Knowledge and Information Systems (2023) 65:4185–4212
https://doi.org/10.1007/s10115-023-01899-2

REGULAR PAPER

Accelerating massive queries of approximate nearest neighbor search on high-dimensional data

Yingfan Liu[1] · Chaowei Song[1] · Hong Cheng[2] · Xiaofang Xia[1] · Jiangtao Cui[1]

Received: 5 January 2022 / Revised: 20 April 2023 / Accepted: 26 April 2023 /
Published online: 19 May 2023

Abstract

Approximate nearest neighbor (ANN) search on high-dimensional data is a fundamental operation in many applications. In this paper, we study massive queries of ANN (MQ-ANN) search, which deals with a large number of queries simultaneously. To improve the throughput, we combine the parallel capacity of multi-core CPUs and the filtering power of the state-of-the-art index methods, i.e., proximity graphs. However, there are no solutions that exploit proximity graphs to handle MQ-ANN in parallel, except the one called **query view**, which simply assigns each query to a hardware thread but suffers from numerous cache misses. As the first attempt, we design efficient methods for MQ-ANN with proximity graphs and propose a novel scheduling mechanism called **bridge view**, which shares the same data access across multiple queries in order to reduce cache misses. Moreover, we extend our method to deal with MQ-ANN on large-scale data sets (e.g. 10^8 points). Finally, we conduct extensive experiments on real data sets to demonstrate the advantages of our method. According to our experimental results, bridge view significantly outperforms query view in various settings. In particular, bridge view with 8 hardware threads even outperforms query view with 24 hardware threads.

Keywords Massive queries · Approximate nearest neighbor search · High-dimensional data · Proximity graphs · Parallel algorithms

✉ Yingfan Liu
liuyingfan@xidian.edu.cn

Chaowei Song
chwsong@foxmail.com

Hong Cheng
hcheng@se.cuhk.edu.hk

Xiaofang Xia
xiaofangxia89@gmail.com

Jiangtao Cui
cuijt@xidian.edu.cn

[1] School of Computer Science and Technology, Xidian University, Xi'an, Shaanxi, China

[2] The Chinese University of Hong Kong, Hong Kong, China

1 Introduction

Approximate Nearest Neighbor (ANN) search on high-dimensional data is a fundamental operation in many applications such as multimedia retrieval, data mining and machine learning. Given a data set $D \subset \mathbb{R}^d$ and a query vector $q \in \mathbb{R}^d$, ANN search returns a point in D, which is sufficiently close to q. *A typical solution to ANN search builds an index offline and then accelerates the search process for the online queries with the index*. During the past decades, many index methods have been developed, including locality sensitive hashing (LSH) based methods [1, 2], tree structures [3] and inverted index based methods [4, 5]. Recently, proximity graphs become more and more popular, due to their superior search performance [6–8]. Hence, a few proximity graphs were proposed, including DPG [8], HNSW [9] and NSG [10]. A proximity graph treats each point $u \in D$ as a graph vertex[1] and then builds edges between u and its close neighbors in D, which are selected according to specific neighbor selection strategy.

In this paper, we focus on an important variant of ANN search, i.e., massive queries of ANN (MQ-ANN). Unlike ANN that deals with a single query, MQ-ANN has to deal with a query set Q with massive queries (e.g. millions or even more) simultaneously and conduct ANN search for each query $q \in Q$. In this case, we usually use the **throughput**, i.e., the number of queries processed per second, as the measure of efficiency, while we care about the latency for each query in ANN search. MQ-ANN could be treated as a key component of several important operations. Large K-Means on large-scale data, e.g. clustering 1 billion high-dimensional vectors into a million clusters [11], involves MQ-ANN in each iteration in order to assign each point to the closest cluster centroid. KNN Join [12] with a large data set and a large query set conducts MQ-ANN after building an index to accelerate the search process. Moreover, K-nearest neighbor graph construction [13, 14] could be considered as a special case of KNN Join, where the query set is the data set itself.

In order to improve the throughput of MQ-ANN, a direct idea is to combine the parallel capacity of a modern server and the pruning power of an efficient index method. First, a modern server is equipped with multi-core CPUs, which have tens or even hundreds of hardware threads. Second, proximity graphs present strong pruning power, which is able to return highly accurate results ($recall > 0.9$) by only accessing a small part ($< 0.5\%$) of D [8]. However, there are no specific methods designed for MQ-ANN with proximity graphs in parallel, except the naive method, called **query view**, which assigns each query to a single hardware thread to exploit the parallelism. *To the best of our knowledge, this work is the first attempt to design efficient MQ-ANN approaches that combine multi-core CPUs and proximity graphs.*

First, let us brief ANN search with a proximity graph. Given a query q, the search process maintains a candidate pool *pool* with a limited size, which contains the closest neighbors of q found so far. *pool* is initialized with an entering point *ep* (randomly selected or specified in advance). Then, it iteratively extracts a vertex u, which is the closest but unexpanded vertex from the current *pool*, and then expands u, which treats all the neighbors of u in G as the candidates of q to refine *pool*. On average, each vertex in G has tens of neighbors [10]. Once the termination condition is triggered, the iterative process ends and the first neighbor of *pool* is returned as the ANN of q. *Hence, the essential operations of the search process are to expand a series of selected vertices in a greedy manner.*

However, due to the nature of the graph structure, ANN search with proximity graph leads to numerous cache misses. Note that the expansion on a vertex u accesses both the

[1] We interchangeably use point, vector and vertex in this work.

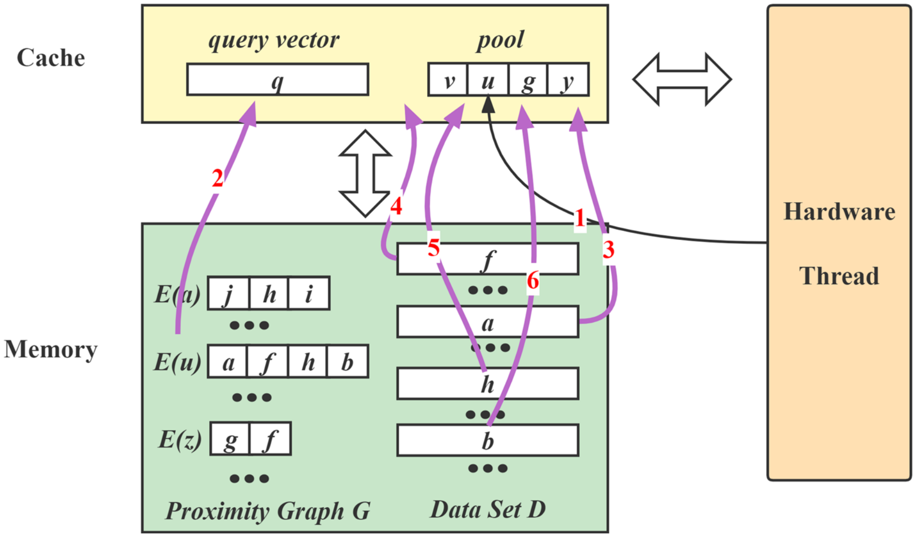


Fig. 1 Illustrating cache misses during the expansion on a vertex u. The thread needs to access the edge set $E(u)$ and the corresponding vectors in $E(u)$. However, those data are probably not in the cache, since D and G require far more space than the size of cache. Hence, the access to them causes cache misses

adjacency list $E(u)$ and the corresponding vectors in $E(u)$. Since those vectors in $E(u)$ could not be stored in a consecutive area and the cache is too small to hold the data set D, the accesses to them are probably in a random manner with many cache misses. We illustrate this phenomenon in Fig. 1, where a single hardware thread deals with ANN search for q and u is the vertex to be expanded. Before the expansion, q and *pool* have been loaded in the cache. To conduct the expansion, the thread first loads $E(u)$ in the cache with one cache miss and then fetches the corresponding vectors from the main memory with 4 cache misses. Further, consider that each query usually expands tens or hundreds of vertices, and thus the search process for a single query will cause a large number of cache misses.

Second, let us focus on efficient parallel solutions to MQ-ANN. The current solution (query view) simply assigns each query to a hardware thread, in order to exploit the parallel power, but suffers from numerous cache misses. This is mainly caused by the nature of ANN search with proximity graph. What's worse, each thread has less cache available on average, since the cache is shared by the threads. We illustrate the memory hierarchy of a multi-core CPU in Fig. 2, where the CPU has two cores or hardware threads. Both L1 cache (L1I and L1D) and L2 cache are only accessed by their corresponding hardware thread, while L3 cache is shared by all hardware threads in the CPU. In modern CPUs, there are usually tens or even more cores, which leads to less L3 cache available for each core.

In this work, we propose a new method, called **bridge view**, which is able to reduce cache misses by a large margin. The key idea is to share the access to the neighborhood of the same vertex among queries, like MS-BFS [15].[2] Unlike query view that deals with massive queries independently, bridge view handles those queries simultaneously by a smart scheduling mechanism. Our approach is based on the fact that a vertex u on G may be expanded by multiple queries. In particular, MQ-ANN has to deal with massive queries and thus each vertex may be expanded multiple times. Hence, we assign those expansions on the

[2] We discuss the difference between our solution and MS-BFS in Sect. 6.4.

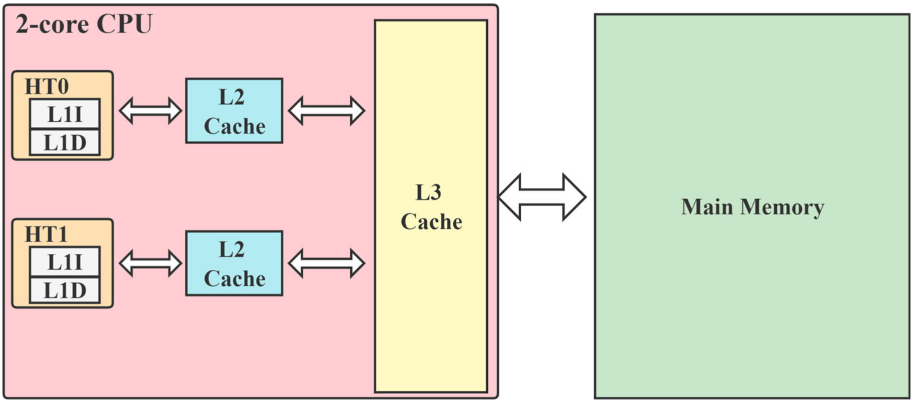


Fig. 2 Illustrating the memory hierarchy of a multi-core CPU

same vertex u to the same hardware thread, since they share the same access to $E(u)$ and the vectors in $E(u)$. Moreover, L2 cache in modern multi-core CPUs is large enough to hold both the edge set and the corresponding vectors for most vertices, as shown in Sect. 3.1. By this means, the queries after the first one will probably find the data required in the cache. As a result, bridge view obviously reduces cache misses.

