Dynamic Group Nearest Neighbor Group Query over Streaming Data

Yunzhe An[1] Sainan Tong[1] Qian Wang[2] Rui Zhu[1,3] Anzhen Zhang[1]
Chuanyu Zong[1] Hong Jiang[4] Bin Wang[5]

[1] Shenyang Aerospace University, China
[2] Shenyang Aircraft Corporation, China
[3] Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, JiLin University, China
[4] Shenyang university of technology, China
[5] Northeastern University, China
{anyunzhe,tongsainan}@mail.sau.edu.cn,wangxinwpu91@163.com, {zhurui,azzhang,zongcy}@mail.sau.edu.cn, jianghong@sut.edu.cn, binwang@mail.neu.edu.cn

Abstract. This paper studies the problem of dynamic group nearest neighbor group query (DGNNQ for short) over streaming data, an important problem in the domain of streaming data management. Let $\mathcal{S}$ be the set of streaming data. A DGNNQ, denoted as $q(n, s, k, \mathcal{Q})$, monitors objects within the window that contains $q(n)$ objects. Whenever $q(s)$ objects pass, q returns $q(k)$ objects with the smallest *distance sum* to the query point set $\mathcal{Q}$. Although some efforts can support this kind of queries, they incur highly running cost, especially when query points are allowed to be inserted into, or remove from $\mathcal{Q}$.
In this paper, we propose a novel framework named Q2WP (short for Query points and Window-based Partition) over streaming data. We group nearby query points into k subsets, and form a group of k virtual points based on these subsets. We then propose a novel index named PM-Tree to organize these virtual points, as well as evaluate which objects have chance to become query result objects based on PM-Tree. We further propose a group of novel algorithms to support incremental maintenance when $\mathcal{Q}$ updates. Extensive performance studies on large real-world and synthetic datasets demonstrate that the proposed framework can efficiently support DGNNQ over streaming data.

Keywords: Streaming Data · PM-Tree · Dynamic Group Nearest Neighbor Group Query · Partition

1 Introduction

Dynamic group nearest neighbor group query (DGNNQ for short) over streaming data is an important problem in the domain of streaming data management[4, 13, 5]. It finds applications in various domains [3, 10, 2], including takeout delivery systems, point-of-interest recommendation systems, healthcare resource optimization in healthcare and bioinformatics, and so on.

Let $\mathcal{S}$ be the set of streaming data. Each DGNNQ, denoted as $q(n, s, k, \mathcal{Q})$, monitors objects within the window that contains $q(n)$ objects. Whenever $q(s)$ objects pass, q returns $q(k)$ objects with the smallest *distance sum* to these query points (Detailed see Section 2.2). Here, $\mathcal{Q}$ denotes the query point set containing a set of query points. Without loss of generality, it is allowed that new queries are entered into $\mathcal{Q}$, or deleted from $\mathcal{Q}$. Furthermore, we use sliding window to model streaming data. The window can be time-based or count-based. For simplicity, in this paper, we focus only on count-based windows. However, our techniques also can be applied to support DGNNQ over time-based window.

Many efforts have been proposed to support group nearest neighbor group query [4, 8, 9]. Their key idea is to use pruning strategies to reduce the search space. However, these methods cannot support DGNNQ under data stream. The main reason is that both the query point set $\mathcal{Q}$ and objects within the window change over time, leading that these algorithms have to re-scan the window and re-compute the NN for all new query points, resulting in highly computational costs. Here, an object o is considered as a nearest neighbor (NN for short) of a query point q_i if the distance between o and q_i is the smallest of all.

Only one effort [13] can support group nearest neighbor group query under data stream. Their core idea is to use a quadtree-based index to manage streaming data. Additionally, they partition query points into a group of subsets, and support group nearest neighbor group query based on the partition result. However, in many cases, it cannot form a high quality partition. Besides, this algorithm still has to spend highly running cost when the query point set dynamically changes. Last of all, when the query parameter k is large, the running cost of the algorithm is still high, which cannot support DGNNQ in real-time. Therefore, an efficient algorithm that can efficiently support DGNNQ in real-time is desired.

In this paper, we propose a novel framework named Q2WP (short for Query points and Window-based Partition) to support DGNNQ over streaming data. It is based on the following observation. That is, let q, q' be two query points within $\mathcal{Q}$, o be an object. If the distance between q and o (also q and q') is small, the distance between q' and o is not large. Based on the above observation, we partition query points within $\mathcal{Q}$ into k subsets, i.e., called as q-subsets. We should guarantee that, after partitioning, query points within the same q-subset are near to each other, while query points within the different q-subsets are far from each other as much as possible. Based on the partition result, we further form a group of virtual points, i.e., called as core points. In this way, we can support DGNNQ via monitoring o-NN(short for object-based NN) of each core point, return these k o-NNs to the system when the window slides, i.e., regarded them as query result objects.

Here, we call an object o within the window W as the o-NN of a core point c if o has the smallest distance to c among all objects within the window, i.e., $\forall o' \in W - o$, $\mathrm{D}(c, o)$ is smaller than $\mathrm{D}(c, o')$. Compared with evaluating distance relationships among ALL query points and objects within the window, the run-

ning cost of monitoring o-NN for each core point could be significantly reduced. However, we also should address the following three challenges.

Firstly, it is difficult to find an effective way to form these k q-subsets. Obviously, after forming, if objects within the same q-subset are not near to each other, we cannot return high quality query results to the system. Secondly, if k is relatively large, it is also difficult to monitor o-NN for each core point. As it is equivalent to support k continuous NN search at the same time. Last of all, when the query point set is updated, we may have to incur high running cost to re-partition the query point set, as well as re-find o-NNs for new core points. Under such cases, it is difficult to meet the real time requirement of users.

In this paper, we propose efficiently algorithms to address these challenges. Contributions of this paper are as follows.

