6-2024

Closest pairs search over data stream

Rui Zhu ZHU

Bin WANG

Xiaochun YANG

Baihua ZHENG
Singapore Management University, bhzheng@smu.edu.sg

Citation

ZHU, Rui Zhu; WANG, Bin; YANG, Xiaochun; and ZHENG, Baihua. Closest pairs search over data stream. (2024). *Proceedings of the ACM on Management of Data, Santiago, Chile, June 9-15*. 1, 1-26.
Available at: https://ink.library.smu.edu.sg/sis_research/9097

CVoR Issued. April 15, 2024. DOI: https://doi.org/10.1145/3617326

Closest Pairs Search Over Data Stream

RUI ZHU, Shenyang Aerospace University, China
BIN WANG, Northeastern University, China
XIAOCHUN YANG, Northeastern University, China
BAIHUA ZHENG, Singapore Management University, Singapore

k-closest pair (KCP for short) search is a fundamental problem in database research. Given a set of d-dimensional streaming data $\mathcal{S}$, KCP search aims to retrieve k pairs with the shortest distances between them. While existing works have studied continuous 1-closest pair query (i.e., $k = 1$) over dynamic data environments, which allow for object insertions/deletions, they require high computational costs and cannot easily support KCP search with $k > 1$. This paper investigates the problem of KCP search over data stream, aiming to incrementally maintain as few pairs as possible to support KCP search with arbitrarily k. To achieve this, we introduce the concept of NNS (short for Nearest Neighbour pair-Set), which consists of all the nearest neighbour pairs and allows us to support KCP search via only accessing $O(k)$ objects. We further observe that in most cases, we only need to use a small portion of NNS to answer KCP search as typically $k \ll n$. Based on this observation, we propose TNNS (short for Threshold-based NN pair Set), which contains a small number of high-quality NN pairs, and a partition named τ-DLBP (short for τ-Distance Lower-Bound based Partition) to organize objects, with τ being an integer significantly smaller than n. τ-DLBP organizes objects using up to $O(\log \frac{n}{\tau})$ partitions and is able to support the construction and update of TNNS efficiently.

CCS Concepts: • **Information systems → Multidimensional range search**.

Additional Key Words and Phrases: Streaming Data, k-Closest Pair Search, Partition, Cube

ACM Reference Format:
Rui Zhu, Bin Wang, Xiaochun Yang, and Baihua Zheng. 2023. Closest Pairs Search Over Data Stream. *Proc. ACM Manag. Data* 1, 3 (SIGMOD), Article 205 (September 2023), 26 pages. https://doi.org/10.1145/3617326

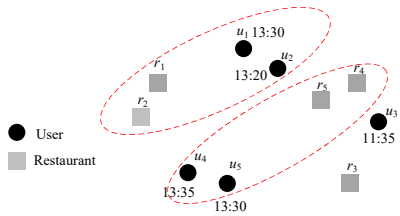
1 INTRODUCTION

The focus of this paper is the problem of finding k-closest pair (KCP search) [1][2][3][4] over data stream. Specifically, we consider a dynamic setup where objects are from a d-dimensional streaming dataset. When $k = 1$, the goal is to find a pair of objects with the smallest distance among the given n objects. However, we extend this problem to the more general case where $k > 1$.

In a d-dimensional streaming dataset $\mathcal{S}$, every two objects o_i and o_j construct a pair (o_i, o_j). The goal of KCP search is to retrieve the k pairs with the smallest scores among all pairs constructed by objects in $\mathcal{S}$. In this paper, we use the Euclidean distance as the scoring function for explanation purposes. However, the techniques proposed in this paper are applicable to many common distance functions over d-dimensional space, including Manhattan, Chebyshev, and Minkowski distances.

Authors' addresses: Rui Zhu, Shenyang Aerospace University, No.37 Daoyi South Avenue, Daoyi District, Shen Yang, China, zhurui@sau.edu.cn; Bin Wang, Northeastern University, NO. 3-11, Wenhua Road, Heping District, Shen Yang, China, binwang@neu.edu.cn; Xiaochun Yang, Northeastern University, NO. 3-11, Wenhua Road, Heping District, Shen Yang, China, yangxc@mail.neu.edu.cn; Baihua Zheng, Singapore Management University, 80 Stamford Road, Singapore, bhzheng@smu.edu.sg.

2836-6573/2023/9-ART205 $15.00
https://doi.org/10.1145/3617326



bookings	tuples
b_1	<u_1, r_1,13:30>
b_2	<u_2, r_2,13:20>
b_3	<u_3, r_3,11:35>
b_4	<u_4, r_4,13:35>
b_5	<u_5, r_5,13:30>

Fig. 1. Supporting Food Delivery Services via KCP(k=2)

To investigate this problem, we consider a general case, where objects in $\mathcal{S}$ can be inserted and deleted and their arrival order may differ from their expiry order.

KCP search is a fundamental problem that has been studied for over thirty years [5–7]. It presents a challenge due to the potentially large number of pairs that need to be considered. For example, given a dataset $\mathcal{S}$ of n objects, there are $O(n^2)$ pairs in total. The dynamic nature of data streams further complicates KCP search. A single update can affect all objects in $\mathcal{S}$, triggering the construction of $O(n)$ pairs in the case of an insertion or the expiry of $O(n)$ pairs in the case of a deletion.

KCP search is a building block for a wide range of data mining tasks [8], including clustering [9, 10], outlier detection, and more. For instance, clustering algorithms like C2P [11] uses KCP search as a primitive operation, and the algorithm ClusTree [12] maintains clusters incrementally by finding closest pairs under data stream.

KCP search also has many practical applications, such as supporting food delivery services by finding similar bookings for assignments. In this scenario, each booking request can be represented as $b\langle d, r, t\rangle$, where d and r denote the delivery address and restaurant location, respectively, and t is the requested delivery time. KCP search can efficiently identify highly similar bookings that have nearby restaurants and delivery addresses and a small time difference between their requested delivery times, which shall be assigned to the same delivery person.

Consider an example illustrated in Fig 1, where the system receives 5 booking requests (b_1, b_2, b_3, b_4, and b_5). Among these bookings, b_1 and b_2 have nearby delivery addresses and restaurant positions, with a small difference in their requested delivery times. Similarly, b_4 and b_5 also share these characteristics. To optimize resource allocation, the system employs a KCP search ($k = 2$) to identify booking pairs with similar properties. As a result, it identifies pairs (b_1, b_2) and (b_4, b_5) due to their proximity in delivery locations and restaurants, as well as the small time difference between their requested delivery times. By assigning a single delivery person to handle both bookings in each pair, the system significantly improves efficiency and cost-effectiveness of the delivery process.

Compared to traditional route-based and range-based methods, KCP search can provide ideal similar bookings. The former assesses the suitability of a booking pair by determining the length of the common sub-route, which is far more complicated than KCP search. The latter uses range queries to form suitable booking pairs, where bookings located in the same query region are considered similar. However, it can be challenging to capture the real-time distribution of the restaurants and delivery addresses, which makes it difficult to find a suitable query radius for each range query.

In a product assembly line, identical small parts are used to build complete products, but production errors can result in variations in the sizes of different parts. Small parts continuously arrive in the product line and expire when they are selected for completing products. KCP search can efficiently select small parts with similar sizes, such as four wheels for a car, two eyes for a toy doll, two or three single-connector sockets for a multi-connector socket. In event-based social networks [13], KCP search provides a simple way to find "event-partner" by considering a small

number of users' attributes, while in stock recommendation systems, it can find stocks that share similar prices and volumes in the last five minutes. Overall, KCP search is a powerful tool that can provide simple and effective solutions to many practical problems in various domains.