Moreover, we extend bridge view to deal with MQ-ANN on large-scale D (e.g. 10^8 vectors). However, directly building proximity graph on those large data is of expensive cost. To reduce the building cost, the large-scale data is partitioned into a series of disjoint moderate-scale data and then build a proximity graph for each partition. Bridge view is able to enhance those partition based methods, since they all contain the same process, i.e., MQ-ANN on moderate-scale data.

To demonstrate the effectiveness of our approach, we conduct extensive experiments on both moderate-scale and large-scale data sets with three state-of-the-art proximity graphs, i.e., DPG [8], HNSW [9] and NSG [10]. According to our experimental results, our approach significantly outperforms query view in various settings. Moreover, bridge view with 8 hardware threads even outperforms query view with 24 hardware threads. Further, the performance of our method will be further enhanced with more threads and more queries, which is pretty suitable for MQ-ANN with multi-core CPUs. As to the experiments on large-scale data, bridge view obviously outperforms query view. Hence, our method also enhances MQ-ANN on large-scale data.

The contributions of this paper is listed as follows:

- As the first attempt to study MQ-ANN with proximity graphs, we propose a new method called bridge view, which is applicable to all proximity graphs.
- We extend bridge view to deal with large-scale data by dividing the large data into a series of moderate-scale data.
- We conduct extensive experiments to demonstrate the superiority of bridge view over the existing method, i.e., query view.

The rest of this paper is organized as follows. In Sect. 2, we show the preliminaries. In Sect. 3, we present our approach. In Sect. 4, we discuss the extensions of our method on large-scale data. In Sect. 5, we discuss our experiments. In Sect. 6, we list the related works. Finally, we conclude this paper in Sect. 7.

2 Preliminaries

In this section, we first formally define our problem and then present more details about proximity graphs, on which our solution is built. Finally, we discuss the motivation for this work.

2.1 Problem I

Definition 1 (*ANN*). Given a data set $D \subset \mathbb{R}^d$ with n points and a query $q \in \mathbb{R}^d$, ANN search returns a point $p \in D$, which is sufficiently close to q.

ANN search could easily be extended to k-ANN search, where k close points to q in D will be returned. For simplicity, we just use ANN search to represent both cases in this paper. Besides, we take the widely used Euclidean distance as the distance measure between two vectors.

Definition 2 (*MQ-ANN*). Given a data set $D \subset \mathbb{R}^d$ with n points and a query set $Q \subset \mathbb{R}^d$ with nq points, MQ-ANN returns the ANN for each $q \in Q$ in D.

We can see that ANN search only considers a single query, while MQ-ANN takes multiple queries into consideration simultaneously. Moreover, the number nq of queries in Q is large (e.g. millions or even more). In the paper, we design efficient solutions to MQ-ANN, which combine the parallel capacity of multi-core CPUs and the filtering power of proximity graphs.

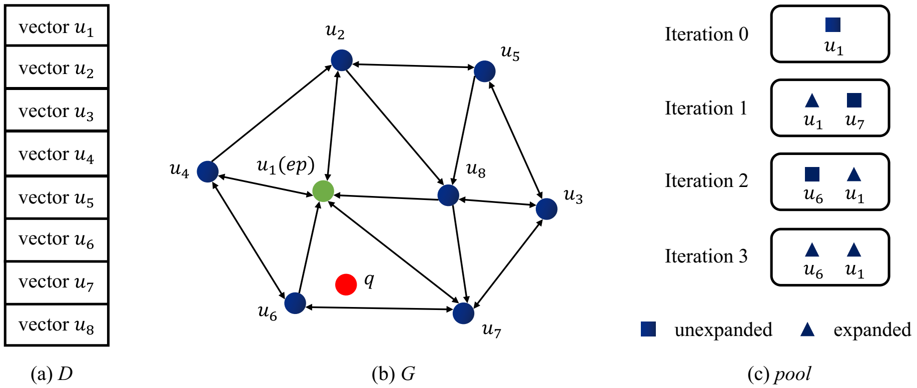
2.2 Proximity graphs

In this part, we discuss proximity graphs, which are the building bricks of our solution. Using proximity graphs to deal with ANN search contains two key steps, i.e., (1) building a proximity graph G offline and (2) accelerating ANN search with G online.

Offline building a proximity graph Given a high-dimensional data set D, a proximity graph G treats each point $u \in D$ as a graph vertex. During the construction of G, a set of close neighbors $E(u)$ is selected for each $u \in D$ according to the specific strategy of neighbor selection. For each neighbor $v \in E(u)$, a directed edge will be created from u to v. G is represented by adjacency lists, and thus $E(u)$ forms the adjacency list of u. We illustrate such a proximity graph with 8 vertices in Fig. 3.

Different proximity graphs share the same vertex set, but various edge set. For each $u \in D$, $E(u)$ is generated by distinct strategies of neighbor selection among proximity graphs. To be specific, KGraph [16] simply builds directed edges between u and its k nearest neighbors (KNN), while SW [17] creates undirected edges between u and its KNN. DPG, HNSW and NSG take the edge diversification strategy to select diverse neighbors from a set of closest neighbors for each vector u and then add both directed edges from u to those neighbors and the reverse edges from those neighbors to u. HNSW and NSG limit the size of $E(u)$ to a specified value, while DPG has no such limitation. Even though different edge set, each vertex usually has tens of out edges on average [10]. In this paper, we use three state-of-the-art proximity graphs, i.e., NSG [10], HNSW [9] and DPG [8], in our experiments as shown in Sect. 5. *We have no assumption on the properties of proximity graphs, such as connected or not.*

Online ANN search With an offline built proximity graph G for the data set D, we show how to find the ANN for a given query q in Algorithm 1 [10]. Except the basic parameters



(a) D (b) G (c) *pool*

Fig. 3 Illustrating ANN search with a proximity graph, where $k = 1$ and $efSearch = 2$. D is the high-dimensional data set. G is a proximity graph built on D, where each vector $u_i \in D$ is treated as a vertex in G. In (c), we show the iterative process of ANN search with a proximity graph

Algorithm 1 *Search_on_Graph*

```
Require: $G, D, q, k$ and $efSearch$;
Ensure: KNN of $q$ in $D$;
1: $L = max(k, efSearch)$ and $i = 0$;
2: initialize the candidate pool $pool$;
3: for each $u \in D$ do
4:     $visited[u] = false$;
5: end for
6: while $i < efSearch$ do
7:     $u = pool[i]$ and mark $u$ as expanded;
8:     /* Procedure: $expand(q, u, G)$*/
9:     for each $v \in E(u)$ do
10:        if $visited[v]$ then
11:            continue;
12:        end if
13:        /* Procedure: $update(pool, v)$*/
14:        $pool.add(v, dist(q, v))$;
15:        sort $pool$ in ascending order of $dist(q, \cdot)$;
16:        if $pool.size() > L$, then $pool.resize(L)$;
17:        $visited[v] = true$;
18:    end for
19:    $i$ = index of the first unexpanded vertex $u$ in $pool$;
20: end while
21: return the first $k$ points in $pool$;
```

including the proximity graph G, the data set D, the query q and the number k of returned neighbors, the search process requires another key parameter $efSearch$, which controls the number of expanded vertices during the search process.

Essentially, the search process takes an iterative and greedy strategy to expand a series of graph vertices. We use a data structure *pool* to store the L nearest neighbors of q found so far, where $L = \max(k, efSearch)$, as defined in Line 1. Moreover, the candidate pool *pool* is implemented by a sorted array, where its neighbors are sorted in ascending order according to their distance to q. In Algorithm 1, *pool* plays two roles, i.e., (1) storing the candidates of vertices to be expanded and (2) storing the best k-nearest neighbor found so far since $k \leq L$. We initialize *pool* in Line 2 according to the proximity graph. NSG and HNSW provide an entering point ep as the first element in $pool$, while DPG randomly selects a few vertices from D to fill $pool$. Afterwards, it iteratively expands a series of vertices as in Line 6–20. First, it extracts the currently closest but unexpanded vertex u from the $pool$ in Line 7, where i is its position in $pool$. The expansion on u, denoted as $expand(q, u, G)$ (Line 9–18), treats each neighbor $v \in E(u)$ as a candidate of KNN and then computes the distance $dist(q, v)$, followed by refining $pool$ with v, denoted as $update(pool, v)$ (Line 14–17). After expanding the current u, it determines the next vertex to be expanded by its index in $pool$ in Line 19. Once the termination condition, that the first $efSearch$ neighbors in $pool$ have been expanded, is triggered, the first k neighbors of $pool$ is returned as the KNN of q, as in Line 21.

Since a candidate v could be the neighbor of multiple expanded vertices, it will be visited more than once, which leads to repeated distance computations between q and v. To this end, the method employs a bitmap $visited$ that records whether or not a point has been visited. $visited$ is initialized as $false$ for each vector, as in Line 3–5. For each candidate v, the search process first checks $visited[v]$ and just skips the visited one to avoid repeated distance computation, as in Line 10–12. Otherwise, the search method conducts $update(pool, v)$ and marks v as visited, as in Line 13–17.

We illustrate the search process in Fig. 3, where we set $k = 1$ and $efSearch = 2$. Hence, $pool$ stores 2 nearest neighbors found so far. Let u_1 be the entering point ep. Given a query q, u_1 is first added to $pool$, and u_1 is first expanded, where the neighbors $\{u_2, u_4, u_7\}$ are treated as candidates to refine $pool$ and thus $pool = \{u_1 : expanded, u_7 : unexpanded\}$. Afterwards, u_7 is extracted from $pool$. The neighbors $\{u_1, u_3, u_6\}$ will be used to further refine $pool$. However, u_1 has been checked before and thus such a check with an expensive distance computation should be avoided by checking $visited[u_1]$ first. After expanding u_6, the first $efSearch = 2$ neighbors of $pool$ have been expanded, and thus the termination condition is triggered. Finally, u_6 is extracted from $pool = \{u_6, u_1\}$ and returned as the result of ANN search for q.

Notably, $efSearch$ is the key parameter to the search performance. A large $efSearch$ indicates more vertices expanded as well as more candidates checked to refine $pool$, which increases the probability of finding true KNN as well as the computational cost. To achieve highly accurate (e.g., > 95% recall) results, $efSearch$ is usually a quite small value from tens to hundreds [8, 10], due to the strong power of proximity graphs. As a result, the search process only accesses a small part of points in D for satisfied accuracy. We study the effect of $efSearch$ in Sect. 5.

2.3 Our motivation

Algorithm 2 Query View for MQ-ANN

```
Require: $G, D, Q, k$ and $efSearch$;
Ensure: KNNs of each $q \in Q$ in $D$;
$\mathcal{N} = \emptyset$;
#pragma omp parallel for
for each $q$ in $Q$ do
    $N_k(q) = Search\_on\_Graph(G, D, q, k, efSearch)$;
    $\mathcal{N} = \mathcal{N} \cup N_k(q)$;
end for
return $\mathcal{N}$;
```

As shown in Algorithm 2, the existing solution called query view simply assigns a query to each hardware thread so as to exploit the parallel computing power. This solution is easy to implement, but causes a huge number of cache misses, which decreases its efficiency. Those cache misses are brought out by the nature of the ANN search with a proximity graph, whose main operations are to conduct expansions on a set of vertices. To expand such a vertex u, the search process first visits its neighbor list $E(u)$ and the corresponding vectors in $E(u)$. Moreover, those accesses are probably taken in a random manner, since those vectors cannot be stored in a consecutive area and the cache. As a result, the access to each vector in $E(u)$ indicates a potential cache miss. This phenomenon has been illustrated in Fig. 1.