- We propose a novel query named dynamic group nearest neighbor group query over streaming data. To the best of our knowledge, this is the first work to address the problem of DGNNQ over streaming data.
- We propose a novel algorithm, named NNP (short for Nearest Neighbor-based Partition) for supporting q-subsets construction. This algorithm is based on the nearest neighbor pair search, where we repeatedly find the nearest neighbor pair from all nearest neighbor pairs, merge the corresponding elements to the same q-subsets, until all query points are merged into k q-subsets.
- We propose a novel index called the PM-Tree(short for Partition-based with M-Tree) to organize these core points. Compared with traditional M-Tree, it provides us with a group of thresholds for evaluating the chance of newly arrived objects becoming query result objects. It helps us efficiently monitor o-NN of core points.
- We propose a novel algorithm to support query points incremental maintenance. It is ensured that, when some queries are entered into, or deleted from $\mathcal{Q}$, we need not to update core points as low as possible. Even needed, we only need to use lower cost to support o-NN search.

The rest of this paper is as follows. Section 2 reviews related work and presents the problem definition. Section 3 explains the framework Q2WP. Section 4 explains the incremental maintenance algorithms. Section 5 evaluates the performance of Q2WP. Section 6 concludes this paper.

2 Preliminary

In this section, we first review some important existing results related to k-nearest neighbor query and group nearest neighbor group query. Next, we introduce the problem definition.

2.1 Related Works

Nearest neighbor query over data stream is an important problem in the domain of streaming data management, which has been well-studied for over 10 years.

Existing algorithms can be divided into two types: *k-nearest neighbor query algorithms* [6, 7] and *group nearest neighbor group query algorithms* [4, 12, 13]. In the following, we first explain k-nearest neighbor query algorithms.

Among all efforts, Lee et al.[6] proposed the algorithm named PKNN-Grid, which efficiently addresses kNN queries by leveraging spatial proximity. It precomputes kNN for each object and identifies the nearest neighbor o of q. Based on the searching result, the algorithm quickly retrieves query results. Another notable contribution in this area is an efficient index named LαB. It is proposed by Miao et al.[7]. It utilizes a grid-based structure to partition objects, support kNN queries over incomplete data. They also proposed the algorithm named LP, which uses two pruning parameters (α and distance) to eliminate irrelevant objects, improves query performance.

Moreover, Yang et al.[11] proposed a method that combines an extended quad-tree with an adaptive block-based ordered inverted index to efficiently evaluate continuous kNN over spatial-textual data streams. They introduced a memory-based cost model and k-Skyband technique to optimize the search range. Bahri et al.[1] developed a sliding-window-based algorithm to support kNN search under data stream, which emphasizes recent data with dynamic weights to improve classification accuracy and efficiency.

Group nearest neighbor group query (GNNGQ for short), a variant of the kNN query, was first introduced by Papadias et al.[8]. They prune the search space using the minimum bounding rectangle of query points. Several subsequent methods aim to improve the efficiency of GNNGQ. Deng et al.[4] proposed algorithms named EHC and SHR. EHC employs exhaustive combinations within hierarchical subsets. However, the k is large, its running cost is still high. To address this problem, Deng et al. further proposed the algorithm named SHR. It is an approximate approach that focuses on pruning irrelevant subsets through hierarchical data blocks. However, its efficiency declines when query points undergo dynamic changes or when dealing with high-speed data flows.

Xu et al.[9] developed the algorithm named ADM for supporting continuous GNNGQ, leveraging hill-climbing within candidate sets to reduce complexity. Furthermore, Zhu et al.[13] introduced the KMPT algorithm for continuous GNNGQ over streaming data. It uses K-Means algorithm to partition query points into a group of subsets, and then support continuous GNNGQ based on the partition result. However, it still has to spend high running cost when the query point set dynamically changes.

2.2 Problem Definition

In this section, we first introduce the concept of group nearest neighbor group query over streaming data.

Let $\mathcal{D}$ denote a set of objects, and $q(k, \mathcal{Q})$ be a query that contains a group of query point set. The objective of GNNGQ is to select k object from $\mathcal{D}$ such that, the sum of Euclidean distances between each query point and its nearest selected object point is minimized. In a streaming data environment, it is extended to support continuous queries. To be more specially, a continuous GNNGQ query,

denoted as $q(n, s, k, \mathcal{Q})$, monitors the window W. Whenever the window slides, the system returns the k object points such that, the sum of Euclidean distances between each query point and its nearest selected object point is minimized.

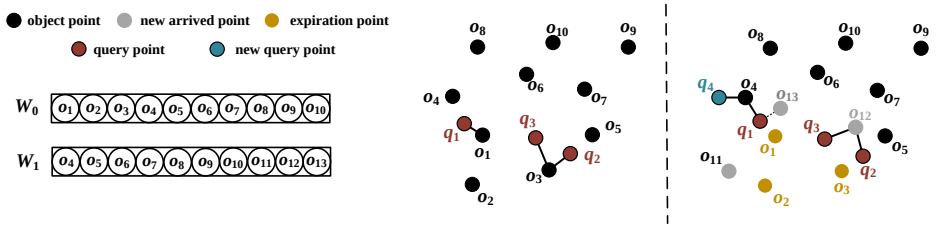


Fig. 1. The example of a dynamic group nearest neighbor group query when $n = 10$, $k = 2$, and $s = 3$.

Take the example in Fig.1. Let $n = 10$, $k = 2$, and $s = 3$. Suppose there are 10 object points $\{o_1, o_2, \ldots, o_{10}\}$ within window W_0, and the query point set $\mathcal{Q}$ contains 3 query points $\{q_1, q_2, q_3\}$. We can calculate the distances between these 3 query points and each object point in the window W_0. Since the total distance between $\{q_1, q_2, q_3\}$ and object points o_1 and o_3 is the smallest of all within window W_0, the object set $\{o_1, o_3\}$ is returned. When the window slides to W_1, objects $\{o_1, o_2, o_3\}$ expire from the window, while objects $\{o_{11}, o_{12}, o_{13}\}$ flow into the window. At that moment, the total distance from o_{12} and o_{13} to the query points $\{q_1, q_2, q_3\}$ is minimized. Thus, we return $\{o_{12}, o_{13}\}$ to the system.