To the best of our knowledge, none of the existing approaches can handle KCP search over data stream. The closest work is the study of continuous 1CP ($k = 1$) queries over a dynamic data environment, where object insertion and deletion are allowed. Based on the ways objects are organized, the main efforts in continuous 1CP query over dynamic data can be categorized into two groups, *tree-based* and *partition-based* approaches. The former [5][7] uses a tree-based index to maintain objects and monitors the nearest neighbour (NN) of every object. When objects are inserted into or deleted from the dataset, both the index and the NN of related objects are updated. However, the cost of maintaining the NN of every object is very high. The latter [6] uses random algorithms to partition objects into subsets based on distances (or distance lower-bound) between objects and their NNs. Compared to tree-based approaches, they avoid maintaining the NN of every object. However, the cost of processing an insertion or a deletion is $O(n)$ in the worst case.

Moreover, the computational complexity of both types of algorithms is very high, making it challenging to extend them for $k > 1$ in a data streaming setting. Firstly, the algorithms need to find result pairs from a massive set of pairs rather than just the NN pairs constructed by objects and their NNs. Secondly, the frequency of object insertions and deletions under data stream is much higher than that under dynamic data environments, making it difficult to process the influx of newly arrived/expired objects in real-time. Thus, it is crucial to develop an approach that can quickly process KCP search and meanwhile support efficient update under data stream.

Solution Overview. Our solution is based on the following observations. Given an object $o \in \mathcal{S}$, if its NN pair is not part of the answer set to a KCP, then none of the pairs containing o will be in the answer set either. Furthermore, for a given KCP search, we can find the result pairs via only considering distances between objects that form k NN pairs with the shortest distances. Using this insight, we construct the set NNS (short for Nearest Neighbour pair Set), which consists of all NN pairs. Our KCP search algorithm developed based on NNS is able to support KCP search at a cost of $O(k^2)$. We also introduce a new index called QC-Tree to facilitate the construction and update of NNS. QC-Tree, similar to quad-tree, has a bounded height of $O(\log n)$ regardless of the distribution of objects. In addition, the leaf nodes in QC-Tree reflect the upper-bounds of distances between objects and their NNs, which enables efficient NN search based on prior known search ranges.

However, maintaining NNS incurs a high computational cost, as the incremental cost of updating every object's NN is $O(n)$ per insertion/deletion in the worst cases, which is too expensive under data stream. We have observed that in most cases, a small set of NNS suffices to answer KCP search as typically $k \ll n$. This observation has led us to consider maintaining a subset of "high quality" NN pairs, i.e., pairs with the shortest distances, to answer KCP search. Therefore, we propose TNNS (short for Threshold-based NN pair Set), which contains only a small number of NN pairs, guided by a controlling parameter τ.

To incrementally maintain TNNS under data stream, we propose a novel partition named τ-DLBP (short for τ-Distance Lower-Bound based Partition). This approach organizes objects using up to $O(\log \frac{n}{\tau})$ partitions. τ-DLBP uses one partition to maintain objects that contribute to TNNS, and uses other partitions to organize the rest of objects based on the lower bounds of their distances to their NNs. As compared with QC-Tree, τ-DLBP significantly reduces the update cost to $O((d+3^d) \log \frac{n}{\tau}+\tau)$ per update. Additionally, τ-DLBP organizes objects based on their likelihood of contributing to KCP search with smaller partitions containing objects that have shorter distances to their NNs and tighter distance lower bounds. The size of partitions increases exponentially with the increase of their distance lower bounds. When a KCP search is submitted, if k is small, we only need to access

Table 1. Frequent Notations

Notation	Definition
$\mathcal{S}$	a d-dimensional streaming dataset with n objects
$NN(o)$	the nearest neighbor of an object o
$RNN(o)$	the reverse nearest neighbor(s) of an object o
D(o)	the distance between o and its nearest neighbor, also termed as the *score* of object o
$q(o, r, S')$	a range search to locate objects in S' that are within r distance to o
(o, o')	a pair formed by two objects o and o'

the partition containing TNNS to perform the search directly. Even if k is large, we can retrieve query result pairs via accessing objects located in only a few partitions. This way, τ-DLBP reduces the computational cost of incrementally maintaining TNNS under data stream while still being able to support KCP search.

2 RELATED WORKS

This section mainly reviews algorithms about KCP search [1, 7, 14] and threshold-based continuous spatial queries[15].

2.1 KCP Search

KCP search is a well-known problem that has been extensively studied in previous research across various data dimensions and environments [16–20], such as high-dimensional data spaces [21, 22], event-based social networks [13], and moving-object databases [14]. In this study, we will focus on relevant existing research, including three main types of algorithms: tree-based, partition-based, and other types. To facilitate understanding, we provide a list of symbols used throughout this paper in Table 1.

Tree-based Algorithms. Many tree-based approaches, such as those proposed in [5, 23, 24], support 1CP search by maintaining a tree-based structure that stores the nearest neighbour (NN) of each object. These structures update both the index and NNs of relevant objects when objects are inserted or deleted from the dataset. For example, the structure proposed in [23] uses $O(n \log^d n)$ space and runs in $O(\log \log n \log^d n)$ amortized time per update to support 1CP search. In contrast, C-Box, the first deterministic data structure presented in [5], maintains 1CP in $O(\log n)$ time per update by only maintaining mutual nearest neighbour (MNN) pairs. Note, an object pair (o, o') is considered as MNN, if they are the nearest neighbor to each other, i.e., $o = NN(o') \wedge o' = NN(o)$. However, it has a large hidden constant in its computational complexity. The cost of processing one insertion or deletion is $O(dN_d \log d \log n)$ and $O(dN_d M_d \log d \log n)$ respectively, where $N_d = (2(s+2)(s+1)d+1)^d$, $M_d = (s(s+1)^2(d+0.5)+1)^d$, and s is a parameter used for fairly splitting the space (e.g., $N_d \geq 2401$ and $M_d \geq 8281$ when $d = 2$).

Other works, such as those presented in [4] and [25] study how to find the k-closest pairs between two spatial data sets. [25] also investigates the KCP search over one spatial database. It uses R-Tree to maintain spatial objects. The corresponding algorithms do not consider how to process newly arrived/expired objects. [8] examines the problem of KCP search under a metric space and uses M-Tree-based index [26, 27] to maintain all NN pairs. They obverse that given any query result pair (o_i, o_j), scores of pairs $(o_i, o_i.NN)$ and $(o_j, o_j.NN)$ are not larger than the score of the top-$2k$ NN pair.

Partition-based Algorithms. Golin et al [6] propose a partition-based algorithm named Random to support continuous 1CP search, based on a randomized data structure. The algorithm initializes

with $S_1 = \mathcal{S}$, where $\mathcal{S}$ is the set of objects. It then randomly selects an object $o \in S_1$ as the pivot and forms the set S_1' based on $\mathrm{D}(o)$, following a group of partition rules. Next, it sets S_2 to $(S_1 - S_1')$ and partitions S_2 using the same logic used previously to partition S_1. This process is repeated m times until the subset S_m has only two objects. When an object is inserted into $\mathcal{S}$, the algorithm evaluates whether it contributes to 1CP by accessing these m partitions, with a cost bounded by $O(3^d m)$. In ideal cases, m is excepted to be $O(\log n)$, and the cost of processing an object is $O(3^d \log n)$. However, the performance of the algorithm is heavily dependent on the *pivot* objects selected. In the worst-case scenario, m might approach n, and the running time for one insertion/deletion is $O(n)$. Furthermore, a parallel batch-dynamic data structure for 1CP search is proposed in [28], which supports batches of insertions and deletions in parallel using the data structure proposed in [6].