In this work, we aim at reducing those cache misses to improve the throughput for MQ-ANN. Our solution is based on the observation that a vertex $u \in D$ may be expanded by multiple queries. Hence, we can share the access to both $E(u)$ and the vectors in $E(u)$ across multiple queries. Especially, suppose that we have massive queries such as millions or even more. If we assign the expansions on the same vertex u to the same thread, it will obviously reduce the cache misses. After the expansion for the first query, the subsequent expansions on u will find $E(u)$ and the corresponding vectors in the cache.

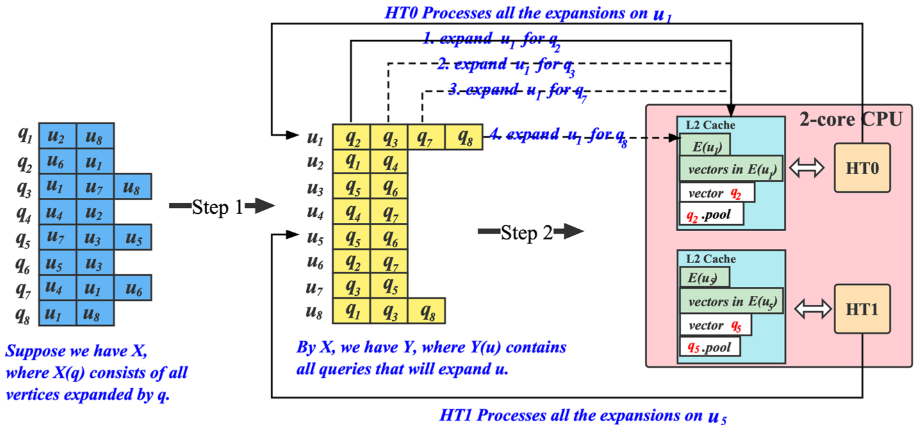
3 Our solution: bridge view

In this section, we discuss our solution. Let us consider a typical search path $q \rightarrow u \rightarrow v$. In this path, q is the query point, u is a vertex expanded by q and $v \in E(u)$. Here, we call u as the **bridge** vertex that connects the **query** q and its **candidate** v. We call the method described in Algorithm 2 as query view, since it centers at the query point. In contrast, our method centers at the bridge vertex and thus we call it bridge view. We first present a naive version of our solution to introduce the key idea of our solution and then show a mature one that works in practice. Moreover, we compare the pros and cons of two views.

3.1 A naive version

The key idea behind bridge view is to share the access to the neighborhood of the same vertex across multiple queries. To better explain the idea, it is assumed that we know the set $X(q)$ of all vertices expanded by each $q \in Q$ during the search process in advance. Accordingly, we derive $Y(u) = \{q|u \in X(q)\}$ for each $u \in D$. Bridge view simply assigns all the expansions on the same vertex u to the same hardware thread. Note that $Y(u)$ usually contains multiple queries in MQ-ANN. After the expansion for the first query, we will find the neighborhood including $E(u)$ and the corresponding vectors in the cache with a high probability for the remaining queries. Moreover, the neighborhood of most points in D could be held by the cache in modern CPUs, as shown in the following. Hence, more queries $Y(u)$ contains, more cache misses will be reduced. For simplicity, we denote $X = \{X(q)|q \in Q\}$ and $Y = \{Y(u)|u \in D\}$ in this work.

We illustrate the idea above in Fig. 4 with 8 queries and 8 points. Suppose that we have X in advance. Then, bridge view easily derives the set Y from X as illustrated as Step 1. With Y, bridge view simply assigns all the expansions on a single vertex to a hardware thread as illustrated as Step 2. `HT0` and `HT1` process all the expansions on u_1 and u_5 respectively. Note that there are 4 queries (q_2, q_3, q_7 and q_8) that will expand u_1. When dealing with the first expansion for q_2, five data structures need to be loaded into the cache, including $q_2.pool$, the query vector q_2, the edge set $E(u_1)$, the corresponding vectors in $E(u_1)$ and the set $Y(u_1)$.



Suppose we have X, where X(q) consists of all vertices expanded by q.

By X, we have Y, where Y(u) contains all queries that will expand u.

Fig. 4 Illustrating how bridge view reduces cache misses. The proximity graph G is the same one in Fig. 3. With given X, Y is easily derived in Step 1. Then, two threads HT0 and HT1 process all expansions on u_1 and u_5 respectively. After HT0 expands u_1 for q_2, subsequent expansions for q_3, q_7 and q_8 will find $E(u_1)$ and vectors in $E(u_1)$, which occupy a significant amount of space, still in cache. Thus, cache misses are obviously reduced

Notably, all queries in $Y(u_1)$ share the same access to $E(u_1)$ and the vectors in $E(u_1)$, which occupy much more space than a query vector and its candidate pool as shown in Table 1. After expanding u_1 for q_2, `HT0` finds $E(u_1)$ and the vectors in $E(u_1)$ still in the cache for the remaining queries, which thus reduces the cache misses in a significant scale. Moreover, those data structures are loaded into L3 cache, L2 cache and L1D cache successively. Since the size of L1D cache is usually pretty small (e.g. 32 KB), we focus on L2 cache and L3 cache when discussing cache misses.

Fortunately, we find that L2 cache in modern multi-core CPUs is large enough to hold the data required for a single expansion in most cases. Let us consider the CPU Intel Xeon E5-2697 V3 CPU, used in our experiments, as an example. It has 14 cores, L2 cache of size 14 × 256 KB and L3 cache of size 35 MB. Each piece of L2 cache with 256 KB is exclusively used by its corresponding core, while L3 cache is shared by all the 14 cores. Let us consider two well-known high-dimensional data sets, i.e., Sift and Gist, whose details could be found in Sect. 5.1. First, we study the distribution of out-degrees, i.e., $|E(u)|$, on three proximity graphs (NSG, HNSW and DPG). We show the results in Fig. 5. We can see that $Pr[|E(u)| \leq 50] \geq 0.8$ in all cases. Moreover, $Pr[|E(u)| \leq 50]$ is at least 0.95 on NSG and HNSW, because they set an upper bound on the out-degree for each vertex. However, there are some vertices with many (hundreds or even thousands) neighbors on DPG, which is caused by its strategy of neighbor selection [8].

To quantify the size of data structures required for a single expansion, we set $|E(u) = 50|$ and $L = 100$, which is the size of the candidate pool $q.pool$. We show the space required by the data structures for such a single expansion in Table 1. We can see that the L2 cache of size 256 KB is large enough to hold those data structures in our settings. As a result, once they have been loaded into L2 cache, they will stay in L2 cache until all the expansions on u are finished.

As to expanding large $E(u)$, whose data structures cannot be fitted in the L2 cache or even L3 cache, the idea discussed above will not work. To address this issue, we can evenly divide the large edge set $E(u)$ into several subsets and the expansion is divided into several sub-expansions accordingly, in order that L2 cache is able to cover the data structures for each sub-

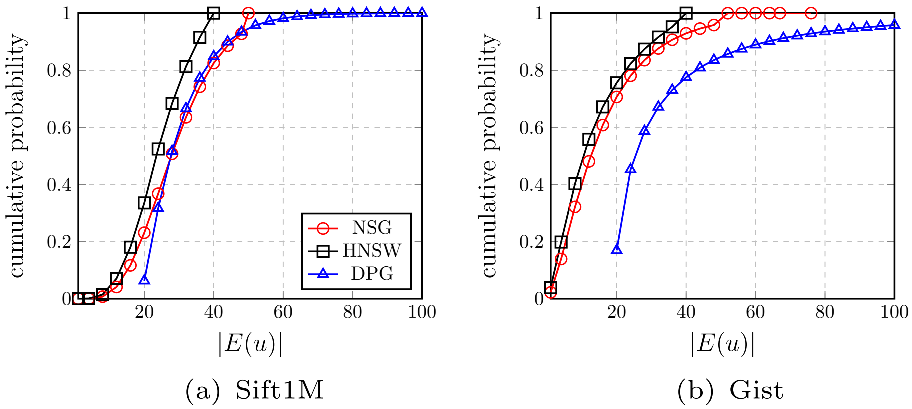


Fig. 5 Illustrating the distribution of out-degree $E(u)$ of proximity graphs

Table 1 Statistics of space required for an expansion. Each dimension of a vector is represented by a float value with 4 bytes. Each neighbor in $q.pool$ requires 9 bytes (4 bytes for the ID, 4 bytes for the distance value and one byte recording expanded/unexpanded)

Data	Sift	Gist
Size of $E(u)$	50 × 4 B	200 B
Dimensions	128	960
Size of vector q	0.5 KB	3.75 KB
Size of vectors in $E(u)$	25 KB	187.5 KB
Size of $q.pool$	900 B	900 B
Total size	26.57 KB	192.32 KB

expansion. During the expansion, bridge view conducts those sub-expansions successively for all queries in $Y(u)$. On the other hand, this issue could be relieved by the fact that L2 cache is increasingly larger and larger. For example, another CPU used in our experiments, Intel(R) Xeon(R) Gold 6238, has L2 cache of 1 MB for each core, which is 3 times larger than the other one.

To be formal, we show this naive idea in Algorithm 3. It contains two key steps. First, with the given X contains each $X(q)$, we derive $Y(u)$ for each $u \in D$, as in Line 3–7. Second, we simply assign all expansions on each vertex u to a hardware thread, in order to exploit the parallel computing power of the multi-core CPUs, as in Line 8–13.

3.2 A mature version

The naive version is built on the assumption that we have the knowledge of X in advance. Unfortunately, this is impossible in the real world, because $X(q)$ is iteratively extracted from the candidate pool one by one in a greedy manner, as shown in Algorithm 1. To deal with this issue, we propose an iterative method to generate an approximate $X(q)$. Note that each member of $X(q)$ is extracted as the currently first unexpanded vertex in the sorted $q.pool$. In our approximate method, we extract at most S unexpanded vertices from $q.pool$ in each iteration. Those vertices are denoted as $\bar{X}_i(q)$ for simplicity. The union set $\cup_i \bar{X}_i(q)$ in all iterations forms an approximation of $X(q)$, denoted as $\bar{X}(q)$. If $S = 1$, we have that $\bar{X}(q) = X(q)$. With the given $\bar{X}_i(q)$ for each $q \in Q$, $\forall u \in D$ we can accordingly derive the set $\bar{Y}_i(u)$ of queries that will expand u in the i-th iteration. Larger S is, larger the size of

Algorithm 3 The Naive Version of Bridge View

```
Require: $G$, $D$, $Q$, $k$ and $X$
Ensure: KNNs of each $q \in Q$
$Y(u) = \emptyset$ for each $u \in D$;
$\mathcal{N} = \emptyset$;
for each $q$ in $Q$ do
    for each $u$ in $X(q)$ do
        $Y(u).add(q)$;
    end for
end for
#pragma omp parallel for
for each $u$ in $D$ do
    for each $q$ in $Y(u)$ do
        conduct $expand(q, u, G)$ to refine $N_k(q)$;
    end for
end for
for each $q$ in $Q$ do
    $\mathcal{N} = \mathcal{N} \cup N_k(q)$;
end for
return $\mathcal{N}$;
```

$\bar{Y}_i(u)$ for each u will be on average, which reduces more cache misses. On the other hand, a large S decreases the quality of $\bar{Y}_i(u)$ for each u. Hence, S should be carefully selected. However, the performance of bridge view is slightly sensitive to S and a small value (e.g. 10) will be enough for a good performance, as discussed in Sect. 5.2.3.