We now formally explain the problem definition. Here, $\mathrm{D}(o_i, q_j)$ denotes the Euclidean distance between object o_i and query point q_j. Given a set of query points $\mathcal{Q}$, $|\mathcal{Q}|$ refers to the scale of $\mathcal{Q}$.

Definition 1. **DGNNQ**. *Let $\mathcal{S}$ be the set of streaming data, a query q, denoted as $q(n, s, k, \mathcal{Q})$, monitors objects within the window that contains $q(n)$ objects. Whenever $q(s)$ objects pass, q returns $q(k)$ objects with the smallest distance sum to query points within the query point set $\mathcal{Q}$, i.e., $\mathcal{Q} = \{q_1, q_2, \cdots, q_{|\mathcal{Q}|}\}$. Here, it is allowed that new queries are entered into $\mathcal{Q}$, or deleted from $\mathcal{Q}$. Additionally, the distance sum d is calculated based on Equation (1).*

$$d = \sum_{i=1}^{k} \sum_{q_j \in \mathcal{Q}} D(o_i, q_j) \tag{1}$$

Back to the example shown in Fig.1. When the window slides to W_1, the query point q_4 is inserted into $\mathcal{Q}$. Based on distance calculations, the minimum total distance is achieved by selecting o_4 (closest to q_4 and q_1) and o_{12} (closest to q_2 and q_3). Thus, the updated query result is $\{o_4, o_{12}\}$.

3 The Framework Q2WP

In this section, we propose a novel framework named Q2WP (short for Query points and Window-based Partition) for supporting DGNNQ. First, we introduce the query point partition algorithm.

3.1 The Query Point Partition Algorithm

In this section, we propose a novel algorithm named NNP(short for Nearest Neighbor-based Partition), designed to partition the set of query points into a group of subsets, referred to as q-subsets. Based on the partition result, the algorithm generates k points, referred to as core points, based on these q-subsets. Subsequently, it monitors the NN of these core points and use these searched NN to support DGNNQ. Therefore, it is significant to find high quality core points. Our goal is to minimize the distances between points within each q-subset while maximizing the distances between points across different q-subsets.

Formally, we first find the nearest neighbor for each query point, and then form a pair for each query point $q_i \in \mathcal{Q}$ and its nearest neighbor query point. For simplicity, we call a pair (q, q') as a NN pair if q is the nearest neighbor of q', and vice versa. The distance between q and q' is used as the score of (q, q'). Consider the running example in Fig.2. There are 9 query points within the query set, i.e., $\mathcal{Q} = \{q_1, q_2, \ldots, q_9\}$. We then form NN pairs for each query point q within $\mathcal{Q}$ and its nearest neighbor query point. The results are shown in Fig.2(a).

We then partition these query points based on their distance relationships. Specifically, we first find the NN pair (q_i, q_i') with the smallest score. Next, we form a q-subset C_1 using q_i and q_i', i.e., calculate the center point between q_i and q_i', use this center point as a temporarily core point c_1. Furthermore, we find the nearest neighbor (also reverse nearest neighbors) of c_1, and update the NN pair set accordingly. As shown in Fig.2(b), the pair (q_1, q_2), formed by q_1 and q_2, has the smallest score. We merge them into C_1, and form a core point c_1 based on q_1 and q_2, i.e., the center of q_1 and q_2. Subsequently, we re-find the nearest neighbor of c_1, form a new pair accordingly, i.e., the pair (q_4, c_1). Additionally, as pairs (q_1, q_4) and (q_2, q_5) turn to meaningless, we re-find nearest neighbors for both q_4 and q_5, and then form new NN pairs. Finally, we update the NN pair set.

From then on, we repeatedly merge query points until all query points are merged into k subsets. Here, the merging process can be summarized into the following three cases:

Case 1: Both points are query points. we follow the logic discussed above to apply the merge operations. Back to the example in Fig.2(b), where the query points q_3 and q_7 are involved, we process them in the same manner as q_1 and q_2.

Case 2: One is a core point c, and the other is a query point q. Assume that the core point c belongs to the subset C. In this case, q is inserted into C, and we update the nearest neighbors for the reverse nearest neighbors of q. As shown in Fig.2(c), the pair (q_4, c_1) has the smallest score, so we insert q_4 into C_1, forming a new subset C_1'. The core point of C_1' is then recalculated as v_1', and v_1' is assigned new NN, forming a new pair (v_4, v_1'). After insertion,

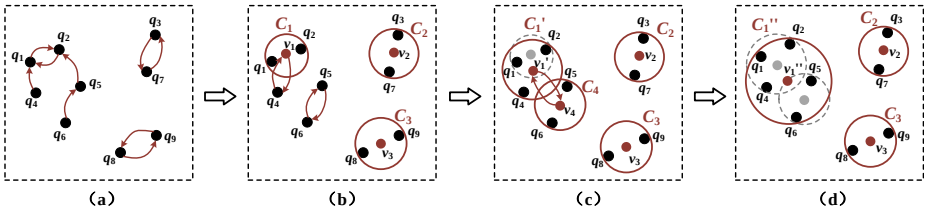


Fig. 2. The example of query point grouping with $k = 3$.

4 subsets remain, which is still greater than the target $k = 3$, so the merging process continues.