Other Algorithms. Instead of maintaining NN/MNN of objects, a buffer-based algorithm [1] tracks only a small subset of n' (where $n' < n$) pairs with the lowest scores. In [29], the domination relationships among pairs, which only exist in the sliding window model, are used to support KCP search over sliding windows. However, this property is not available in the non-sliding window model. [30] proposes a unified framework for answering KCP search. It utilizes inverted-list-based index for maintaining streaming data, applies a TA-based algorithm for NN search, and finally supports KCP search via maintaining all NN pairs. However, the cost of NN searches over inverted-lists is high, and this approach is not efficient for data streams. Another algorithm, based on "reference points", is proposed in [3, 31]. It selects a collection C of random reference points, calculates distances between object points and these reference points, and uses triangular inequality to reduce search scale. However, in many cases, the algorithm cannot find suitable reference points, especially under data stream where the distribution of streaming data changes timely. Thus, the running cost of processing a newly arrived object in the worst cases is $O(n)$.

Discussion. Tree-based algorithms must maintain all NN/MNN pairs, resulting in high computational costs. Partition-based algorithms can be unstable as their performance relies heavily on the quality of the pivots. In the worst cases, the cost of processing one object can be linear to the size of the dataset. Other methods discussed above are also not efficient for supporting KCP search under data streams. Thus, a more efficient algorithm is needed to provide stable and effective performance for KCP search over data streams.

2.2 Threshold-based Continuous Spatial Query

Continuous spatial query is a type of search that continuously monitors spatial data points around a specific query point [32]. The query runs continuously and updates in real time as both the query point and data points change their positions over time. It has been undergone extensive research in various environments, such as spatial database [33], on-air broadcasting [34–36], data streaming environments [37], and scenarios involving obstacles [38–41].

Threshold-based continuous spatial query is a specific type of continuous spatial query that determines relevant data points using a threshold [42]. The threshold defines a safe region around the query point, and data points that fall outside this safe region are disregarded. This approach is motivated by the spatio-temporal correlation between query and data objects, as the differences in positions between two adjacent timestamps for a point are typically not significant. It is efficient as it minimizes the number of data points that must be processed.

Several threshold-based continuous spatial query algorithms have been developed, such as the RIS-kNN algorithm [43] that maintains safe regions by considering the moving directions of the query object, the V*-diagram algorithm [44] that exploits the current knowledge of the query point for increasing computational efficiency, and the SRB algorithm [45] that computes rectangular safe

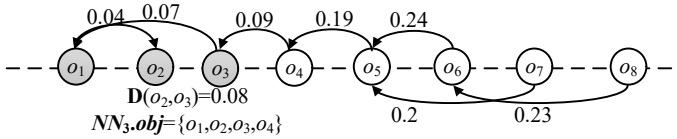


Fig. 2. 3-Closest Pair Search under NNS

regions incrementally for multiple queries. Li et al. have also developed an algorithm in [46] that implicitly computes safe regions by identifying a set of safeguarding objects that can provide tight safe regions and are efficient to compute.

It is worth noting that these queries differ from KCP search, which focuses on finding data pairs with small distances instead of data points near a query point. Additionally, in our paper, the position of streaming data remains unchanged until it expires from $\mathcal{S}$, and there is no spatio-temporal correlation between inserted and expired objects. As a result, techniques used in threshold-based continuous spatial query algorithms cannot be applied to support KCP search.

3 SUPPORTING KCP VIA NN PAIR SET

As mentioned in Section 1, NN pairs are essential in KCP search. In this section, we formally introduce the concept of NNS (short for Nearest Neighbour pair Set), explain how NNS can support KCP search, and present an index structure that can construct NNS for a given dataset and maintain NNS in a dynamic environment.

3.1 Supporting KCP Search Via NNS

Definition 1. **NNS**. *Given a streaming data set $\mathcal{S}$, NNS $\mathcal{N}$ ($\subset \mathcal{S} \times \mathcal{S}$) consists of all the NN pairs, i.e., $\mathcal{N} = \cup_{o \in \mathcal{S}}(o, NN(o))$.*

Lemma 1. *Given a streaming dataset $\mathcal{S}$, let NN_k be the set of top-k NN pairs having the shortest distances, $NN_k.obj$ denote the set of objects that form the pair(s) in NN_k, and R_k refer to the result set of a KCP search. It is guaranteed that $\forall(o, o') \in R_k$, $o \in NN_k.obj \wedge o' \in NN_k.obj$.*

Lemma 1's intuition is that, for any object o in $\mathcal{S}$, if its NN pair $(o, NN(o)) \notin NN_k$, any pair formed by o will not contribute to R_k. Thus, for any pair (o, o') in R_k, we can guarantee that $\{o, o'\} \subseteq NN_k.obj$. We can answer KCP search based on pairs formed by objects in NN_k, in total $O(k^2)$ pairs. Figure 2 illustrates this idea via the following example. We present the NN relationships between objects using a direct graph. For example, o_3's NN is o_1, so we construct an in-edge from o_3 to o_1 with a weight set to $\mathrm{D}(o_3)$=$\mathrm{D}(o_3, o_1)$= 0.07. Here, $\mathrm{D}(o_3)$ refers to the score of o_3, and $\mathrm{D}(o_1, o_3)$ refers to the distance between o_3 and o_1. Given $k = 3$, we have $NN_3 = \{(o_1, o_2), (o_1, o_3), (o_3, o_4)\}$. There are four objects in $NN_3.obj$, and they can form in total of six pairs: $\{(o_1, o_2), (o_1, o_3), (o_3, o_4), (o_1, o_4), (o_2, o_3), (o_2, o_4)\}$. The top-3 pairs with the shortest distances form the result set $R_3 = \{(o_1, o_2), (o_1, o_3), (o_2, o_3)\}$.

Accordingly, we propose a KCP Search Algorithm with its pseudo-code listed in Algorithm 1. Initially, the result set R is empty, and the set of candidate pairs Can is set to NN_k, i.e., the top-k NN pairs in $\mathcal{N}$ with the shortest distances (Line 1). The algorithm updates the result set R incrementally by locating the next result pair and updating Can until R contains k result pairs. In each iteration, the pair (o, o') in Can with the shortest distance is moved to R as the next result pair (Lines 3-4). For instance, when R is empty, the pair in Can (which is actually NN_k when R is empty) with the minimum distance is inserted into R as the result pair for the 1CP query. Then, the algorithm updates Can based on the candidate pairs newly introduced by the result pair (o, o') (Lines 5-9). For

Algorithm 1: The KCP Search Algorithm

Input: dataset $\mathcal{S}$, NNS $\mathcal{N}$, k
Output: Query Result Set R

```
1  R ← ∅, Can ← construct(N, k);
2  while |R| < k do
3      let (o, o') ∈ Can be the pair with the minimum distance;
4      Can ← Can − {(o, o')}, R ← R ∪ {(o, o')};
5      for each object o_j ∈ Can.obj do
6          let (a, b) ∈ Can be the pair with the max distance;
7          if D(o_j, o) < D(a, b) ∧ (o_j, o) ∉ (Can ∪ R) then
8              Can ← (Can − {(a, b)}) ∪ {(o_j, o)};
9      repeat Lines 6-8 to process o';
10 return R;
```

example, if (a, b) is the pair in Can with the maximal distance, and (o, o_j) is a new pair formed by o and another object o_j in $Can.obj$, we replace (a, b) in Can with (o, o_j) if $\mathrm{D}(o, o_j) < \mathrm{D}(a, b)$. Here, $Can.obj$ refers to the set of objects that form pairs in Can. The algorithm is terminated when $|R|$ reaches k with total running cost bounded by $O(k^2)$.