We show the mature version of bridge view in Algorithm 4. Like ANN search in Algorithm 2, bridge view requires the inputs including the proximity graph G, the data set D, the query set Q, the number k of returned neighbors and the search parameter $efSearch$. In addition, it requires another parameter S, which determines the maximum number of unexpanded vertices from $q.pool$ for each $q \in Q$.

In general, bridge view contains three parts, i.e., **initialization** (Line 1–5), **main loop** (Line 6–9) and **result extraction** (Line 10–14). In **initialization**, bridge view initializes the key parameters and the key data structure, i.e., $q.pool$ for each $q \in Q$, which has the same meaning as $pool$ used in Algorithm 1. We set the result set $\mathcal{N}$ empty in Line 1. Like Algorithm 1, we determine the capacity of each candidate pool $q.pool$ as $L = max(k, efSearch)$. In Line 3–5, bridge view initializes each $q.pool$ and guarantees that it contains at least S unexpanded neighbors after the initialization. In both HNSW and NSG, which contain an entering point ep, we add ep to $q.pool$ and conduct the iterative process of expanding close vertices until there are at least S unexpanded neighbors in $q.pool$. Since S is set as a small value (e.g. 10), the initialization could be finished after expanding ep that contains usually tens of neighbors. As to DPG, we randomly select S points like the ANN search algorithm [8].

After initialization, bridge view conducts the **main loop**, as shown in Line 6–9. We can see that each iteration has three procedures, i.e., `ASSIGN` (Line 17–30), `CONNECT` (Line 32–41) and `IsTerminated` (Line 6). In `ASSIGN`, bridge view computes the set $\bar{Y}_i(u)$ for each $u \in D$ in this iteration. To achieve this goal, it retrieves $q.pool$ for each q to collect at most S unexpanded vertices and then adds those vertices to $\bar{Y}_i(u)$ accordingly, as in Line 22–28. In `CONNECT`, bridge view assigns all the expansions on the same vertex to a single hardware thread, as shown in Line 32–41. Each expansion connects each query $q \in \bar{Y}_i(u)$ and its candidates in $E(u)$, to refine $q.pool$ as in Line 37. Note that both `ASSIGN` and `CONNECT`

Algorithm 4 The Mature Version of Bridge View

```
Require: $G$, $D$, $Q$, $k$, $efSearch$ and $S$
Ensure: KNNs for each $q \in Q$
1: $\mathcal{N} = \emptyset$;
2: /* Initialization of Bridge View */
3: for each $q$ in $Q$ do
4:     initialize $q.pool$;
5: end for
6: while !IsTerminated() do
7:     ASSIGN();
8:     CONNECT();
9: end while
10: #pragma omp parallel for
11: for each $q$ in $Q$ do
12:     extract the first $k$ points in $q.pool$ to form $N_k(q)$;
13:     $\mathcal{N} = \mathcal{N} \cup N_k(q)$;
14: end for
15: return $\mathcal{N}$;
16:                                            ▷ the procedure of ASSIGN
17: proecedure ASSIGN()
18: $\bar{Y}_i(u) = \emptyset$ for each $u \in D$;
19: #pragma omp parallel for
20: for each $q$ in $Q$ do
21:     $num = 0$;
22:     for $i : 0 \rightarrow efSearch - 1$ do
23:         $u = q.pool[i]$;
24:         if ($u$ is unexpanded) & ($num < S$) then
25:             $\bar{Y}_i(u).add(q)$ and mark $u$ as expanded;
26:             $num$++;
27:         end if
28:     end for
29: end for
30: return;
31:                                            ▷ the procedure of CONNECT
32: proecedure CONNECT()
33: #pragma omp parallel for
34: for each $u$ in $D$ do
35:     for each $q$ in $\bar{Y}_i(u)$ do
36:         for each $v$ in $E(u)$ do
37:             $update(q.pool, v)$;
38:         end for
39:     end for
40: end for
41: return;
```

could be done in parallel, as shown in Line 19 and Line 33 respectively. However, ASSIGN centers at each query, while CONNECT centers at each graph vertex.

Besides, the **main loop** needs a termination condition, which is implemented by the procedure IsTerminated. A simple and naive one could be that none expanded vertex has been found in the ASSIGN procedure. This condition means that the first $efSearch$ vertices in each $q.pool$ has been expanded, which is just the termination condition in Algorithm 1. However, this condition will waste much cost in the later iterations, where only a small number of unexpanded vertices are found with the same cost in ASSIGN. This condition decreases the efficiency of bridge view. To relieve this issue, we set an early termination condition by monitoring the number $\#total\_num\_unexp$ of unexpanded vertices extracted in each iteration. Due to the parameter S, bridge extracts at most S unexpanded vertices in each iteration and thus there are at most $nq \times S$ unexpanded vertices found in each iteration. We can set the early termination condition as $\#total\_num\_unexp \leq \Delta \times nq \times S$, where $\Delta \in [0, 1)$ is a threshold specified in advance. In this way, bridge view just terminates when there are not enough unexpanded vertices found in the last iteration.

As to **result extraction**, we just extract k close neighbors from $q.pool$ for each query q, as shown in Line 14 to 17. Bridge view returns those neighbors as the results of MQ-ANN and ends the whole process.

3.3 Comparing two views

In this part, we discuss the pros and cons of two views. Generally speaking, bridge view is able to reduce the cache misses in a significant scale, but at the expense of repeated distance computations and more memory. By contrast, query view is memory-efficient and has no repeated distance computations, but suffers from numerous cache misses as well as low throughput.

For a query q, some candidates may appear in the neighborhood of multiple expanded vertices, which will lead to multiple repeated distance computations if ignoring this issue. As shown in Fig. 3, $u_4 \in E(u_1)$ and $u_4 \in E(u_6)$. The distance between q and u_4 has been computed after expanding u_1. If we ignore this fact, the same distance will be computed when expanding u_6 subsequently. Query view addresses this issue by the bitmap $visited$ for q as shown in Algorithm 1. In fact, $visited$ is a bitmap of n bits and each bit records whether or not the corresponding point has been accessed. In the search process for q, query view allocates the space for $visited$ in the beginning and frees it at the end. Hence, it only requires additional $O(n \times T)$ memory space, where T is the number of hardware threads. Since T is usually tens in a modern server, $O(n \times T)$ additional space could be tolerated, especially considering the data D of space $O(n \times d)$, where d usually ranges between tens and thousands.

Unfortunately, this simple strategy is not practically useful for bridge view. Since bridge view deals with all the queries in the meantime, it cannot afford the bitmaps for all queries, which requires $O(n \times nq)$ space. Since nq could be far larger than hundreds, the required space will be huge. However, each distance computation in bridge view costs significantly less than query view on average, since our method accesses the vectors faster due to far fewer cache misses as demonstrated in Sect. 5.2.2.

Moreover, bridge view requires more memory space than query view, even without those bitmaps. This is mainly caused by the fact that bridge view deals with all the queries simultaneously. As a result, we have to store $q.pool$ for each query q and $\bar{Y}_i(u)$ for each bridge vertex u. The total space for all candidate pools is $O(nq \times L)$, while the total space for $\bar{Y}_i(\cdot)$ is $O(n \times S)$. L is affected by $efSearch$ that is usually large enough (tens or hundreds) to achieve high accuracy. In contrast, query view only needs to keep at most T candidate pools in the memory, since there are at most T queries simultaneously.

In addition, the number nq of queries in Q has no effect on the performance of query view, since all the queries are processed independently. But, bridge view will benefit from a larger nq, as demonstrated in Sect. 5.2.3. This is because a larger nq will increase the average size of $\bar{Y}_i(u)$ for each bridge vertex u and thus lead to more cache misses reduced for expansions on u.

Algorithm 5 Single Query of Partition Based Methods

```
Require: ${G_1, ..., G_{n_{par}}}$, ${D_1, D_2, ..., D_{n_{par}}}$, $q$, $k$ and $efSearch$
Ensure: KNNs of $q$
1: #pragma omp parallel for
2: for $1 \leq i \leq n_{par}$ do
3:     $N_k^i(q) = Search\_on\_Graph(G_i, D_i, q, k, efSearch)$;
4: end for
5: /* the merge operation */
6: find the best $k$ neighbors in $\cup_{i=1}^{n_{par}} N_k^i(q)$ to form $N_k(q)$;
7: return $N_k(q)$;
```

4 Discussions on large-scale data

In this section, we discuss massive queries on large-scale D (e.g. 10^8 vectors or more). The key challenge on large-scale data is the construction cost of proximity graphs, which is superlinear with the number n of points in D. When n is large enough, the construction cost will be pretty large. To relieve this issue, we follow the idea in NSG [10], called partition based methods in this work, which simply divide the large data into equal-size partitions and build a proximity graph for each partition. By this way, they are able to reduce the construction cost. In this following, we show more details of partition based methods and discuss how to apply our method bridge view to enhance the search performance for partition based methods.

To be specific, partition based methods evenly divide the large-scale D into n_{par} partitions, i.e., $\{D_1, D_2, ..., D_{n_{par}}\}$ and then build a proximity graph for each partition [10], i.e., $\{G_1, G_2, ..., G_{n_{par}}\}$ accordingly. As a classical instance in [10], when dealing with large-scale data, the data is first divided into 16 partitions (w.r.t. 16 cores) and then 16 NSG graphs are built. Note that the construction cost of a proximity graph is $O(n^t \times d)$ [18], where $t \in (1, 2)$ depends on the construction method and the data distribution, since the graph construction needs to find close neighbors for each point in D. However, partition based methods build n_{par} proximity graphs on the same number of equal-size data partitions. Thus, their construction cost is reduced to $O(n_{par}^{1-t} \times n^t \times d)$.

Moreover, partition based methods exploit the parallel power of multi-core CPUs to accelerate the search process. During the search for a single query, they conduct ANN search on each partition in parallel and finally merge those results from n_{par} partitions. As shown in Algorithm 5, given a query q, it is assigned to n_{par} cores simultaneously and each core conducts ANN search on a single partition, as shown in Line 1–4. Finally, a simple merging operation returns the best result, as shown in Line 6. Since n_{par} cores are exploited to deal with the same query, partition based methods improve the data-level parallelism.

