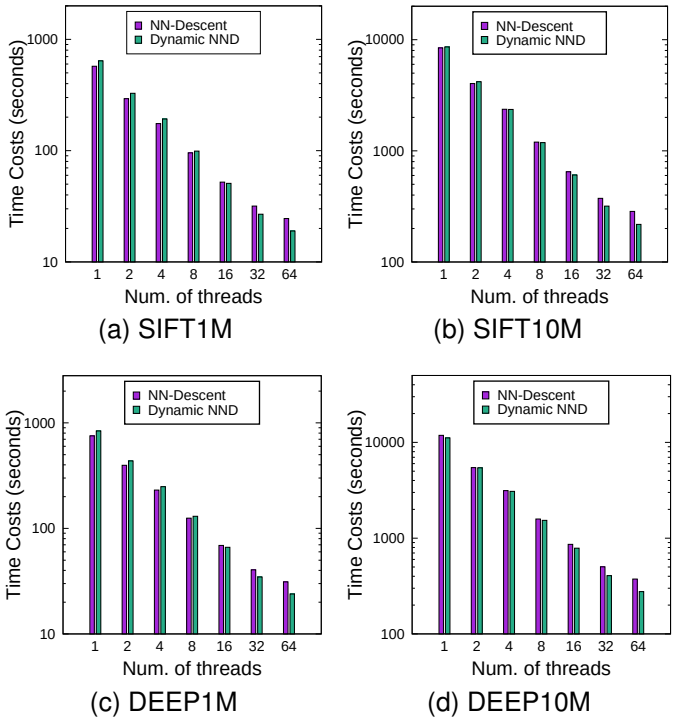

Fig. 5. Time efficiency with different num. of threads, *K=100*, *iterN=10*. The *k*-NN graph quality (*Recall@10*) for SIFT1M, SIFT10M, DEEP1M, and DEEP10M are maintained at *0.99*, *0.98*, *0.99*, and *0.98* respectively for both algorithms.

is no longer required. Since there is no boundary between consecutive iterations, the barrier to high parallelization is broken. In the last scalability test, the time efficiency of

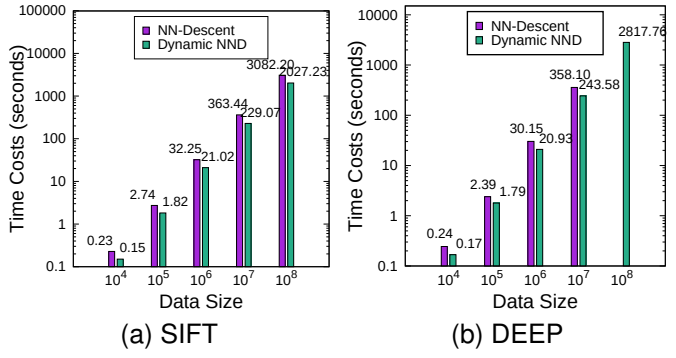

Fig. 6. Time efficiency under different data sizes, *#Thread=80*, *K=100*, *iterN=10*. For the SIFT and DEEP datasets, the *smpN* are set to *16* and *24* respectively. The *k*-NN graph quality (*Recall@10*) for SIFT and DEEP decreases from *0.99* to *0.90* and *0.96* respectively as the data size increases for both NN-Descent and Dynamic NN-Descent.

Dynamic NN-Descent is reported on SIFT and DEEP datasets, when their data size varies from 10^4 to 10^8. The parameters for Dynamic NN-Descent and NN-Descent are fixed to the same. The result on DEEP 100M is not reported for NN-Descent as its required memory is beyond the memory support of our machine. The time efficiency against the size of data is shown in Figure 6. Our method shows a higher percentage of efficiency over NN-Descent as the data size increases. This is again largely attributed to the high parallelization nature of our method.

TABLE II
PERFORMANCE EVALUATION ON GPU-BASED *k*-NN GRAPH CONSTRUCTION. *K=64*, *iterN=6* FOR DYNAMIC NN-DESCENT, GNND AND RAFT.