3.2 The Construction of NNS

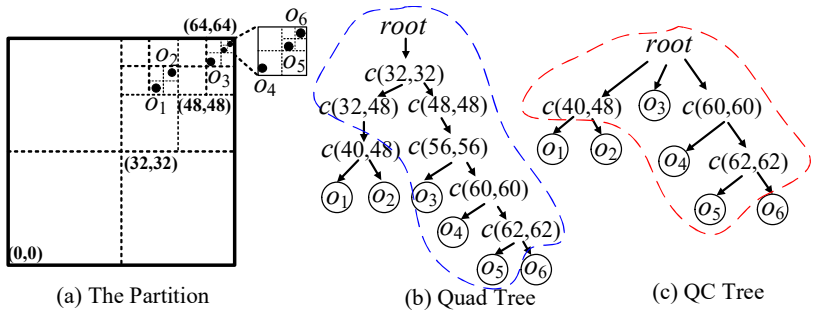
As mentioned earlier, NNS can significantly improve the processing of KCP search. In the following, we will explain how to construct NNS. Intuitively, objects in the streaming data set can be partitioned into different groups according to their coordinates in different dimensions, and two objects in adjacent partitions tend to be closer than objects in non-adjacent partitions. Therefore, we can reduce the cost of NN search by fully utilizing the partition result.

Equal Space Partition. One straightforward way to organize objects in a d-dimensional space $[0, 1]^d$ is by recursively partitioning the space into 2^d equal-sized hypercubes, and then using a quad-tree T to arrange the objects based on these partitions. In this paper, we use the term *hypercube* (in short *cube*) to refer to a d-dimensional cube that can be formed by partitioning the unit hypercube $[0, 1]^d$ into $(2^d)^i$ equal-sized subspaces. Each power of 2 hypercube has a side length of 2^{-i}, and its bottom-left coordinate $c[j]$ in the j-th dimension should be an integer multiple of 2^{-i}. We will use the term hypercube (or cube) throughout the rest of this paper, provided that the context is clear.

The partition process continues until every leaf node contains *only one* object. Each object in T corresponds to a hypercube $c_{(v_1,\ldots,v_d)}$, where $(v_1, \ldots, v_d)$ indicates the bottom-left coordinates of the standard hypercube. The side length of the hypercube is denoted by $|c_{(v_1,\ldots,v_d)}|$. For ease of presentation, we assume $d = 2$ in the following examples. However, our proposed algorithm can construct quad-trees in higher-dimensional spaces with arbitrary d. The quad-tree T constructed using equal space partition has the following property that can help find NNS efficiently.

Property 3.1. *Given a leaf node e in the quad-tree T that contains an object o, let $c_{(v_1,\ldots,v_d)}$ be its associated hypercube. The distance from o to its NN in $\mathcal{S}$ is bounded by $2\sqrt{d}|c_{(v_1,\ldots,v_d)}|$, i.e., $D(o) \leq 2\sqrt{d}|c_{(v_1,\ldots,v_d)}|$*[1].

[1]Property 3.1 has different forms under different distance functions, e.g., under Chebyshev Distance, the distance from o to its NN is bounded by $2|c_{(v_1,\ldots,v_d)}|$.



(a) The Partition (b) Quad Tree (c) QC Tree

Fig. 3. Quad-Tree and QC-Tree

Property 3.1 enables us to efficiently find objects' NN based on prior known search ranges centered at each object. This approach reduces the number of internal nodes that need to be accessed [47]. For ease of presentation, we adjust the coordinates from [0,1] to [0,64] for the running example in Section 3. Consider the set of six objects shown in Figure 3(a), located in the $[0, 64]^2$ space, where $o_1 = (42, 50)$, $o_2 = (46, 55)$, $o_3 = (56.1, 56.1)$, $o_4 = (60.1, 60.1)$, $o_5 = (62.1, 62.1)$, and $o_6 = (63.1, 63.1)$. Figure 3(b) shows the corresponding quad-tree formed by equal space partition. Let us use object o_6 as an example. The leaf node associated with $c_{(63,63)}$ contains o_6. According to the construction of quad-tree, we know that there is at least one object in the search range of $2\sqrt{d}|c_{(63,63)}|$ (i.e. $2\sqrt{2} \cdot 1 = 2\sqrt{2}$) centered at o_6. Accordingly, we can submit a range query with a radius $2\sqrt{2}$ on the quad-tree, prune the internal node $c_{(40,48)}$ whose minimal distance to o_6 is larger than $2\sqrt{2}$, and find NN of o_6 ($= o_5$) after accessing 7 internal nodes and 3 objects (bounded by blue dotted line in Figure 3(b)).

Cube-Based Partition - Improving Equal Space Partition. In Figure 3(a), the objects in $\mathcal{S}$ are skew-distributed, with many densely packed in a small subspace while many subspaces are empty, leading to an imbalanced quad-tree shown in Figure 3(b). For example, the first partition, corresponding to node $c_{(32,32)}$ in Figure 3(b), fails to seperate the objects into different cubes. Subsequent partitions can only separate a small number of objects (e.g., 1 or 2 in our example) from the others, resulting in a deep and imbalanced tree. The height of this quad-tree is 6. Generally, an imbalanced tree like this one typicall requires more partitions and longer search time to construct NNS because the dense hypercube, which contains numerous objects but occupies a small space, can only be located through multiple steps of equal space partitions.

To address the issue of imbalanced quad-trees caused by skew-distributed objects, we propose an efficient method that uses *median search* to locate dense hypercubes and construct a more balanced tree. Specially, given a set of objects $O = \{o_1, o_2, \cdots, o_l\}$, we compute the median value $\eta_{\frac{1}{2}}[j]$ of the objects' coordinates along the j^{th} dimension and define the virtual object $\eta_{\frac{1}{2}} = (\eta_{\frac{1}{2}}[1], \ldots, \eta_{\frac{1}{2}}[d])$. We then create a set of hypercubes $C = \{c_1, c_2, \cdots, c_l\}$ for O, where each c_i is the minimal hypercube that contains both $\eta_{\frac{1}{2}}$ and o_i.

Lemma 2. *Given a hypercube $c' \in C$ and let $C' = \{c | c \in C \wedge |c| \leq |c'|\}$, all cubes in C' are contained in c'.*

Proof Sketch. Given any two hypercubes $c, c' \in C$ with $|c'| \geq |c|$, they both contain point $\eta_{\frac{1}{2}}$ and hence c' must contain c. Since C' contains all the cubes in C having their side-length bounded by $|c'|$, the cubes in C' must be covered by the hypercube c'.