We can see that current partition based methods (e.g. [10]) actually employ query view when dealing with massive queries. They assign each hardware thread the same query simultaneously but conduct ANN search on different data partitions in parallel. Hence, we can improve those methods by replacing query view with bridge view. By this means, we can exploit the advantages of bridge view to enhance the performance of partition based methods. We show this idea in Algorithm 6. Like bridge view, it requires the parameters as inputs including Q, k and $efSearch$. However, it has a set of data sets $\{D_1, D_2, ..., D_{n_{par}}\}$and their corresponding proximity graphs $\{G_1, G_2, ..., G_{n_{par}}\}$ as its inputs, due to the partitioning strategy. In Line 1–4, the KNN set $N_k(q)$ for each query is initialized as $\emptyset$. In Line 5–12, it iteratively conducts the bridge view for each partition and then merges the results found before this iteration and the results found in the current partition. In each iteration, it takes

Algorithm 6 Partition Based Methods with Bridge View

```
Require: $\{G_1, G_2, \ldots, G_{n_{par}}\}$, $\{D_1, D_2, ..., D_{n_{par}}\}$, $Q$, $k$ and $efSearch$
Ensure: KNNs of each $q \in Q$
1:  $\mathcal{N} = \emptyset$;
2:  for each $q \in Q$ do
3:      $N_k(q) = \emptyset$;
4:  end for
5:  for $1 \le i \le n_{par}$ do
6:      conduct bridge view on $G_i$, $D_i$ and $Q$, with $\mathcal{N}'$ as returned results;
7:      /* the merge operation */
8:      for each $q \in Q$ do
9:          obtain $N_k'(q)$ from $\mathcal{N}'$;
10:         find the best $k$ neighbors of $N_k(q) \cup N_k'(q)$ to form $N_k(q)$;
11:     end for
12: end for
13: for each $q \in Q$ do
14:     $\mathcal{N} = \mathcal{N} \cup N_k(q)$;
15: end for
16: return $\mathcal{N}$;
```

D_i, G_i, Q, k and $efSearch$ as inputs of bridge view in Algorithm 4, and return the result $\mathcal{N}'$, as shown in Line 6. In Line 8–11, it conducts the merge operation, which selects the best k neighbors from $N_k(q) \cup N_k'(q)$ to form a new version of $N_k(q)$ for each $q \in Q$. Here, $N_k(q)$ represents the best neighbors found before this iteration, while $N_k'(q)$ those found in this iteration. Finally, we collect each KNN set $N_k(q)$ to generate the final set $\mathcal{N}$, as shown in Line 13–15.

5 Experiments

In this section, we present our experimental results, in order to demonstrate the advantages of bridge view over query view in various settings. First of all, we list the experimental settings. Then, we present the main experimental results, explore the underlying reasons and investigate the effects of key parameters on bridge view. In addition, we show the experimental results on large-scale data with partition based methods.

5.1 Experimental settings

The statistics of the data used in this paper are listed in Table 2. Here, each data contains a pair of data set and query set.

- **Moderate-scale data:** there are totally four pairs of moderate-scale data. The points and queries of Sift1M are randomly sampled from the learn set of Sift1B.[3] Both sets of Deep1M are randomly sampled from the learn set of Deep1B.[4] Both sets of Tiny1M are randomly sampled from Tiny80M.[5] Gist is from the data with the same name Gist [3], but is equally divided into data set and query set.
- **Large-scale data:** two pairs of large-scale data are used. Sift100M is the learn set of Sift1B [3], divided into two parts in a random manner, i.e., 90 million data points and 10

[3] http://corpus-texmex.irisa.fr/.

[4] https://yadi.sk/d/11eDCm7Dsn9GA.

[5] http://horatio.cs.nyu.edu/mit/tiny/data/.

Table 2 Data statistics

Data	n	d	nq
Sift1M	1,000,000	128	1,000,000
Deep1M	1,000,000	96	1,000,000
Tiny1M	1,000,000	384	1,000,000
Gist	500,000	960	500,000
Sift100M	90,000,000	128	10,000,000
Deep100M	90,000,000	96	10,000,000

million query points. Deep100M is a random sample of 100 million points from the learn set of Deep1B [4]. Similar to Sift100M, Deep100M is randomly divided into two parts, i.e., 90 million data points and 10 million query points.

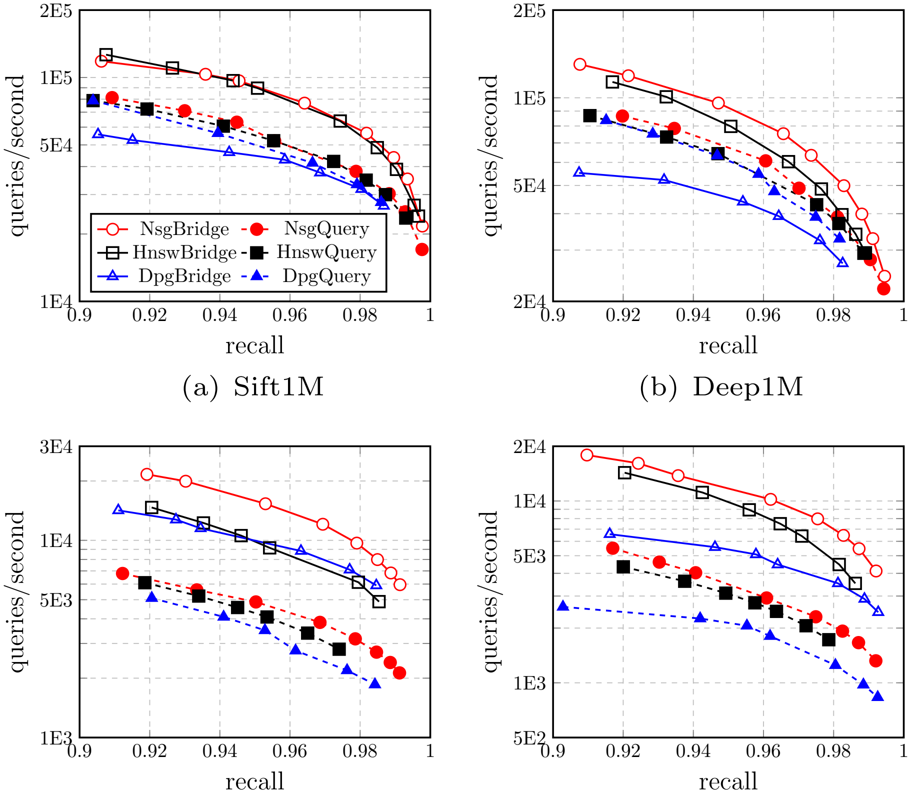
Computing Environments: our experiments are conducted on two workstations. The first workstation is equipped with two Intel(R) Xeon(R) E5-2697 v3 CPUs, while the second one with two Intel(R) Xeon(R) Gold 6238 CPUs. An Intel(R) Xeon(R) E5-2697 v3 CPU has 14 cores, L1D cache of 14×32 KB, L2 cache of 14×256 KB and L3 cache of 35 MB. An Intel(R) Xeon(R) Gold 6238 CPU has 22 cores, 22×32 KB, L2 cache of 22×1 MB and L3 cache of 30.25 MB. We use the first workstation for our experiments by default, unless specified. The codes are implemented by C++ and compiled by `g++4.8`. We use the SIMD instructions to accelerate distance computations. By default, we set the number T of threads as 16 and the number k of returned neighbors as 20, unless specified.

Performance Indicators: for each method of MQ-ANN, we use the number of queries handled per second to evaluate its efficiency, i.e., $queries/second$. We use $recall$ to estimate the accuracy of returned results. Let $knn(q)$ be the returned results for the query q and $knn^*(q)$ be the exact KNNs of q. The recall of q is defined as $recall(q) = |knn(q) \cap knn^*(q)|/k$. The $recall$ for a query set is averaged over all the queries. Besides, we use the $scan\_rate$ to evaluate the number of distance computations. Let $\#dists$ be the total number of distance computations and $scan\_rate = \#dists/n/nq$. We use $cache\_misses$ to indicate the total cache misses during massive queries and estimate it by the instruction `perf stat -e cache-misses` for each method.

Abbreviations: we compare query view and bridge view with three state-of-the-art proximity graphs, i.e., DPG (D for short), HNSW (H for short) and NSG (N for short). For simplicity, we use NsgQuery or NQ to represent query view with NSG and NsgBridge or NB to represent bridge view with NSG. As to the methods on large-scale data, we use the prefix "PAR-" to represent the partition based methods. For example, PAR-NQ indicates that we use NSG graphs as the index and take query view on each NSG graph during the search process. For each method, we carefully select its parameters in order to achieve the best performance.

5.2 Main experimental results

The purpose of this part is to demonstrate the advantages of bridge view over query view. Further, we explore the reasons behind the advantages of our method. Besides, we investigate the effects of key parameters on the performance of bridge view. All experiments are conducted on the moderate-scale data.



(a) Sift1M

(b) Deep1M

(c) Tiny1M

(d) Gist

Fig. 6 The main results of comparing two views

5.2.1 Comparing two views

We compare those two views with three state-of-the-art proximity graphs, i.e., DPG, HNSW and NSG, as shown in Fig. 6. Overall, bridge view obviously outperforms query view in almost all cases, especially with NSG and HNSW. Let us take NSG as an example. On Tiny1M, NsgBridge deals with 3× queries as many as NsgQuery when *recall* is around 0.98. Among all the six methods, NsgBridge has the best performance. This demonstrates the superiority of bridge view over query view.

The advantages of bridge view over query view are significantly affected by the dimensions. Bridge view has a larger advantage over query view on data with higher dimensions (i.e., Tiny1M and Gist) than data with lower dimensions (i.e., Sift1M and Deep1M). Let us take NSG as an example. The speedup of NsgBridge over NsgQuery on Tiny1M is as high as 3, while it is at most 1.5 on Sift1M. In addition, the index has an obvious effect on the performance in both views. NSG usually has the best performance in both views, while DPG has the worst one. This phenomenon is pretty significant on Tiny1M and Gist.

Notably, we obtain the performance curve for each method in Fig. 6 by varying the key search parameter *ef Search* between 20 and 320, in order to present its full search performance. As discussed above, $efSearch$ controls the tradeoff between efficiency and accuracy, i.e., enlarging $efSearch$ costs more but returns more accurate results.

5.2.2 Exploring two views

As discussed in Sect. 3.3, bridge view reduces the cache misses at the expense of repeated distance computations and increased memory space. To show this phenomenon, we present $scan\_rate$ and $cache\_misses$ in Fig. 7 and memory footprint in Fig. 8.

In Fig. 7, we can see that bridge view reduces the cache misses in a large scale w.r.t. query view, when achieving the same $recall$ value. Let us take NSG as an example and consider a specified $recall$ value such as 0.98. In this case, NsgBridge computes obviously more distances than NsgQuery, but produces far fewer cache misses. To be specific, NsgBridge computes 60% more distances than NsgQuery, but only produces 16% cache misses as many as NsgQuery on Gist. Similar phenomena could be found in other settings. This means that each distance computation in bridge view leads to fewer cache misses and even less cost than that in query view. The intuition behind bridge view is to enhance massive queries by reducing cache misses. Hence, we verify that our design fulfills our motivation.