	TIME (Seconds)			
	Dynamic NND	GNND	RAFT	GGNN
SIFT1M	2.198	2.274	2.958	4.664
SIFT10M	24.463	25.474	29.838	68.337
DEEP1M	1.909	1.949	2.937	5.233
DEEP10M	20.892	21.833	30.923	64.631

	Recall@10			
	Dynamic NND	GNND	RAFT	GGNN
SIFT1M	0.994	0.994	0.990	0.993
SIFT10M	0.978	0.984	0.976	0.978
DEEP1M	0.990	0.989	0.985	0.990
DEEP10M	0.956	0.969	0.958	0.967

	GPU Memory (MB)			
	Dynamic NND	GNND	RAFT	GGNN
SIFT1M	1,534	1,786	492	730
SIFT10M	14,892	17,412	4,886	6,828
DEEP1M	1,412	1,664	370	606
DEEP10M	13,672	16,192	3,668	5,604

D. Performance on the GPU

In this experiment, GPU-based Dynamic NN-Descent is compared with GNND, RAFT, and GGNN. Among them, both GNND and RAFT are the GPU version of NN-Descent. RAFT differs from GNND in the arrangement of memory. The *k*-NN graph **G**, the supporting graphs **H** and **R** are mainly maintained in the main memory for RAFT. While GGNN builds the *k*-NN graph on GPU in a divide-and-conquer strategy. For GGNN, the parameters are loyal to its original implementation. The simulations are conducted on NVIDIA RTX4090.

The construction time costs, *k*-NN graph quality, and the GPU memory consumption (including memory consumption for raw vectors and the *k*-NN graph) are reported on Table II for four datasets. As shown in the table, Dynamic NN-Descent is the most efficient method. The construction efficiency of Dynamic NN-Descent, GNND, and RAFT are on the same level as they follow similar construction strategies. While the graph quality achieved by different methods is similar, RAFT shows much lower GPU memory consumption simply because the major parts of three graph structures are kept on the main memory. A similar scheme is also feasible for Dynamic NN-Descent if one would like to save up the GPU memory. GGNN builds the complete *k*-NN graph by merging sub-graphs, which are built in brute-force, via NN search one against the rest. It, therefore, takes less memory footprint than the other as no extra big data structure is involved. Nevertheless, GGNN hurts its construction efficiency due to the multiple random memory access during NN search, which is the processing bottleneck on the GPU memory with high IO latency.

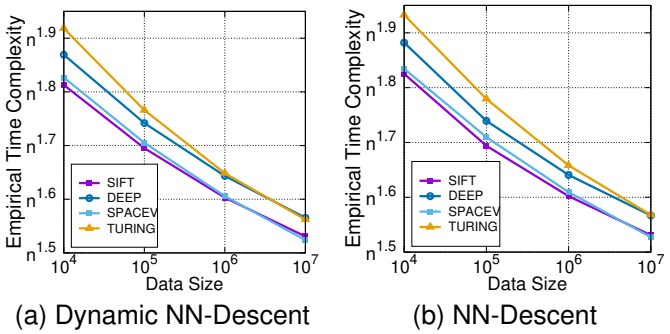

Fig. 7. The empirical time complexity of Dynamic NN-Descent and NN-Descent on SIFT, DEEP, SPACEV, and Turing. The size of data varies from 10^4 to 10^8. $K = 100, iterN = 10$. For the SIFT and DEEP datasets, the *smpN* values are set to *16* and *24* respectively.

E. Empirical Time Complexity

In this experiment, we study the trend of time complexity of Dynamic NN-Descent in comparison to that of NN-Descent. The time complexity is estimated by counting the intensive operations in both algorithms, namely the distance computation during the *local-join*. Four types of datasets, namely SIFT, DEEP, SPACEV, and TURING are considered in the experiment. The empirical time complexity is then inferred by examining the logarithmic relationship between the number of distance calculations and the dataset size n. The parameter K is fixed at *100* across all the experiments.

As seen from Figure 7, the empirical time complexity of the Dynamic NN-Descent and NN-Descent is $O(d{\cdot}n^p)$, where p is in the range $[1.5, 1.92]$. The two algorithms share a similar time complexity trend across different datasets, while the empirical time complexity of Dynamic NN-Descent is slightly lower than that of NN-Descent. Both of them show high time complexity on the datasets with high local intrinsic dimension (given in the *3rd* column of Table I). Moreover, as the dataset size increases, the empirical time complexity decreases steadily.

V. Conclusion

We have presented an efficient *k*-NN graph construction method, namely Dynamic NN-Descent. Compared to the classic NN-Descent, the *local-join* on the neighborhood of each sample is conducted on a dynamic neighborhood. Specifically, the old and new neighbors are sampled directly from the *k*-NN graph under construction, which is under update consistently. Moreover, the reverse neighbors are also sampled from the dynamic *k*-NN graph. These innovations on the NN-Descent lead to several advantages. First of all, the extra memory consumption has been reduced by *85%*. And the extra memory no longer grows proportionally as the graph quality improves. Moreover, the boundary between two consecutive iterations of NN-Descent about the *k*-NN graph update is broken. As a consequence, no synchronization on the threads within one iteration is needed. It allows the algorithm to run at its full speed under the multi-thread context. The extensive simulations on both the multi-thread CPU and GPU confirm the significance of our innovation.