Algorithm 2: BUILD-QC-TREE

Input: Object Set O
Output: QC-tree $\mathcal{I}$

Hypercube $c_{min} \leftarrow$ minHyperCube(O);
Hypercube Set $C \leftarrow \emptyset$, Object Set List $SL \leftarrow \emptyset$, $f_{den} \leftarrow 0$;
Node $e \leftarrow$createNode($O, |O|, c_{min}$);
Construct a virtual object $\eta_{\frac{1}{2}} \leftarrow (\eta_{\frac{1}{2}}[1], \ldots, \eta_{\frac{1}{2}}[d])$;
for *each object* o_i *in* O **do**
    $c_i \leftarrow$ minHyperCube($o_i, \eta_{\frac{1}{2}}$), $C \leftarrow C \cup \{c_i\}$;
$l_{\frac{1}{2}} \leftarrow$ medianSideLen(C);
if $l_{\frac{1}{2}} < \frac{|c_{min}|}{2}$ **then**
    Object Set $S_{den} \leftarrow$getObject($l_{\frac{1}{2}}, O$), $SL \leftarrow SL \cup S_{den}$, $O \leftarrow O - S_{den}$, $f_{den} \leftarrow 1$;
$SL \leftarrow SL \cup$ equalPartition(O);
for *i from 1 to* $|SL|$ **do**
    if $|SL_i.O| = 1$ **then**
        Node $e' \leftarrow$CreateLeafNode($SL_i.O, f_{den}$);
        $e.childrenSet \leftarrow e' \cup e.childrenSet$;
    else if $|SL_i.O| > 1$ **then**
        Node $e'' \leftarrow$BUILD-QC-TREE($SL_i.O$);
        $e.childrenSet \leftarrow e'' \cup e.childrenSet$;
return e;

The key idea behind Lemma 2 is that, given two cubes c and c' in the hybercube set C, if the size of c is greater than that of c' (i.e., $|c| > |c'|$), c must contain c'. Therefore, once the hybercube set C is constructed for a given set of l objects O, we can locate the hypercube in C with the $\frac{l}{2}$-th largest side-length, denoted as c_{med}. Since each hypercube in C contains at least one object, c_{med} must cover at least $\frac{l}{2}$ out of l objects. Let $O.c$ be the minimal hypercube that contains all the objects in O. We can regard c_{med} as a dense hypercube if its side length is less than half of the side length of $O.c$, i.e., $|c_{med}| < \frac{|O.c|}{2}$. This is because c_{med} occupies less than $\frac{1}{2^d}$ space of $O.c$, but contains at least half of the objects in O. Otherwise, we consider that the object distribution of O is not skewed, and we partition the space into 2^d sub-space with equal size. Note that, as we will show in Theorem 1, this allows us to control the height of the index tree, regardless of the object distribution.

Returning to the dataset shown in Figure 3, we construct a virtual object $\eta_{\frac{1}{2}} = (60.1, 60.1)$ and form a group of six cubes, $C\{c_1, c_2, \cdots, c_6\}$, where $|c_1| = |c_2| = 32$, $|c_3| = 8$ and $|c_4| = |c_5| = |c_6| = 4$. For example, c_3 is cube $c_{(56,56)}$ with a side-length of 8, which is the minimal cube that bounds o_3 and $\eta_{\frac{1}{2}}$. The cube $O.c$ that bounds all six objects is a cube $c_{(32,32)}$ with a side-length of 32. We find the hypercube in C with the $\frac{l}{2}$-th largest side-length, denoted as c_{med}. As $|c_{med}| = 4$ ($< \frac{|O.c|}{2}$), c_{med} (a cube of side-length 4) bounds $\{c_4, c_5, c_6\}$, and is a dense cube. Note that o_4 and $\eta_{\frac{1}{2}}$ have the same coordinates, and we cannot find a minimal hypercube that bounds both. In our implementation, for an object o_i that shares the same coordinates as $\eta_{\frac{1}{2}}$, its hypercube $c_i \in C$ is set to the one that bounds o_i and meanwhile shares the same side-length as the minimum cube in $C - \{c_i\}$. For example, $|c_4|$ is set to 4 in our example.

The Index QC-Tree. Based on cube-based partition, we propose a novel index named *Quad Cube-Tree* (in short QC-Tree). Regardless of the distribution of streaming data in $[0, 1]^d$ space, QC-Tree has a bounded height of $O(\log n)$. Given a set $\mathcal{S}$ of streaming data to be organized by a QC-Tree $\mathcal{I}$, each node $e \in \mathcal{I}$ is represented as $\langle c, n, O, max \rangle$, where $e.c$ is a hypercube, $e.O$ is a set of $e.n$ objects that fall within $e.c$ (i.e., $e.n = |e.O|$), and $e.max$ is the maximal score of the objects in e (i.e., $e.max = max_{o \in e.O} \mathrm{D}(o)$). We maintain $e.max$ to support some operations that will be explained in Section 3.3.

Algorithm 2 explains the construction of QC-Tree, which is similar to the quad-tree. However, there are a few differences. Firstly, instead of partitioning objects step-by-step via equal space partition, it finds a minimal hypercube to cover objects in O (Line 1). Secondly, to prevent a deep path in the constructed tree, before constructing child nodes for an internal node $e \in \mathcal{I}$ under equal space partition, it checks whether c_{med} could be regarded as the dense hypercube (Lines 4-7). If so, QC-Tree introduces an extra child node e_{den} to maintain objects S_{den} located inside the dense hypercube (Lines 8-9). For the remaining objects, the algorithm uses equal space partition to construct corresponding nodes as quad-tree does (Line 10). Finally, when constructing a leaf node e' corresponding to an object o (Lines 12-14), if it has a sibling node e_{den} (i.e., $f_{den} = 1$), $e'.c$ is set to a cube that bounds o with $|e'.c| = \frac{|c'|}{2}$, where c' is the minimum cube that bounds $e_{den}.c$ and o.

Back to the example shown in Figure 3. We first initialize the root node e with $e.O = \mathcal{S}$, $e.n = 6$ and $e_0.c$ being a cube with a side-length of 32. We next locate the dense cube, which is a cube with a side-length of 4 that contains $\{o_4, o_5, o_6\}$. We then create a new node e_{den} based on these three objects. For the remaining objects, we partition them into two cubes $c_{(32,48)}$ and $c_{(48,48)}$, which contain $\{o_1, o_2\}$ and $\{o_3\}$, respectively. When forming the leaf node e_f for o_3, we set $e_f.c$ to a cube with a side-length of 4, which is half the side-length of cube $c_{(56,56)}$ that bounds o_3 and e_{den}. As a result, we obtain a more balanced tree with a height of 3, as shown in Figures 3(c).