On the other hand, bridge view requires more memory space than query view, which is its side effect. As in Fig. 8, the memory usage of NsgQuery keeps unchanged w.r.t. different $recall$ values, while that of NsgBridge grows as $recall$ increases for a larger $recall$. A larger $recall$ indicates a larger $efSearch$, which determines the size of $q.pool$ for each query q. Note that bridge view has to store $q.pool$ for each query q during the whole process, while query view does not. However, considering the improvement of bridge view in performance, we can tolerate such additional cost for the sake of search performance.

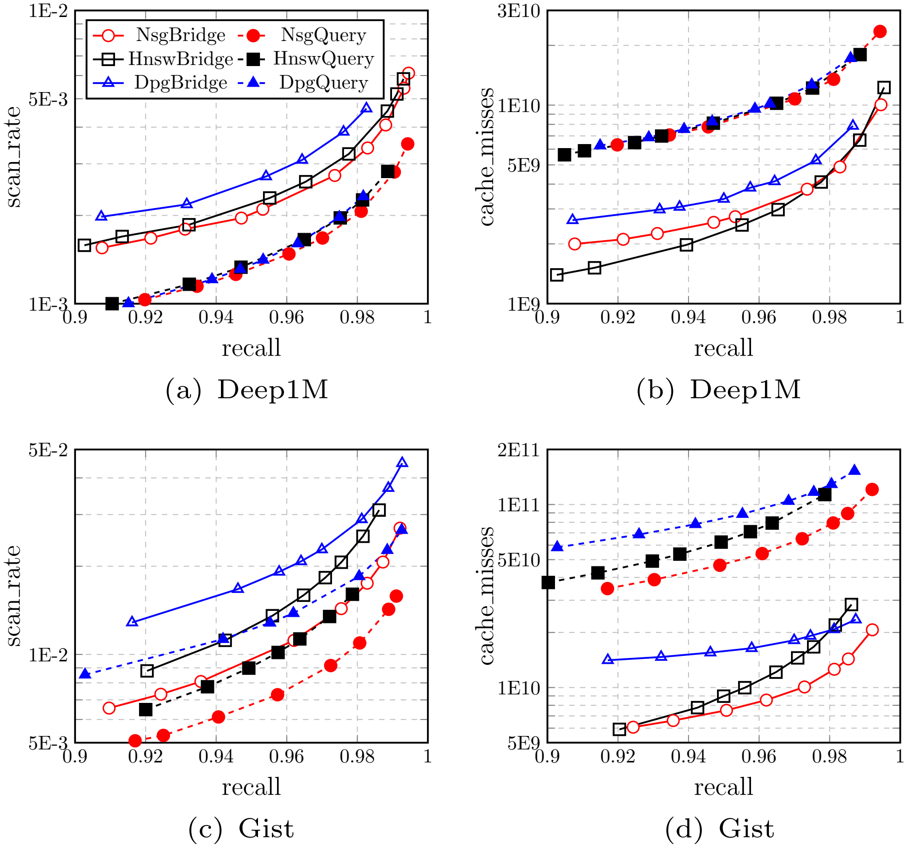
5.2.3 Effects of key parameters

In this part, we investigate the effects of three parameters on the performance of bridge view, i.e., the number T of threads, the parameter S and the number nq of queries. Besides, we have discussed the influence of $efSearch$ in Sect. 5.2.1.

Effects of the number T of parallel threads: Since we solve massive queries in multi-core CPUs, the number T of hardware threads will affect the performance. We show the results with NSG in Fig. 9. We can see that the advantages of NsgBridge over NsgQuery increase as the number T of threads rises. This means that bridge view will further enhance its performance for more hardware threads. In particular, NsgBridge with 8 threads obviously outperforms NsgQuery with 24 threads on Tiny1M.

Effects of the parameter S: Note that S indicates the maximum number of unexpanded vertices extracted for each query in each iteration. We show its effects in Fig. 10. Overall, the effects of S is pretty slight and a moderate value such as 10 usually achieves the best performance. If S is a small value such as 5, its efficiency significantly decreases as shown on Gist. This is because $\bar{Y}_i(u)$ will have fewer queries in each iteration and thus bridge view fails to make full use of the data locality. On the other hand, a larger S such as 15 may influence the quality of $\bar{Y}_i(u)$, especially for a small $efSearch$ value, as shown on Deep1M.

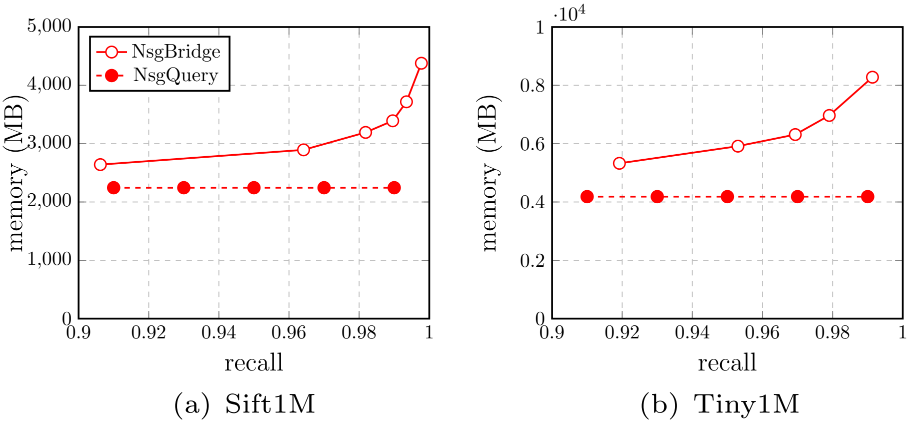
Effects of the number nq of queries: We also care about the effects of the number nq of queries on bridge view. We show the results of NsgBridge in Fig. 11. As nq increases, the performance of NsgBridge obviously grows. This is because a larger nq leads to a larger $\bar{Y}_i(u)$ for each bridge vertex u in each iteration, which reduces more cache misses. In contrast, query view is not affected by nq, since it deals with each query independently. This demonstrates that bridge view will further enhance its performance for more queries.



(a) Deep1M (b) Deep1M

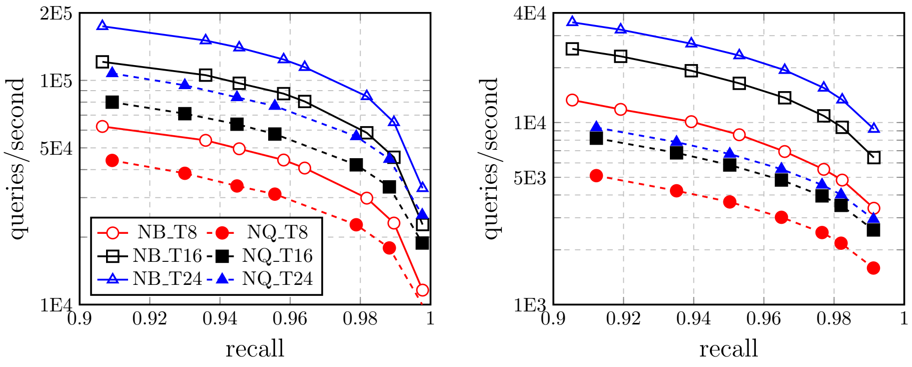
(c) Gist (d) Gist

Fig. 7 Comparing two views in *scan_rate* and *cache_misses*



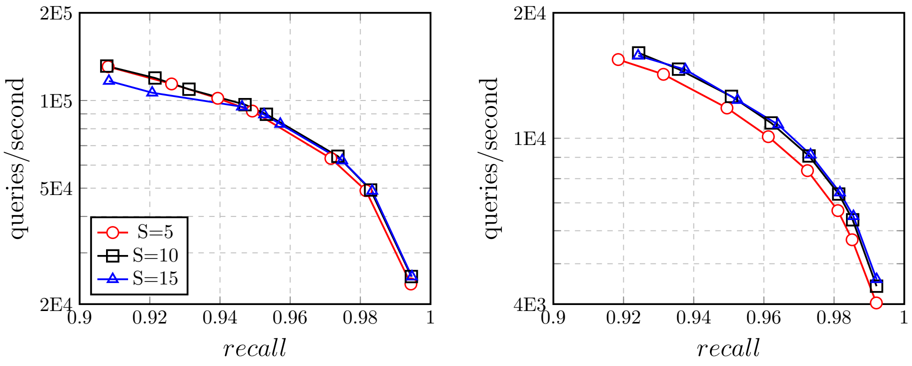
(a) Sift1M (b) Tiny1M

Fig. 8 Comparing two views of NSG in memory footprint



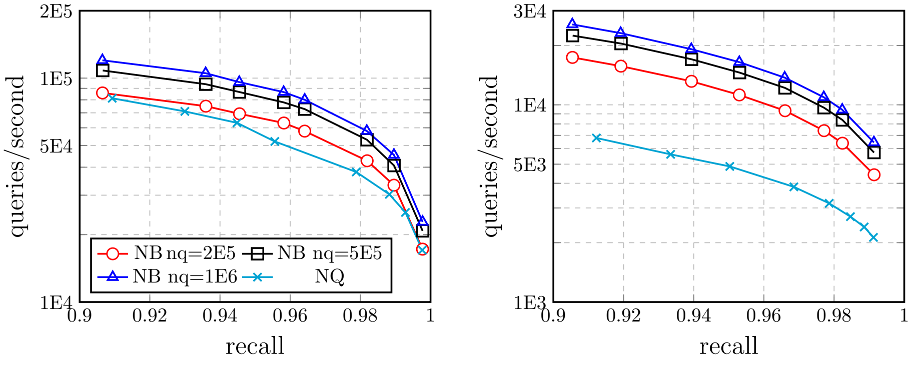
(a) Sift1M (b) Tiny1M

Fig. 9 Effects of the number T of parallel threads on two views of NSG. NB is short for NsgBridge and NQ for NsgQuery. T8 means 8 threads



(a) Deep1M (b) Gist

Fig. 10 Effects of the parameter S on NsgBridge



(a) Sift1M (b) Tiny1M

Fig. 11 Effects of the number nq of queries on two views of NSG

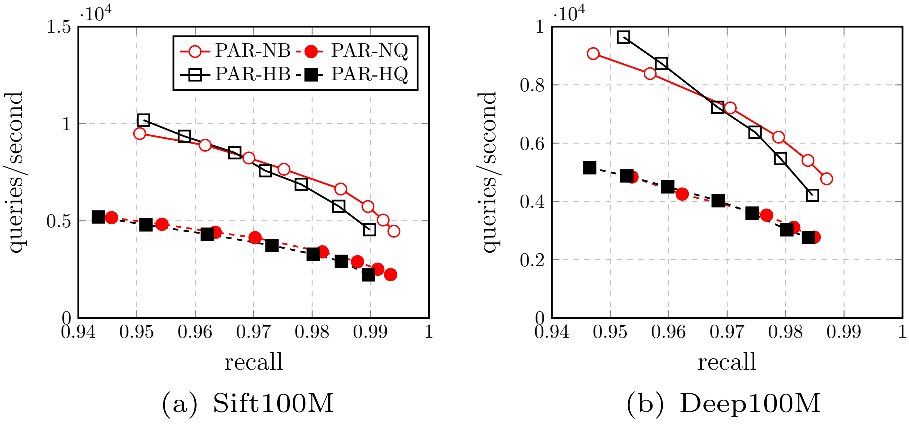


Fig. 12 Comparing two views of the partition based methods with 16 partitions on large-scale data