References

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

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

[3] Y. Zhang, F. Sun, X. Yang, C. Xu, W. Ou, and Y. Zhang, "Graph-based regularization on embedding layers for recommendation," *ACM Trans. on Information Systems*, vol. 39, no. 1, 2020.

[4] Y. Zhu, M. Yang, C. Deng, and W. Liu, "Fewer is more: A deep graph metric learning perspective using fewer proxies," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, 2020, pp. 17 792–17 803.

[5] L. Song, P. Pan, K. Zhao, H. Yang, Y. Chen, Y. Zhang, Y. Xu, and R. Jin, "Large-scale training system for 100-million classification at alibaba," in *Proceedings of the 26th ACM SIGKDD International Conf. on Knowledge Discovery & Data Mining*, 2020, p. 2909–2930.

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

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

[8] C. Fu and D. Cai, "EFANNA: An extremely fast approximate nearest neighbor search algorithm based on kNN graph," *arXiv preprint arXiv:1609.07228*, 2016.

[9] 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 Trans. on Knowledge and Data Engineering*, vol. 32, no. 8, pp. 1475–1488, 2020.

[10] C. Fu, C. Wang, and D. Cai, "High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 44, no. 8, pp. 4139–4150, 2021.

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

[12] J. Chen, H.-r. 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. 9, 2009.

[13] Y.-M. Zhang, K. Huang, G. Geng, and C.-L. Liu, "Fast knn graph construction with locality sensitive hashing," in *Joint European Conf. on Machine Learning and Knowledge Discovery in Databases*. Springer, 2013, pp. 660–674.

[14] W.-L. Zhao, H. Wang, and C.-W. Ngo, "On the merge of k-NN graph," *IEEE Trans. on Big Data*, vol. 8, no. 6, pp. 1496 – 1510, 2021.

[15] H. Wang, W.-L. Zhao, X. Zeng, and J. Yang, "Fast k-NN Graph Construction by GPU based NN-Descent," in *Proceedings of the Conf. on Information and Knowledge Management*, 2021, pp. 1929 – 1938.

[16] F. Groh, L. Ruppert, P. Wieschollek, and H. P. A. Lensch, "Ggnn: Graph-based gpu nearest neighbor search," *IEEE Trans. on Big Data*, vol. 9, no. 1, pp. 267–279, 2023.

[17] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

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

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

[20] 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

[21] L. Amsaleg, O. Chelly, M. E. Houle, K.-I. Kawarabayashi, M. Radovanović, and W. Treeratanajaru, "Intrinsic dimensionality estimation within tight localities," in *Proceedings of SIAM International Conf. on Data Mining*, 2019, pp. 181–189.

[22] H. Ootomo, A. Naruse, C. Nolet, R. Wang, T. Feher, and Y. Wang, "CAGRA: Highly parallel graph construction and approximate nearest neighbor search for GPUs," *arXiv preprint arXiv:2308.15136*, 2023.

Jie-Feng Wang received his bachelor degree from Colledge of Ocean and Earth Sciences, Xiamen University in 2020. He is currently a gradudate student in Xiamen University, China. His research focuses on nearest neighbor search on the large-scale high-dimensional data.

Wan-Lei Zhao received his Ph.D degree from City University of Hong Kong in 2010. He received M.Eng. and B.Eng. degrees in Department of Computer Science and Engineering from Yunnan University in 2006 and 2002 respectively. He currently works with Xiamen University as an associate professor, China. Before joining Xiamen University, he was a Postdoctoral Scholar in INRIA, France. His research interests include multimedia information retrieval and video processing.

Shihai Xiao received the bachelor's degree and the master's degree in mechanics from Zhejiang University, Hangzhou, China, in 2005 and 2007, respectively. He is currently a Senior Researcher with Huawei Technologies, Shenzhen, China. His research topics include big data and machine learning DSA.

Jiajie Yao received master degree in mechanical engineering from Zhejiang University in 2019. He is currently a research engineer in Huawei Central Software Institute, and his work focuses on big data applications and information retrieval.

Xuecang Zhang received his Ph.D and bachelor degree from Zhejiang University in 2011 and 2006 respectively. Now he is a senior researcher and research manager in Huawei Technologies. His research topics include big data and machine learning algorithms, parallel and distributed computing system, scientific and engineering computing.