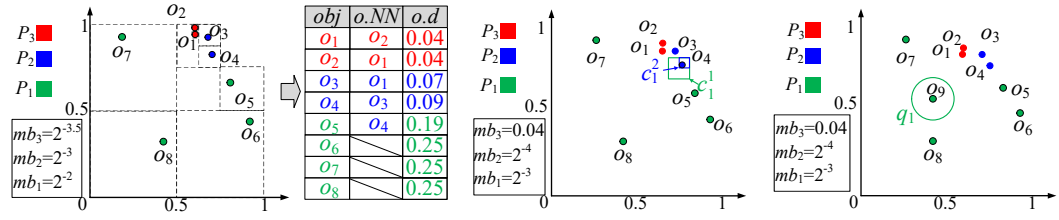
Theorem 1. *Given a non-leaf node e of QC-Tree, let Child(e) represent all its child nodes. If there is one child node $e_i \in Child(e)$ having more than $\frac{e.n}{2}$ objects, it is guaranteed that all the child nodes of e_i have no more than $\frac{e.n}{2}$ objects.*

Proof Sketch. Let e be an interval node of $\mathcal{I}$ and $\eta_{\frac{1}{2}}$ be the median point corresponding to $e.O$. If we can form a dense hypercube based on $e.O$, we only need to prove that all the child nodes of e_{den} have no more than $\frac{e.n}{2}$ objects. When e_{den} is partitioned into 2^d quadrants (e'_1 to e'_4, and e'_1 contains $\eta_{\frac{1}{2}}$), e'_1 must contain no more than $\frac{e.n}{2}$ objects based on the construction of e_{den}. Each of the other quadrants must be located at one-side of $\eta_{\frac{1}{2}}$ under at least one dimension, and hence the number of objects covered is no more than $\frac{e.n}{2}$. Otherwise (we cannot form the dense hypercube), we can obtain the same result following the logic discussed above.

According to Theorem 1, each object o can be partitioned into a leaf node *at most* $O(\log n)$ times, meaning the height of a QC-Tree is bounded by $O(\log n)$. Since we use *median search* to form both $\eta_{\frac{1}{2}}$ and the dense cube, partitioning objects in a node into its children has a cost that is linear to the dataset scale with a hidden constant d. As a result, the overall construction cost is $O(dn \log n)$.

Constructing NNS Using QC-Tree. The QC-Tree construction algorithm guarantees that each leaf node e has at least one sibling node e'. Both $e.c$ and $e'.c$ are covered by a hypercube with side-length $2|e.c_{(v_1,\ldots,v_d)}|$, which means Property 3.1 still applies to a QC-Tree. Specifically, the distance between o and its NN is bounded by $2\sqrt{d}|e.c_{(v_1,\ldots,v_d)}|$. To find the NN of each object $o \in \mathcal{S}$, we submit a range query with query radius of $2\sqrt{d}|c_{(v_1,\ldots,v_d)}|$. As the search process on $\mathcal{I}$ is similar to that on quad-tree, we skip the details.

Once the NN of every object is found, we construct NNS $\mathcal{N}$. In the event that o and o' are MNN, they only form one pair in NNS $\mathcal{N}$. Finally, we update $e.max$s of different nodes, propagating from



(a) L cube-based Partition & Score lower-bound Estimation. (b) Partition Refinement. (c) Incremental Maintenance.

Fig. 4. Example TNNS and τ-DLBP with $\tau = 1$.

leaf nodes to their parent nodes until the root node. To illustrate, consider the example shown in Figure 3(c). We can find the NN of o_6 by checking fewer nodes and objects than in a quad-tree search, specifically, four nodes and three objects bounded by red dotted line.

3.3 The Maintenance Algorithm

Object Insertion. When an object o_{in} flows into $\mathcal{S}$, we traverse the QC-Tree $\mathcal{I}$ to find nodes whose corresponding hypercubes cover o_{in}. If we encounter a leaf node e_f, we split e_f following the logic of QC-Tree construction. If we reach an internal node e such that none of e's children contains o_{in}, we construct a leaf node $e'\langle\{o_{in}\}, c, 1, +\infty\rangle$ for o_{in}. The hypercube $e'.c$ covers o_{in} and has a side length of $\frac{|e.c|}{2}$. If o_{in} falls within two child nodes (i.e., e_{den} and e_i that covers e_{den}), we insert o_{in} into e_{den}. After the insertion, we search for both NN of o_{in} and RNN (short for reverse nearest neighbour) of o_{in}, and update NNS based on the search result.

To be more specific, we only access each node $e \in \mathcal{I}$ from the root down to the leaf level with their minimum distances to o_{in} no larger than $r_d = \max(e.max, 2\sqrt{d}|c_l(o_{in})|)$. Here, $c_l(o)$ refers to the hypercube of a leaf node in $\mathcal{I}$ that contains object o, which is called the *L cube* (short for leaf node hypercube). Recall that the value of $e.max$ represents the maximum score of e's underlying objects. If o_{in} becomes a new NN for an object $o \in e.O$, then the distance $\mathsf{D}(o_{in}, o)$ must be smaller than $e.max$. We introduce $e.max$ to each node of $\mathcal{I}$ to facilitate the RNN search. On the other hand, $2\sqrt{d}|c_l(o_{in})|$ bounds the distance from o_{in} to its NN. Lastly, for each node e we have accessed, we update $e.max$ if necessary.

Object Deletion. When an object o_{exp} expires from $\mathcal{S}$, we search the QC-Tree $\mathcal{I}$ for nodes whose corresponding hypercubes cover o_{exp} until we reach the leaf node e_f. Along the traversal, we update the value of $e.n$ for each node e by subtracting 1 (i.e., $e.n \leftarrow e.n - 1$). When we reach e_f, we delete it from the tree. Additionally, for each RNN o of o_{exp}, we need to find its new NN and update NNS.

QC-Tree Local Re-construction. As new objects arrive and existing objects expire, the distribution of streaming data may change, causing the distribution of objects among nodes to change as well. This change can trigger the reconstruction of certain nodes in QC-Tree $\mathcal{I}$. Let e be a node of $\mathcal{I}$, e' be one of its child nodes, and e'' be a child node of e'. As stated in Theorem 1, $e''.n \leq \frac{e.n}{2}$ when $\mathcal{I}$ is constructed. If $e''.n \geq \frac{3}{4}e.n$, we need to invoke the construction algorithm to re-organize the objects maintained by e. We have chosen $\frac{3}{4}e.n$ as the threshold to reduce the frequency of node re-construction under the condition that the height of $\mathcal{I}$ is not high.

4 SUPPORTING KCP VIA PARTIAL NN PAIRS

In fact, it is not always necessary to use all NN pairs to answer KCP, as typically $k \ll n$. In this section, we present a method for maintaining a small number of NN pairs that can be used to efficiently answer KCP search.

4.1 The Threshold-based NN Pair Set

Definition 2. **TNNS**. *Let $\mathcal{S}$ be a streaming data set. TNNS $\mathcal{T}$ ($\subset \mathcal{S} \times \mathcal{S}$) consists of all objects o in $\mathcal{S}$ with their scores $D(o)$ no larger than $g(\tau)$, in the form of NN pairs $(o, NN(o))$. Here, $g(\tau)$ is the τ^{th} smallest score of objects in $\mathcal{S}$ given an integer τ ($\ll n$), and each qualified MNN pair is only captured once in TNNS.*

When a KCP search is submitted, if we can find k NN pairs in $\mathcal{T}$ with scores bounded by $g(\tau)$ (i.e., $|\mathcal{T}| \geq k$), we can use these k NN pairs to find the query result following the idea of Algorithm 1. To enable TNNS $\mathcal{T}$ to support KCP search even when $|\mathcal{T}| < k$, we propose a novel partition strategy τ-DLBP that divides objects into disjoint partitions $\{P_1, \ldots, P_m\}$. Each partition P_i maintains a score lower bound for the objects in P_i, and we can find the query results by searching a small number of partitions.