5.3 Experimental results on large-scale data

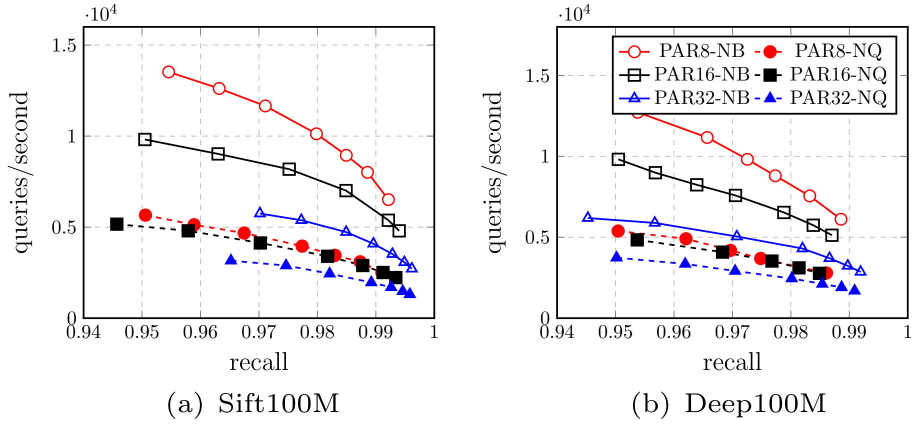
In this part, we show the experimental results of partition based methods, in order to demonstrate the advantages of bridge view over query view on large-scale data. As a key parameter for partition based methods, we investigate the effects of the number n_{par} of partitions. Moreover, we present the joint effects of n_{par} and the number T of threads. By default, we set the number T of threads as 32 in this part, unless specified.

First, we fix the number n_{par} of partitions as 16, as in [10], and vary the parameters $efSearch$ to observe the performance of different methods. The results are presented in Fig. 12. We can see that PAR-NB (PAR-HB) significantly outperforms PAR-NQ (PAR-HQ). This demonstrates that bridge view is able to enhance the partition based methods, as discussed in Sect. 4. Similar phenomenon could be found with DPG.

Second, we show the effects of n_{par} on the performance in Fig. 13, where we set three values of n_{par}, i.e., 8, 16 and 32. Here, we show the results with NSG for simplicity. We can see that a smaller n_{par} presents better performance. Clearly, fewer partitions enhance the connections among the points in the whole data set, while more partitions reduce those connections. The motivation of partition based methods is to reduce the construction cost and employ the hardware threads to accelerate the search process in parallel [10], but sacrifice the performance to some extent.

Lastly, we discuss the joint effects of n_{par} and T on the performance. We set $n_{par} \in \{8, 16, 32\}$ and $T \in \{8, 16, 24\}$. We compare the results in both PAR-NB and PAR-NQ with various pairs of n_{par} and T, as shown in Fig. 14, where all the experiments are conducted with the second workstation as presented in Sect. 5.1. As discussed above, either decreasing n_{par} or increasing T will improve the performance of both views. Hence, we can see that P8_T24 presents the best performance with the smallest n_{par} and the largest T in our settings. In addition, it is interesting that P8_T16 is pretty close to P16_T24 in performance, but obviously outperforms P32_T24. Similarly, P16_T16 is very close to P32_T24. Hence, to obtain the expected performance, we should carefully select the pairs of n_{par} and T, to balance the construction cost and the number of hardware threads required.

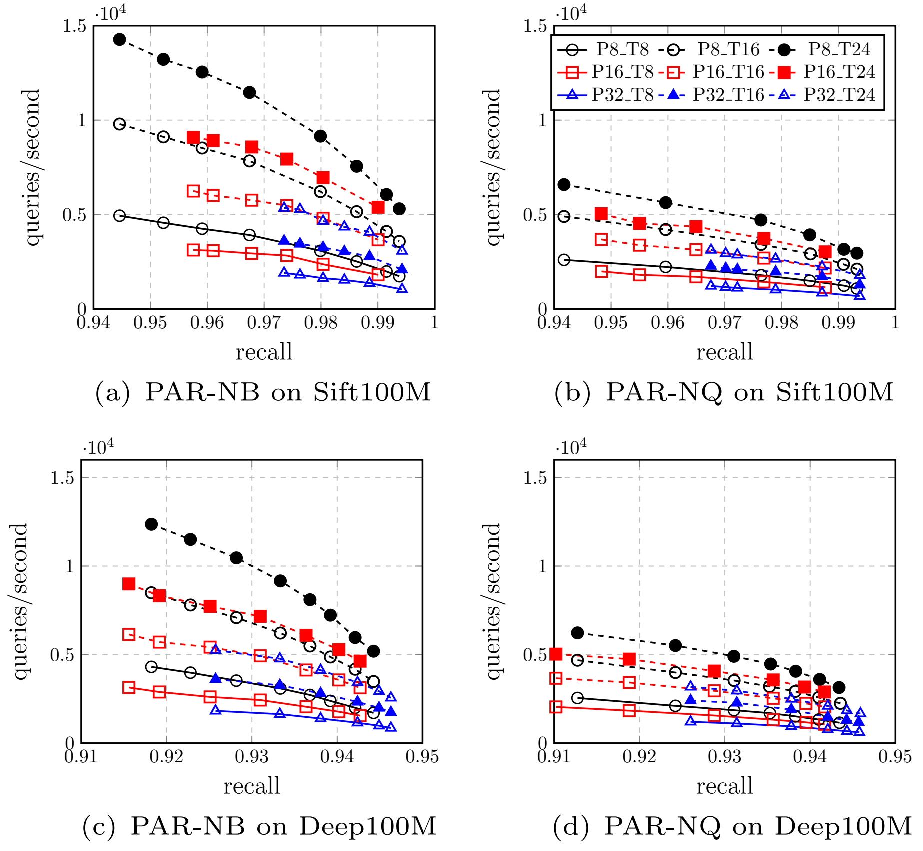
In all, we can see that our new method, i.e., bridge view, significantly outperforms its competitor, i.e., query view, for massive queries on large-scale data.



(a) Sift100M

(b) Deep100M

Fig. 13 Effects of the number n_{par} of partitions on the partition based methods, where n_{par} is set as 8, 16 and 32 respectively. For simplicity, PAR16 indicates the large data is divided into 16 partitions



(a) PAR-NB on Sift100M

(b) PAR-NQ on Sift100M

(c) PAR-NB on Deep100M

(d) PAR-NQ on Deep100M

Fig. 14 Joint Effects of the number n_{par} of partitions and the number T of parallel threads. P8_T16 indicates that we divide D into 8 partitions and use 16 hardware threads to conduct the search process

6 Related works

In this section, we introduce the existing works of four related problems, i.e., ANN search, MQ-ANN, KNN Join and concurrent BFS queries.

6.1 Approximate nearest neighbor search

There are a bulk of works on ANN search in high-dimensional space, due to its wide applications. A typical method builds an index and then accelerates the search process with the index, which smartly selects a small portion of the whole data as candidates and returns the accurate results from those candidates. In the early years, tree structures were very popular and thus many trees were created [19–22]. Locality sensitive hashing based methods have caused many attentions due to their theoretical guarantees [1, 23–25]. Recently, proximity graphs [8–10, 17] attract more and more attentions and have been demonstrated as the state-of-the-art index methods by recent benchmarks [6, 7]. *ANN search deals with a single query simultaneously, while our problem MQ-ANN deals with massive queries in the meantime.*

6.2 Massive queries of approximate nearest neighbor search

The typical idea of MQ-ANN is to create a smart scheduling method for an efficient index. Due to the number of queries, solutions to MQ-ANN prefer parallel platforms to improve the throughput. Some variants of existing index structures have been proposed to exploit the parallel computing platforms such as multi-core CPUs and many-core GPUs. Some researchers proposed to use multi-core CPUs to accelerate the search process [26, 27]. Some methods accelerate tree structures with GPUs [28, 29] and [30]. Pan et. al. proposed to accelerate LSH based methods with GPUs [31, 32]. In addition, there are some works that integrate both CPUs and GPUs together to accelerate similarity search [33, 34]. *In this paper, we focus on solving MQ-ANN by combining the pruning power of proximity graphs and the parallel power of multi-core CPUs.*

6.3 K-nearest neighbor join

As a primitive operation in data mining, KNN Join [12] is another problem that is highly related to our work. Like MQ-ANN, KNN Join usually builds index structures to accelerate the search process for a set of queries. However, those index structures have to be constructed online, while MQ-ANN builds them offline. Hence, KNN Join method has to balance the construction cost and the search performance of the selected index. Moreover, both KNN Join and MQ-ANN would take smart scheduling methods to exploit shared computations among multiple queries, in order to accelerate the search process with the index.

During the past decades, several methods have been proposed under various platforms. MuX [12] employs R-tree as the index to reduce I/O cost during the join process. Gorder [35] takes grid based ordering technique to group nearby data points together and then conducts the scheduled block nested join, in order to reduce the I/O cost and CPU cost. iJoin [36] uses iDistance [37] as the underlying index structure. Yao et al. [38] proposed to take z-order based index methods to conduct KNN Join in large relational databases. Moreover, there are methods [39, 40] that employ MapReduce to process KNN Join for large-scale data sets.

6.4 Concurrent breadth-first search queries

As another related problem, concurrent Breadth-First Search (BFS) Queries [15] is a fundamental operation in the domain of graph algorithms and has attracted a lot of attention for decades. As the classical method, MS-BFS [15] employs multi-core CPUs to accelerate multiple BFS queries. Its key idea is to share the access to the same neighbor list across multiple queries when they try to expand the same vertices in the same step. By this means, it is able to reduce the cache misses obviously. However, due to the nature of BFS, MS-BFS has to maintain a bitmap with the same size of the graph vertices for each query, in order to avoid repeated visits to the same vertex. As a result, the total number of concurrent BFS queries is significantly limited.

As mentioned above, our solution is inspired by the access-sharing strategy of MS-BFS, since they mainly expand a set of graph vertices. However, they are different in both **problems solved** and **solution details**. First, their problems solved, i.e., concurrent BFS queries vs MQ-ANN with proximity graphs, are different in two aspects. (1) MQ-ANN requires access to vectors and distance computations among them, while concurrent BFS queries have no such operations. (2) Each BFS query requires access to all the graph vertices, while each ANN query only a small part of the proximity graph for collecting close candidates. Second, their techniques, i.e., MS-BFS vs bridge view, are different in two aspects. (1) MS-BFS has to maintain a bitmap $visited$ for each query to avoid repeated accesses to the same graph vertex, while bridge view does not. (2) MS-BFS prefers a small query set (e.g., tens) due to the former aspect, while bridge view prefers a large query set (e.g., millions).