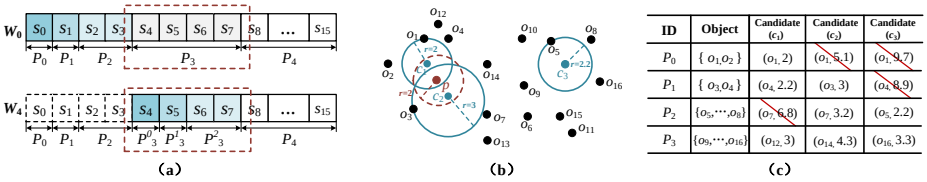
Case 3: Both points are core points. Let we assume that core points c and c' belong to subsets C and C', respectively. In this case, We merge C and C' into one subset C'', where we form a new core point c'' based on all query points within C''. The original two core points are deleted. Finally, we update nearest neighbor relationships among core points and query points following the logic discussed before. As shown in Fig.2(c), we merge C'_1 and C_4, re-form a core point, and so on. At that moment, the number of subsets is reduced to 3, i.e., C''_1, C_2 and C_3. As $k = 3$, we can stop the merging, i.e., $C''_1 = \{q_1, q_2, q_4, q_5, q_6\}$, $C_2 = \{q_3, q_7\}$, and $C_3 = \{q_8, q_9\}$.

Discussion. The algorithm discussed above may spend higher running cost, as the running cost of NN search is high. Therefore, we can use the algorithm discussed [14] to monitor 1NN pair within the data set. Compared with monitoring nearest neighbor of every core point/query point, this algorithm only monitors nearest neighbor of partial core point/query point. For the others, this algorithm only calculate the distance lower-bound between them and their nearest neighbors. Therefore, the running cost of each update is bounded by $\mathcal{O}(\log |\mathcal{Q}|)$. After forming these k q-subsets, the overall running cost is bounded by $\mathcal{O}((|\mathcal{Q}| - k) \log |\mathcal{Q}|)$, i.e.,$\mathcal{O}(|\mathcal{Q}| \log |\mathcal{Q}|)$.

3.2 PM-Tree: A Partition-Based Index

Let W_O denote the set of objects within the query window, and let $\mathcal{C}$ represent the set of core points. We aim to monitor object-based NN(o-NN for short) for every elements within $\mathcal{C}$. Here, an object o within W_O is called as o-NN of $c \in \mathcal{C}$ if the distance between o and c is the smallest of all among all objects within W_O.

However, in a data stream environment, objects within the window updates frequently, it will spend highly running cost to monitor o-NN for each core point within $\mathcal{C}$, especially when the scale of $\mathcal{C}$ is large. To address this problem, we propose a novel index named PM-Tree to organize core points, as well as another partition-based sketch to support o-NN maintenance. They are based on the following two useful observations.



ID	Object	Candidate (c_1)	Candidate (c_2)	Candidate (c_3)
P_0	{ o_1,o_2 }	(o_1, 2)	(o_1, 5.1)	(o_1, 9.7)
P_1	{ o_3,o_4 }	(o_4, 2.2)	(o_3, 3)	(o_4, 8.9)
P_2	{o_5,⋯,o_8}	(o_7, 6.8)	(o_7, 3.2)	(o_5, 2.2)
P_3	{o_9,⋯,o_{16}}	(o_{12}, 3)	(o_{14}, 4.3)	(o_{16}, 3.3)

Fig. 3. The example of partition and observation with $s = 2$

observation 1. *Let O be the set of objects, $NN(O)$ be the set of their nearest neighbors, with distances $\{d_1, d_2, \cdots, d_{|O|}\}$. Let we assume that objects within O are bounded by a ball $b(p, r)$, and o is another object out of O. If $D(p, o) \geq 2 \cdot r + \max(d_1, d_2, \cdots, d_{|O|})$, o is not the nearest neighbor of any object within O, with p and r being the center and radius of the ball $b(p, r)$.*

Observation 1 provides us with an effective way to identify o-NNs for core points. As illustrated in Fig. 3(b), given two core points c_1 and c_2, their o-NNs are o_1 and o_3, i.e., D(c_1, o_1)=2 and D(c_2, o_3)=3. We use a ball $b_{12}(p, 2)$ to bound both c_1 and c_2. Suppose o_{16} is a newly arrived object that flows into the window. Since the distance between $b_{12}(p)$ and o_{16} is larger than max(3,2)+2· 2, we ensure that o_{16} is not the o-NN of both c_1 and c_2. This observation inspires us to use a distance-based index, e.g., M-Tree, to organize core points, which helps us efficiently prune objects that cannot become o-NN of any core point within $\mathcal{C}$.

observation 2. *Let c be a core point, o_1 and o_2 be two objects within the query window. If o_1 arrives later than o_2, and $D(c, o_1)$ is smaller than $D(c, o_2)$, then o_2 cannot become the o-NN of c before it expires from the window.*

Observation 2 provides us with a method of using a small number of objects within the query window to support DGNNQ. In Fig. 3(b) and (c), o_4 arrives later than o_2, and has a smaller distance to c_1. It means o_2 cannot serve as the o-NN of c_1 before it expires from the window. In contrast, o_{12} arrives later than o_4, but D(o_{12}, c_1) is larger than D(o_4, c_1), o_{12} still has chance to become o-NN of c_1 after o_4 expires from the window. Therefore, we could use $\{o_1, o_4, o_{12}\}$ as candidate o-NN of c_1. Similarly, we can use $\{o_3, o_7, o_{14}\}$ and $\{o_5, o_{16}\}$ as candidate o-NN of c_2 and c_3, respectively.

Motivated by these two observations, we introduce a partition-based method. It partitions the query window into a group of sub-windows. Based on the partition results, we further propose a novel index named PM-Tree to organize core points, as well as support DGNNQ. For simplicity, $|P_i|$ refers to the number of objects within P_i.