Definition 3. **τ-DLBP**. *A threshold Distance Lower-Bound based Partition τ-DLBP groups the streaming data set $\mathcal{S}$ into a set of m disjoint partitions $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$. Let $LB(P_i)$ refer to the lower bound of scores for objects in partition P_i, i.e., $\forall o_j \in P_i$, $D(o_j) \geq LB(P_i)$, and $|P_i|$ refer to the number of objects in partition P_i. Then $\mathcal{P}$ satisfies:* (1) $LB(P_{m-1}) > g(\tau)$; (2) $\forall i \in (1, m)$, $LB(P_{i-1}) \geq 2LB(P_i)$; *and* (3) $\forall i \in [1, m)$, $|P_i| \geq \sum_{j=i+1}^{m} |P_j|$.

The τ-DLBP uses Constraint (1) to ensure that all the objects in TNNS are maintained in P_m, although P_m may contain other objects as well. Constraint (2) allows τ-DLBP to use other partitions to maintain objects based on the *lower-bound* of their scores, which facilitates the process of KCP search with a large k while reducing the construction/maintenance cost of τ-DLBP since finding the lower bounds of objects' scores is easier and cheaper than finding objects' exact scores. We set this parameter to 2 so that the distance lower bounds of objects in adjacent partitions are exponentially decreased ($\mathsf{LB}(P_{i-1}) \geq 2\mathsf{LB}(P_i)$). As we will review in Section 4.3.3 and Section 4.4, this enables us to support KCP search and local re-partitioning in many cases by accessing only a small number of objects located in the last few partitions. Constraint (3) further limits the total number of partitions (i.e., m) to $O(\log \frac{|\mathcal{S}|}{\tau})$. In addition, the size of partitions drops exponentially, and objects with lower scores are contained in smaller partitions with finer granularity, as they have a higher chance of contributing to KCP search.

Figure 4(b) shows an example of τ-DLBP that groups objects in $\mathcal{S}$ into three partitions, based on $\tau = 1$. The color of the objects indicates the partition they belong to, and each mb_i value listed inside the box refers to the corresponding $\mathsf{LB}(P_i)$ of partition P_i. We have $P_1 = \{o_5, o_6, o_7, o_8\}$, $P_2 = \{o_3, o_4\}$, and $P_3 = \{o_1, o_2\}$ with $\mathsf{LB}(P_1) = 2^{-3}$, $\mathsf{LB}(P_2) = 2^{-4}$ and $g(\tau) = 0.04$.

4.2 The Partition Construction

Overview of τ-DLBP Partition. The construction algorithm of τ-DLBP partition starts by using a QC-Tree to obtain the L cube $c_l(o)$ for each object or leaf node. Recall that L cube $c_l(o)$, previously introduced in Section 3.3, represents the hypercube of a leaf node in QC-Tree that contains object o. Since a smaller $|c_l(o)|$ typically implies a potentially lower score $\mathsf{D}(o)$, we propose an initial partition of objects based on the side length of their L cubes. This initial partition is cost-effective and allows quick organization of objects into reasonable partitions, making it easy to find their

scores or lower bounds of their scores. The τ-DLBP partition consists of four steps, which we detail below.

Step 1: L cube-based Partition. Our partition algorithm starts by dividing the objects based on $|c_l(o)|$, the side length of their L cubes, using a *median search algorithm*. To be more specific, let S be the set of objects in the original stream dataset $\mathcal{S}$. We compute mb_1, the median side length of $|c_l(o)|$ corresponding to all the objects o in S. We then form the first partition P_1 by including all the objects $o \in S$ with $|c_l(o)| \geq mb_1$ and update S to $(S - P_1)$. Next, we compute mb_2, the median of $|c_l(o)|$ for all the objects o in the updated S, and form the second partition P_2 by including all the objects $o \in S$ with $|c_l(o)| \geq mb_2$ and update S to $(S - P_2)$. We repeat this process until the median side length $|c_l(o)|$ for all the remaining objects o in S is no larger than $2\sqrt{d}|c^{2\tau}|$. We then form the last partition P_m by including all the remaining objects in S and set $mb_m = 2\sqrt{d}|c^{2\tau}|$. Here, $c^{2\tau}$ refers to the L cube having the $(2\tau)^{th}$ shortest side-length among all the L cubes of the objects in the original stream dataset $\mathcal{S}$. The partitions generated in this step satisfy the following properties: (1) $\forall i \in (1, m)$, $mb_{i-1} \geq 2mb_i$; (2) $\forall i \in [1, m)$, $|P_i| \geq \sum_{j=i+1}^{m} |P_j|$; and (3) $\forall o \in P_i$ with $i \in [1, m)$, $|c_l(o)| \geq mb_i > mb_m$.

In Figure 4(a), dashed-line cubes refer to the L cubes of different objects. We have $\tau = 1$ and $|c^{2\tau}| = 2^{-5}$. Initially, $mb_1 = 2^{-2}$ and $P_1 = \{o_5, o_6, o_7, o_8\}$. Next, as $mb_2 = 2^{-3}$, which is still larger than $2\sqrt{d}|c^{2\tau}|$, we further form the partition $P_2 = \{o_3, o_4\}$. Finally, as the remaining two objects have their $|c_l(o)|$ no smaller than $2\sqrt{d}|c^{2\tau}|$, they form the last partition P_3 with $mb_3 = 2\sqrt{d}|c^{2\tau}| = 2^{-3.5}$.

Step 2: Score lower-bound Estimation. In this step, we aim to derive score lower bounds for each object based on a key property of the partitions formed in Step 1. Specifically, we know that for any object o in partition P_i, where $i \in [1, m)$, the side-length of its L cube is between mb_i and mb_{i-1}, i.e., $|c_l(o)| \in [mb_i, mb_{i-1})$.

To score each object, we utilize the distribution of cube side-lengths to set reasonable ranges for NN searches. Specially, we scan the objects partition by partition. For a given partition P_i and an object $o \in P_i$, we submit in total i range queries, denoted as $\cup_{j\in[1,i]} q_j(o, mb_j, P_j)$, to look for its NN in the first i partitions. Each of these i queries focuses on one specific partition P_j with $j \leq i$ and looks for objects $o' \in P_j$ that have their distances to o bounded by mb_j. For example, each object $o \in P_1$ submits one range query to find objects in P_1 that have distances to o bounded by mb_1, each object $o' \in P_2$ submits two range queries, one for each of P_1 and P_2, to find objects in P_1 and P_2 that have distances to o bounded by mb_1 and mb_2 respectively, and so on. We maintain the current NN and corresponding distance for each object o using $o.NN$ and $o.d$, respectively. We update these values based on the search results of range queries. Initially, $o.NN$ and $o.d$ are set to $NULL$ and mb_i, respectively.

After processing all the range queries that correspond to objects in $\mathcal{S}$, for each object $o \in P_i$, we can find ALL objects with distances to o bounded by mb_i. This is because the value of $o.NN$, which represents the NN of an object o in partition P_i, is not solely determined by the objects retrieved by the i range queries $\cup q_j$ issued by o. Instead, it may be updated when o falls within a range query $q_i(o', mb_i, P_i)$ issued by object o' in a partition P_l $(l > i)$ located behind P_i. For example, if o is retrieved by $q_i(o', mb_i, P_i)$ and $\mathrm{D}(o, o') < o.d$ or $\mathrm{D}(o, o') = o.d \wedge o.NN = NULL$, then $o.NN$ is set to o' and $o.d$ is set to $\mathrm{D}(o, o')$. In other words, if $D(o, o.NN) < mb_i$, then $o.NN$ is the actual NN of o. Otherwise, we use mb_i as the bound. Consequently, we can find either its actual nearest neighbor $NN(o)$ or the lower bound of its score for each object $o \in \mathcal{S}$, as stated in Theorem 2.