Besides, there are other methods that employ multi-core CPUs to enhance BFS queries [41, 42] in the literature. iBFS [43] employs GPU to accelerate concurrent BFS queries. On the other hand, Gorder [44] aims at reducing the cache misses for graph algorithms by reordering the graph vertices to keep vertices that will be frequently accessed together locally. We are inspired by MS-BFS that shares access to the same vertex among multiple queries, but adapt this idea to MQ-ANN with proximity graphs by a novel scheduling strategy.

7 Conclusion

In this paper, we aim at solving MQ-ANN in high-dimensional space by combining the pruning efficiency of proximity graphs and the parallel power of multi-core CPUs. The existing method called query view suffers from numerous cache misses, due to the nature of ANN search with a proximity graph. To relieve this issue, we propose a new method called bridge view, which reduces the cache misses in a large scale by improving the data locality. Further, we discuss how to extend bridge view to deal with large-scale data. Moreover, we conduct extensive experiments on real-life data sets to demonstrate the advantages of bridge view with three state-of-the-art proximity graphs, including DPG, HNSW and NSG. According to our experimental results, bridge view obviously outperforms query view for MQ-ANN in various settings.

Acknowledgements This work was supported by Project funded by National Natural Science Foundation of China (NSFC) under Grants 62002274, 61902299 and 61976168. This work was also supported in part by Project funded by China Postdoctoral Science Foundation under Grants 2019TQ0239 and 2019M663636.

References

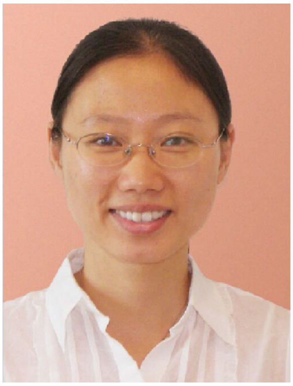
1. Sun Y, Wang W, Qin J, Zhang Y, Lin X (2015) SRS: solving c-approximate nearest neighbor queries in high dimensional Euclidean space with a tiny index. PVLDB 8(1):1–12
2. Huang Q, Feng J, Zhang Y, Fang Q, Ng W (2016) Query-aware locality-sensitive hashing for approximate nearest neighbor search. PVLDB 9(1):1–12
3. Muja M, Lowe DG (2014) Scalable nearest neighbor algorithms for high dimensional data. IEEE Trans Pattern Anal Mach Intell 36(11):2227–2240
4. Jegou H, Douze M, Schmid C (2011) Product quantization for nearest neighbor search. IEEE Trans Pattern Anal Mach Intell 33(1):117–128
5. Babenko A, Lempitsky V (2014) The inverted multi-index. IEEE Trans Pattern Anal Mach Intell 37(6):1247–1260
6. https://github.com/spotify/annoy
7. https://github.com/searchivarius/nmslib
8. Li W, Zhang Y, Sun Y, Wang W, Li M, Zhang W, Lin X (2019) Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. IEEE Trans Knowl Data Eng 32:1475–1488
9. Malkov Y, Yashunin D (2018) Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Trans Pattern Anal Mach Intell 42:824–836
10. Fu C, Xiang C, Wang C, Cai D (2019) Fast approximate nearest neighbor search with the navigating spreading-out graph. PVLDB 12(5):461–474
11. Baranchuk D, Babenko A, Malkov Y (2018) Revisiting the inverted indices for billion-scale approximate nearest neighbors. In: ECCV, pp 202–216
12. Böhm C, Krebs F (2004) The k-nearest neighbour join: turbo charging the KDD process. Knowl Inf Syst 6(6):728–749
13. Dong W, Moses C, Li K (2011) Efficient k-nearest neighbor graph construction for generic similarity measures. In: Proceedings of the 20th international conference on world wide web, pp 577–586
14. Liu Y, Cheng H, Cui J (2021) Revisiting k-nearest neighbor graph construction on high-dimensional data: experiments and analyses. arXiv Preprint arXiv:2112.02234
15. Then M, Kaufmann M, Chirigati F, Hoang-Vu T-A, Pham K, Kemper A, Neumann T, Vo HT (2014) The more the merrier: efficient multi-source graph traversal. PVLDB 8(4):449–460
16. Dong W, Moses C, Li K (2011) Efficient k-nearest neighbor graph construction for generic similarity measures. In: WWW, pp 577–586
17. Malkov Y, Ponomarenko A, Logvinov A, Krylov V (2014) Approximate nearest neighbor algorithm based on navigable small world graphs. Inf Syst 45:61–68
18. Chen J, Fang HR, Saad Y (2009) Fast approximate kNN graph construction for high dimensional data via recursive Lanczos bisection. J Mach Learn Res 10(9):1989–2012
19. Bentley JL (1975) Multidimensional binary search trees used for associative searching. Commun ACM 18(9):509–517
20. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. In: SIGMOD, pp 47–57
21. Berchtold S, Keim DA, Kriegel H-P (1996) The X-tree: an index structure for high-dimensional data. In: VLDB, pp 28–39
22. Jagadish HV, Ooi BC, Tan KL, Yu C, Zhang R (2005) iDistance: an adaptive B^+-tree based indexing method for nearest neighbor search. ACM Trans Database Syst 30(2):364–397
23. Tao Y, Yi K, Sheng C, Kalnis P (2009) Quality and efficiency in high dimensional nearest neighbor search. In: SIGMOD, pp 563–576
24. Gan J, Feng J, Fang Q, Ng W (2012) Locality sensitive hashing scheme based on dynamic collision counting. In: SIGMOD, pp 541–552
25. Liu Y, Cui J, Huang Z, Li H, shen H (2014) SK-LSH?: an efficient index structure for approximate nearest neighbor search. PVLDB 7(9):745–756
26. Uribe-Paredes R, Valero-Lara P, Arias E, Sánchez JL, Cazorla D (2011) Similarity search implementations for multi-core and many-core processors. In: HPCS. IEEE, pp 656–663
27. Gedik B (2013) Auto-tuning similarity search algorithms on multi-core architectures. Int J Parallel Prog 41(5):595–620
28. Gieseke F, Heinermann J, Oancea C, Igel C (2014) Buffer kd trees: processing massive nearest neighbor queries on GPUs. In: ICML, pp 172–180
29. Kim M, Liu L, Choi W (2018) A GPU-aware parallel index for processing high-dimensional big data. IEEE Trans Comput 67(10):1388–1402
30. Kim J, Hong S, Nam B (2012) A performance study of traversing spatial indexing structures in parallel on GPU. In: HPCC. IEEE, pp 855–860

31. Pan J, Manocha D (2011) Fast GPU-based locality sensitive hashing for k-nearest neighbor computation. In: SIGSPATIAL GIS, pp 211–220
32. Pan J, Manocha D (2012) Bi-level locality sensitive hashing for k-nearest neighbor computation. In: ICDE. IEEE, pp 378–389
33. Matsumoto T, Yiu ML (2015) Accelerating exact similarity search on CPU-GPU systems. In: ICDM. IEEE, pp 320–329
34. Wang Y, Shrivastava A, Wang J, Ryu J (2018) FLASH: randomized algorithms accelerated over CPU-GPU for ultra-high dimensional similarity search. In: SIGMOD, pp 889–903
35. Xia C, Lu H, Ooi BC, Hu J (2004) Gorder: an efficient method for KNN join processing. In: VLDB, pp 756–767
36. Yu C, Cui B, Wang S, Su J (2007) Efficient index-based KNN join processing for high-dimensional data. Inf Softw Technol 49(4):332–344
37. Jagadish HV, Ooi BC, Tan KL, Yu C, Zhang R (2005) iDistance: an adaptive B$^+$-tree based indexing method for nearest neighbor search. ACM Trans Database Syst 30(2):364–397
38. Yao B, Li F, Kumar P (2010) K nearest neighbor queries and KNN-joins in large relational databases (almost) for free. In: ICDE, pp 4–15
39. Lu W, Shen Y, Chen S, Ooi B (2012) Efficient processing of k nearest neighbor joins using mapreduce, pp 1016–1027
40. Zhang C, Li F, Jestes J (2012) Efficient parallel KNN joins for large data in MapReduce. In: EDBT, pp 38–49
41. Bader DA, Madduri K (2006) Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray MTA-2. In: ICPP, pp 523–530
42. Chhugani J, Satish N, Kim C, Sewall J, Dubey P (2012) Fast and efficient graph traversal algorithm for CPUs: maximizing single-node efficiency. In: IPDPS, pp 378–389
43. Liu H, Huang HH, Hu Y (2016) iBFS: concurrent breadth-first search on GPUs. In: SIGMOD, pp 403–416
44. Wei H, Yu JX, Lu C, Lin X (2016) Speedup graph processing by graph ordering. In: SIGMOD, pp 1813–1828

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Yingfan Liu is currently an assistant professor in the School of Computer Science and Technology, Xidian University, China. He obtained his Ph.D. degree in the Department of Systems Engineering and Engineering Management at Chinese University of Hong Kong in 2019. His research interests include the management of large-scale complex data and the adaptive query optimization for DBMS.

Chaowei Song is currently studying at the School of Computer Science and Technology, Xidian University in China. He obtained a bachelor's degree in computer science and technology from Xidian University in 2020. His research interest contains the management of large-scale complex data such as high-dimensional data and efficient parallel algorithms.



Hong Cheng is a full professor in the Department of Systems Engineering and Engineering Management at the Chinese University of Hong Kong. She received her Ph.D. degree from University of Illinois at Urbana-Champaign in 2008. Her research interests include data mining, database systems, and machine learning. She received research paper awards at ICDE'07, SIGKDD'06 and SIGKDD'05, and the certificate of recognition for the 2009 SIGKDD Doctoral Dissertation Award. She is a recipient of the 2010 Vice-Chancellor's Exemplary Teaching Award at the Chinese University of Hong Kong.

Xiaofang Xia is currently an associate professor with the School of Computer Science and Technology, Xidian University, China. She received her Ph.D. degree in Control Theory and Control Engineering from Shenyang Institute of Automation, Chinese Academy of Sciences, China, in 2019. She was a visiting student at the Department of Computer Science, University of Alabama, USA, from August 2016 to February 2018. Her research interests are mainly in cyber physical systems, smart grid security, database management system and anomaly detection.

Jiangtao Cui received the MS and Ph.D. degrees both in computer science from Xidian University, China, in 2001 and 2005, respectively. Between 2007 and 2008, he was with the Data and Knowledge Engineering group working on high-dimensional indexing for large scale image retrieval, in the University of Queensland, Australia. He is currently a professor in the School of Computer Science and Technology, Xidian University, China. His current research interests include database systems, data and knowledge engineering, data security, and high-dimensional indexing.