Definition 2. Partition. *Let $W_0 = \{s_0, s_1, \ldots, s_{n-1}\}$ be the current window, $\mathcal{P}$ be a partition that divides objects in W_0 into a group of m disjoint partitions $\mathcal{P}\{P_0, P_1, \cdots, P_{m-1}\}$. It should satisfy:*

- *(i) $\forall o \in P_i$, $\forall o' \in P_j$, if $i < j$, the arrival order of objects in P_i is earlier than that of objects in P_j;*

– *(ii) for any partition* $P_i \in \mathcal{P}(i \neq 0)$, $|P_i| = \sum_{j=0}^{j=i-1} |P_j|$ *is satisfied.*

One reason we introduce the partition is to select a smaller number of objects from each partition as candidate o-NNs of core points, use them to support DGNNQ as efficiently as possible. Formally, we form a group of tuples $\{t_0^u(\theta, o), t_1^u(\theta, o), \cdots, t_{m-1}^u(\theta, o)\}$ for each core point $c_u \in \mathcal{C}$. Here, given $t_i^u(\theta, o)$, $t_i^u(o)$ is the o-NN of c_u among objects in P_i, and θ is the distance between $t_i^u(o)$ and c_u. We call the union of these tuples as P-SKETCH of $\mathcal{Q}$.

Another reason we introduce the partition is to generate a group of thresholds to filter objects that are unlikely to serve as o-NN for any core point. For instance, let c_u be a core point within $\mathcal{C}$, o be a newly arrived object. If $t_m^u(\theta) < D(c_u, o)$, o cannot become o-NN of c_u for a long time. Therefore, we could support DGNNQ via maintaining a group of thresholds, i.e., $\{t_m^1(\theta), t_m^2(\theta), \cdots, t_m^k(\theta)\}$. In other words, if the distance between o and $\{c_1, c_2, \cdots, c_k\}$ is larger than $\{t_m^1(\theta), t_m^2(\theta), \cdots, t_m^k(\theta)\}$, it is unlikely to be the o-NN of any core point within $\mathcal{C}$ before it expires from the window, we could ignore it temporarily.

We now formally explain the PM-Tree. It organizes these core points based on M-Tree. In order to provide PM-Tree with stronger pruning ability, for each node e within $\mathcal{I}$, we record the maximal $t_m^i(\theta)$ among all elements within e. In detail, the query process usually involves distance calculation and range screening. By maintaining the maximum $t_m^i(\theta)$, it is possible to prevent underestimation of the node coverage, ensuring correct pruning. Furthermore, this approach helps avoid missing critical core points, guaranteeing the completeness of the query results. Additionally, storing the maximum value accelerates the determination of whether a node needs to be accessed, reducing the number of child node visits and improving query efficiency.

4 The Incremental Algorithms

First, we explain how to handle newly arrived and expired objects. Next, we discuss how to update PM-Tree when query points are updated.

4.1 Object-Based Incremental Maintenance Algorithm

As the window slides, a set of newly arrived objects flow into the window, while another set of objects expire from the window. We first update the partition. Additionally, we update $t_m^u(\theta, o)$ for each c_u if needed. Finally, we further update the index PM-Tree.

Before formally introducing the algorithm, we first explain the concept of *skyline* o-NN. Let c be a core point, and o_1, o_2 be two objects within the query window. If o_1 arrives later than o_2 and $D(c, o_1) < D(c, o_2)$, we say that o_1 dominates o_2 as stated in observation 2. If o_1 is not dominated by any other object within the query window, o_1 is regarded as a *skyline* o-NN for c.

The Partition Incremental Maintenance. We first insert newly arrived objects into the last partition P_m, while expired objects are deleted from the

Algorithm 1: Incremental Maintenance Algorithm

Input: Streaming Data $\mathcal{S}$, PM-Tree $\mathcal{I}$, New Query Points $\mathcal{Q}$
Output: k nearest neighbor objects

```
1  for each new object o ∈ S do
2      insertion(P_{m-1}, o);
3      if |P_{m-1}| == |W|/2 then
4          form a new partition P_m;
5      search(I, o);
6      if D(e_0(p), o) ≤ e_0(r) + e_0(θ) then
7          for each core point c_u ∈ C where o is o-NN do
8              update c_u(o, θ);
9  for each expired object o' do
10     deletion(P_0, o');
11     if P_0 empty then
12         P_1 ← P_0;
13     for c_u ∈ C where t_i^u(θ) invalid do
14         scan(P_1) and update skyline o-NN set t_i^u(S);
15 for q' ∈ C_i is deleted do
16     remove q' from C_i;
17     if C_i is empty then
18         delete C_i and its core point c_i;
19 for each query q ∈ Q do
20     if q fits in an existing ball then
21         add q to the ball;
22     else
23         create a new ball b(q, r_new);
24 if k' < k/2 then
25     split q-subsets until k' = k;
26 if k' > 2k then
27     merge q-subsets until k' = k;
28 return k' > k ? select k objects : extend o-NN with o-2NNs to meet k objects;
```

first partition P_0. When the first partition P_0 turns to empty, the next partition P_1 is treated as the first partition. Since P_1 contains only one slide, we do not further access objects within it. When the last partition P_{m-1} turns to full, we form a new partition P_m. With the sliding of the window, when a partition P_{i-1} turns to empty, P_i turns to the first partition. As P_i contains more than one slides, for each c_u, if $t_i^u(\theta)$ is larger than the smallest value among $\min(u, i+1, m, \theta)$, it means all objects within P_i are not o-NN of c_u before they expire the window. Otherwise, we should re-evaluate objects within P_i(See lines 1-14). Here, $\min(u, i+1, m, \theta)$ refers to the minimal value among $\{t_{i+1}^u(\theta), t_{i+2}^u(\theta), \cdots, t_m^u(\theta)\}$.

Based on the above discussion, we understand that, after P_{i-1} turns to empty, if each c_u satisfies $t_i^u(\theta) \geq \min(u, i+1, m, \theta)$, we could ignore P_i. Otherwise, we

Table 1. Parameter Settings.

Parameter	value
N	100KB, 300KB, **500KB**, 1MB,2MB
$\|\mathcal{Q}\|$	500, **1000**, 1500, 2000, 2500
s	2%, 4%, **6%**, 8%, 10% ($\times$ N)
k	100, 150, **200**, 250, 300
U_o/U_q	1, 10, **50**, 100, 200

re-partition P_i, form the corresponding P-SKETCH accordingly. In order to avoid repeatedly access objects within P_i, after the re-partitioning, we maintain all *skyline* o-NNs for each core point. For example, given the core point c_u, if it has some *skyline* o-NNs within P_i, we maintain them in the set $t_i^u(S)$.

Back to the example in Fig. 3(a), when the window slides from W_0 to W_4, partitions P_0, P_1, and P_2 expire, P_3 becomes the first partition. Since P_3 contains 4 slides, we check whether we should re-partition P_3. As $t_3^2(\theta) \geq \min(u, 4, 5, \theta)$, we re-partition P_3 into $\{P_3^0, P_3^1, P_3^2\}$, and find all *skyline* o-NNs for c_2.

The PM-Tree Incremental Maintenance. For each newly arrived object o within the new slide s_n, we search on PM-Tree, evaluate whether it has chance to become a candidate o-NN. To be more specifically, we first access the root e_0 of the index $\mathcal{I}$. If $\mathrm{D}(e_0(p), o)$ is larger than $e_0(r) + e_0(\theta)$, it means o cannot become an o-NN of any core point before it the last partition P_m becomes the first partition. In other words, it is unlikely to be the o-NN of any core point within $\mathcal{C}$ before it expires from the window, we could ignore it temporarily. Otherwise, we should further access child nodes of e_0 following the logic discussed before. After search, for each core point c_u that regards o as its o-NN within P_m, we update $c_u(o, \theta)$ accordingly. Furthermore, for each node e that contains c_u, if $e(\theta)$ is set based on the original $c_u(\theta)$, we also should update it(See lines 5-8).

4.2 Incremental Maintenance Algorithm for Query Points

In this section, we explain the incremental algorithms about processing query points updating. Let $\mathcal{C}\{C_1, C_2, \cdots, C_{k'}\}$ the query-based partition result, and query points within them be bounded by a group of balls $\{b(c_1, r_1), b(c_2, r_2), \cdots, b(c_k, r_{k'})\}$, i.e., $\{c_1, c_2, \cdots, c_{k'}\}$ are core points, and $\{r_1, r_2, \cdots, r_{k'}\}$ are the corresponding radius. Note, as will be reviewed later, k' may be different from k.

When a query q within C_i is deleted, we remove it from C_i. After removing, if C_i turns to empty, we further delete C_i, as well as the tuple corresponding to c_i(See lines 15-18). At that moment, if the number of core points k' is within $[\frac{k}{2}, 2k]$, we do not further find new core points, as the running cost of searching o-NN of a core point in each partition is high.

When a new query q_{new} is submitted, we first evaluate whether existing a ball $b(c_i, r_i)$ contains q_{new}. If the answer is yes, we randomly select one ball, insert q_{new} into this ball. If no ball contains q_{new}, we regard q_{new} as a new core point. Additionally, we form a new ball $b(r_{new}, q_{new})$ for q_{new}, with r_{new}

being the minimal value among $\{r_1, r_2, \cdots, r_{k'}\}$. Similarly, after forming, if the number of core points is within $[\frac{k}{2}, 2k]$, we do not apply the merge operation(See lines 19-23). As the logic is simple, we skip the details. In particularly, if k' is smaller than $\frac{k}{2}$, we should split some q-subsets. If k' is larger than $2k$, we should merge some q-subsets. Besides, if $k \neq k'$, we cannot use o-NN of each core point for answering DGNNQ directly(See lines 24-28). In the following, we will explain the corresponding algorithms.

The q-Partition Split. Let $\mathcal{C}\{C_1, C_2, \cdots, C_{k'}\}$ be bounded by a group of balls $\{b(c_1, r_1), b(c_2, r_2), \cdots, b(c_k, r_{k'})\}$. We first find the ball $b(c_i, r_i)$ with maximal radius, and then partition query points within $b(c_i, r_i)$ into two balls as the manner of M-Tree-based splitting. From then on, we repeat the above operations until k' achieves to k. Finally, we re-form the P-SKETCH for the new core points. Compared with forming P-SKETCH based on objects within the window, we apply a different way. It is based on the following observation. That is, given two new core points c_i', c_i'' that are generated based on the original core point c_i, the distance among c_i, c_i' , and c_i'' is small, objects within P-SKETCH are also near to both c_i' and c_i''. Therefore, we only use objects within the original P-SKETCH of each core points to form new P-SKETCHs. In this way, the running cost could be reduced a lot.

The q-Partition Merge. Let $\mathcal{C}\{C_1, C_2, \cdots, C_{k'}\}$ be bounded by a group of balls $\{b(c_1, r_1), b(c_2, r_2), \cdots, b(c_k, r_{k'})\}$. We apply the algorithm discussed in Section 3.1 to merge $\mathcal{C}\{C_1, C_2, \cdots, C_{k'}\}$ into k q-subsets. Similar with the logic discussed before, we use objects within the original P-SKETCH of each core points to form new P-SKETCHs. For the limitation of space, we skip the details.

The Query Algorithm. When the window slides, we should return a group of objects to the system. If $k' = k$, we return o-NN of each core point directly. If $k' > k$, as we maintain more than k core points, they may correspond to more than k objects. At that moment, we select k objects from o-NNs of core points as query result objects. To be more specially, let $\{c_1, c_2, \cdots, c_{k'}\}$ be the set of core points, and $\{o_1, o_2, \cdots, o_{k'}\}$ be o-NNs of these core points. We apply the algorithm discussed in Section 3.1 to partition $\{o_1, o_2, \cdots, o_{k'}\}$ into k subsets $\{O_1, O_2, \cdots, O_k\}$, and then randomly select one object in each subset as query result object. If $k' < k$, as we maintain less than k core points, they may correspond to less than k objects, we should return more objects. At that moment, we select query result objects from both o-NN and o-2NN of each core point following the logic discussed before.

5 Performance Evaluation

In this section, we conducted extensive experiments to demonstrate the efficiency of the Q2WP. Next, we will first describe the experimental settings and then present our research findings.

5.1 Experiment Settings

Data sets. In our experiments, we utilized four datasets: two real-world datasets, TRIP and BEIJING, as well as two synthetic datasets, SYN-R and SYN-U. The TRIP dataset comprises 1.6 GB of trip records from New York City, collected over a period of 72 months. After cleaning the data, we retained four attributes: taxi ID, pick-up time, drop-off time, and travel distance. The cleaned records were then sorted in ascending order by pick-up time. The BEIJING dataset is derived from the Microsoft T-Drive project, which includes GPS trajectories of 10,357 taxis recorded between February 2 and February 8, 2008, in Beijing. This dataset contains approximately 15 million points, covering a total trajectory distance of about 9 million kilometers. After cleaning (resulting in a dataset size of 1 GB), we retained three key attributes: taxi ID, longitude, and latitude. For the synthetic datasets, SYN-R simulates objects following a normal distribution, whereas SYN-U simulates objects following a uniform distribution. They contain 100 million real numbers, respectively.

Parameters Settings. We evaluated the performance of different algorithms under five parameters, including N, $|\mathcal{Q}|$, s, k, and U_o/U_q. Here, N denotes the window size, $|\mathcal{Q}|$ represents the number of query points, s refers to the flow rate of the sliding window, k refers to the number of groups of query points, and U_o/U_q represents the ratio of object updates to query updates. For each parameter evaluation, we constructed five query workloads. When analyzing the impact of a specific parameter, the remaining four parameters were set to their default values, as shown in Table 1, with the default values highlighted in bold.

Competitors. In addition to our algorithm, we implemented the EHC [4], SHR [4], and KMPT[13] methods as competing approaches to address the DGNNQ. All algorithms were implemented in C++ and tested on a 6226R CPU @ 2.90GHz with a 16×2 core processor configuration.

5.2 Performance Comparison

Comparison of data throughput under different algorithms. We first compare the data throughput of four algorithms with different window sizes, as shown in Fig.4(a)-(d). Our findings are summarized as follows: Firstly, PM-Tree achieves the highest data throughput, approximately 1.5 times that of KMPT and 100 times that of SHR and EHC. Secondly, the data throughput of all four algorithms decreases as the window size increases. However, PM-Tree exhibits the smallest reduction, maintaining its performance advantage. This is due to its efficiency in avoiding the computational burden of calculating distances for each growing object. In contrast, KMPT incurs high computational costs due to the need to determine nearest neighbor candidates for each partition, while EHC must access all data points every time the window slides. Although SHR improves on EHC through pruning, it still struggles with unnecessary data access due to inaccurate thresholds for virtual center points. In summary, PM-Tree is the most efficient and robust, with minimal performance impact from data scale.

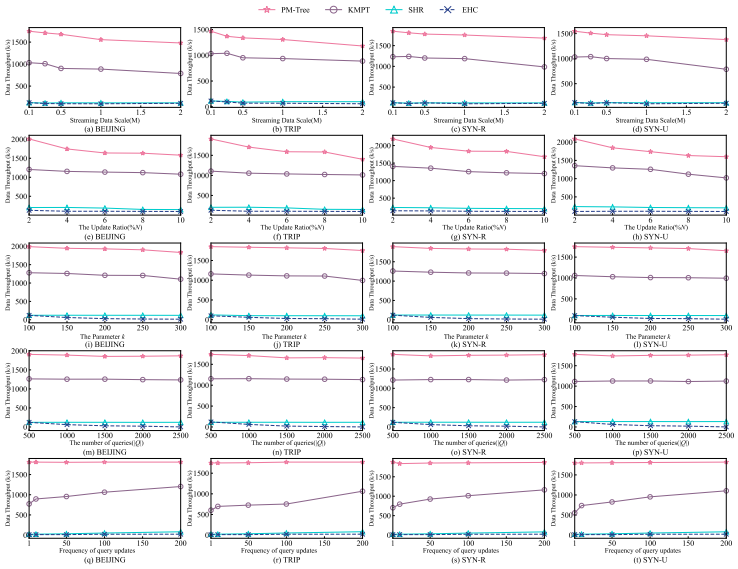


Fig. 4. Data throughput comparison of different algorithms.

Next, we compare the data throughput of the four algorithms under different values of s. As shown in Fig. 4(e)-(h), the throughput of both PM-Tree and KMPT decreases as s increases, whereas the performance of SHR and EHC remains relatively stable. This disparity arises because a larger s requires PM-Tree and KMPT to maintain distances with a greater number of points, thereby increasing their computational overhead. Nevertheless, PM-Tree consistently outperforms KMPT. Given that s is typically not large in most practical applications, PM-Tree remains the most efficient algorithm in the majority of cases.

Thirdly, we assess the impact of the parameter k (ranging from 100 to 300) on algorithm performance. As shown in Fig. 4(i)-(l), the data throughput of all four algorithms exhibits minimal variation with increasing k. This stability can be attributed to the tree-based indexing methods, which effectively filter out irrelevant data and accelerate processing. Among the algorithms, PM-Tree demonstrates significantly higher throughput compared to the other three.

Then, we evaluate the influence of $|\mathcal{Q}|$ on algorithm performance. As illustrated in Fig. 4(m)-(p), the data throughput of the four algorithms changes only slightly with varying $|\mathcal{Q}|$. Both PM-Tree and KMPT employ grouping strategies for query points, relying on core points within the groups to find nearest neighbors rather than computing distances for every query point. This approach makes them more efficient, with PM-Tree being approximately 1.6 times faster than KMPT. In contrast, SHR and EHC compute the nearest neighbors for more query points individually, leading to lower data throughput.

Finally, we analyze the performance of PM-Tree, KMPT, SHR, and EHC under varying frequencies of query updates, keeping other parameters at their default settings. As depicted in Fig. 4(q)-(t), PM-Tree consistently achieves the highest

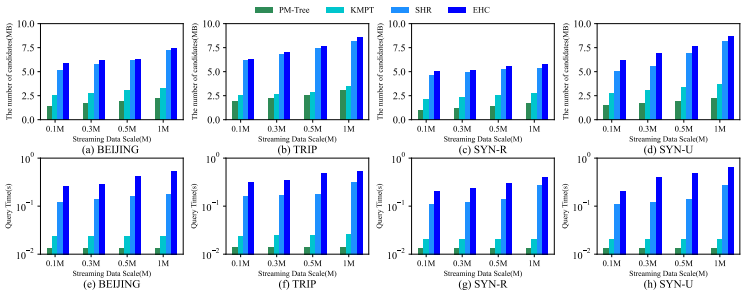


Fig. 5. Comparison of different algorithms under other metrics.

data throughput and demonstrates robustness to changes in update frequency. This robustness is attributed to its efficient q-Partition Merge and q-Partition Split algorithms, which quickly handle new query points upon arrival, avoiding the overhead of frequent reclustering or recalculating neighboring points. In contrast, KMPT is more sensitive to update frequency, as it requires re-clustering for each query point update. Meanwhile, SHR and EHC also experience performance degradation because they must recompute the NNs for the updated query points.

Space cost comparison of different algorithms. We evaluate the space cost of algorithms on various datasets. As shown in Fig.5(a)-(d), the PM-Tree consistently consumes the least memory across all datasets. This is primarily due to the characteristics of its index structure PM-Tree, which requires maintaining only a small subset of objects as candidate results. In contrast, the other three algorithms need to store more objects to support queries, leading to significantly higher space costs.

Query time comparison of different algorithms. As illustrated in Fig. 5(e)-(h), the PM-Tree demonstrates the lowest computational cost for query processing. This efficiency is primarily due to the PM-Tree's ability to directly retrieve o-NNs for core points when $k' = k$ without additional computation. Even when $k' \neq k$, its partitioning strategy ensures efficient selection of query results by minimizing redundant calculations, leveraging the o-NN and o-2NN sets, enabling rapid and effective resolution of the DGNNQ problem.

6 Conclusion

This paper introduces Q2WP, a novel framework designed to efficiently support DGNNQ over streaming data. The framework leverages the NNP algorithm to partition nearby query points into k subsets, creating a virtual point for each subset. These virtual points are then effectively managed using the PM-Tree, a new indexing structure that evaluates which objects have the potential to become query results. Extensive experiments conducted on diverse datasets demonstrate the proposed framework's effectiveness and efficiency in supporting DGNNQ over streaming data.

ACKNOWLEDGMENT

The work is partially supported by the National Natural Science Foundation of China (Nos. 62472293,U23A20309,62102271), Rui Zhu and Chuanyu Zong are corresponding authors of this paper.

References

1. Bahri, M.: Effective weighted k-nearest neighbors for dynamic data streams. In: 2022 IEEE International Conference on Big Data (Big Data). pp. 3341–3347. IEEE (2022)
2. Chen, L., Zhong, Q., Xiao, X., Gao, Y., Jin, P., Jensen, C.S.: Price-and-time-aware dynamic ridesharing. In: 2018 IEEE 34th international conference on data engineering (ICDE). pp. 1061–1072. IEEE (2018)
3. Cui, N., Qian, K., Cai, T., Li, J., Yang, X., Cui, J., Zhong, H.: Towards multi-user, secure, and verifiable k nn query in cloud database. IEEE Transactions on Knowledge and Data Engineering (2023)
4. Deng, K., Sadiq, S., Zhou, X., Xu, H., Fung, G.P.C., Lu, Y.: On group nearest group query processing. IEEE Transactions on Knowledge and Data Engineering **24**(2), 295–308 (2010)
5. Ding, X., Chen, L., Gao, Y., Jensen, C.S., Bao, H.: Ultraman: A unified platform for big trajectory data management and analytics. Proceedings of the VLDB Endowment **11**(7), 787–799 (2018)
6. Lee, J.M.: Fast k-nearest neighbor searching in static objects. Wireless Personal Communications **93**(1), 147–160 (2017)
7. Miao, X., Gao, Y., Chen, G., Zheng, B., Cui, H.: Processing incomplete k nearest neighbor search. IEEE Transactions on Fuzzy Systems **24**(6), 1349–1363 (2016)
8. Papadias, D., Shen, Q., Tao, Y., Mouratidis, K.: Group nearest neighbor queries. In: Proceedings. 20th International Conference on Data Engineering. pp. 301–312. IEEE (2004)
9. Xu, H., Lu, Y., Li, Z.: Continuous group nearest group query on moving objects. In: 2010 Second International Workshop on Education Technology and Computer Science. vol. 1, pp. 350–353. IEEE (2010)
10. Yang, K., Tian, C., Xian, H., Tian, W., Zhang, Y.: Query on the cloud: improved privacy-preserving k-nearest neighbor classification over the outsourced database. World Wide Web pp. 1–28 (2022)
11. Yang, R., Niu, B.: Continuous k nearest neighbor queries over large-scale spatial-textual data streams. ISPRS Int. J. Geo Inf. **9**(11), 694 (2020)
12. Yiying, J., Liping, Z., Feihu, J., Xiaohong, H.: Groups nearest neighbor query of mixed data in spatial database. Journal of Frontiers of Computer Science & Technology **16**(2), 348 (2022)
13. Zhu, R., Li, C., Zhang, A., Zong, C., Xia, X.: Continuous group nearest group search over streaming data. In: Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data. pp. 80–95. Springer (2023)
14. Zhu, R., Wang, B., Yang, X., Zheng, B.: Closest pairs search over data stream. Proc. ACM Manag. Data **1**(3), 205:1–205:26 (2023)