Furthermore, the NNs located by aforementioned range queries allow us to construct TNNS. This is because the search radius of all the range queries is no smaller than mb_m ($mb_m = 2\sqrt{d}|c^{2\tau}|$), and there are at least 2τ objects whose L Cubes' side-length is no larger than $|c^{2\tau}|$. According to

Property 3.1, those objects will have their scores bounded by $2\sqrt{d}|c^{2\tau}|$. Therefore, we are able to find all the NN pairs with scores bounded by $2\sqrt{d}|c^{2\tau}|$ and the number of these pairs is at least τ. As a result, we can compute $g(\tau)$.

Consider the initial partitions shown in Figure 4(a). Object o_5 in P_1 submits a total of one range query $q_1(o_5, 2^{-2}, P_1)$, which returns $\emptyset$. Thus, $o_5.NN$ remains NULL. Object o_4 in P_2 submits in total two range queries $q_2(o_4, 2^{-2}, P_1)$ (= $\{o_5\}$) and $q_3(o_4, 2^{-3}, P_2)$ (= $\{o_3\}$). After these queries, $o_4.d = 0.09$ and $o_4.NN = o_3$. In addition, the query q_2 issued by o_4 will update $o_5.NN$ to o_4, and $o_5.d$ to 0.19. After processing all the range queries submitted by all 8 objects, we can find the nearest neighbours for $\{o_1, o_2, o_3, o_4, o_5\}$ and derive score lower-bound for the remaining 3 objects, as shown in Figure 4(a). Consequently, we compute $g(1)$, which equals $o_1.d = 0.04$.

Theorem 2. *After all the range queries (in total $\sum_{i=1}^{m} i|P_i|$) corresponding to objects in $\mathcal{S}$ are processed, $\forall o \in P_i$, if $o.d = D(o, o.NN) < mb_i$ or $o.d = mb_i \wedge o.NN \neq NULL$, we have $o.NN = NN(o)$; otherwise, $o.d$ (which is initialized to mb_i) serves as the lower bound of $D(o)$.*

Proof Sketch: $\forall o \in P_i$, the range queries (e.g., the queries $\cup_{j=1}^{i} q_j(o, mb_j, P_j)$ issued by o and those $\cup_{o' \in P_l (l>i)} q_i(o', mb_i, P_i)$ issued by o' in partitions P_l with $l > i$) have already accessed all the objects in $\mathcal{S}$ that are within at most mb_i distance to o. Thus, if $o.d = \mathsf{D}(o, o.NN) < mb_i$ or $o.d = mb_i \wedge o.NN \neq NULL$, $o.d = \mathsf{D}(o)$. Otherwise, $o.d$ can serve as the lower bound of $\mathsf{D}(o)$.

Step 3: Partition Refinement. In this step, we refine the initial partition based on scores/scores lower-bounds derived in previous step. First, we calculate the median of the $o.d$ values for all objects o in $\mathcal{S}$ and denote it as mb'_1. Accordingly, we create partition P_1 to contain objects o with $o.d \geq mb_1$ (= $2^{\lfloor \log mb'_1 \rfloor}$). Next, we find the median of $o.d$ values for the objects o in $\mathcal{S} - P_1$, and denote it as mb'_2. We form partition P_2 to contain objects o' with $mb_1 > o'.d \geq mb_2$ (= $2^{\lfloor \log mb'_2 \rfloor}$), and so on. In this step, we use $2^{\lfloor \log mb'_i \rfloor}$ as the distance threshold instead of mb'_i to ensure that $\mathsf{LB}(P_i)$ (i.e., mb_i) is in the form of 2^u. We repeat the above process until the median of $o.d$ for all the remaining objects in $\mathcal{S} - \cup_{i=1}^{j} P_i$ is bounded by $g(\tau)$, where $g(\tau)$ is derived from the score lower-bound estimation step. We then create the last partition P_m (i.e., $m = j + 1$) by including all the remaining objects, set mb_m to $g(\tau)$, and construct TNNS $\mathcal{T}$. Note, partitions P_is with $i < m$ have mb_i set to the score lower bounds of the objects within the partitions, while P_m has mb_m set to $g(\tau)$.

Back to the example depicted in Figure 4. Based on $o.d$ values listed in the table, we first calculate the median of $o.d$ values, which is $o_5.d = 0.19$. Accordingly, we create partition P_1 with $mb_1 = 2^{\lfloor \log 0.19 \rfloor} = 2^{-3}$ to include objects $\{o_5, o_6, o_7, o_8\}$. We then find the median of $o.d$ for the remaining 4 objects, which is $o_3.d = 0.07$. Accordingly, we create partition P_2 with $mb_2 = 2^{\lfloor \log 0.07 \rfloor} = 2^{-4}$ to include objects o_3 and o_4. As the median of $o.d$ for the remaining two objects is 0.04, which is bounded by $g(1) = 0.04$, we create the last partition P_3 with $mb_3 = g(1) = 0.04$ to include o_1 and o_2. The final partitions and their score lower bounds are shown in Figure 4(b), where objects of the same color are located in the same partition. We then construct TNNS $\mathcal{T} = \{(o_1, o_2)\}$ accordingly.

Step 4: Cube-based Structure Construction. After constructing τ-DLBP, we build another data structure C_i for each partition P_i to support incremental maintenance. The data structure C_i is a set of hypercubes with side-length mb_i that bounds at least one object o in $\cup_{l=i}^{m} P_l$. Essentially, for each object o in the subsequent partitions $\cup_{l=i}^{m} P_l$, there must exist a hypercube $c \in C_i$ with side length $|c| = mb_i$ that bounds o. Each hypercube $c_j^i \in C_i$ is in the form of $\langle n_j^i, O_j^i \rangle$, where n_j^i records the total number of objects in the subsequent partitions $\cup_{l=i+1}^{m} P_l$ that fall within the hypercube c_j^i, and O_j^i refers to the set of objects in the current partition P_i that are within c_j^i. For the last partition P_m, its corresponding C_m contains hypercubes with side-length mb_m to maintain objects contained within P_m.

As an example, Figure 4(b) displays two cubes, c_1^1 and c_1^2, with side length of $|c_1^1| = mb_1 = 2^{-3}$ and $|c_1^2| = mb_2 = 2^{-4}$, respectively. c_1^1 contains only o_4, which is an object in P_2. Therefore, $n_1^1 = 1$ and $O_1^1 = \emptyset$. c_1^2 also only contains o_4, and thus $n_1^2 = 0$ and $O_1^2 = \{o_4\}$.

Discussion and Cost Analysis. Our main idea is to minimize the running cost required to form a rough partition based on side-length of objects' L cubes. We achieve this by using an L cube formation algorithm that has a running cost of $O(dn \log n)$, and L cube-based partition (also the partition refinement) via median search that has a running cost of $O(n)$. This allows us to (i) set a group of reasonable ranges for NN searches; and (ii) adopt a suitable search strategy for objects located at different partitions. Specifically, if an object o has a larger side-length of its L cube, it will be located at a partition with a smaller partition ID, and it will require fewer range queries. For the running cost of score lower-bound estimation, each object $o \in P_i$ submits i range queries. When searching in P_j ($j \in [1, i]$), the search region overlaps with 3^d hypercubes. If $i < m$, each hypercube contains at most one object, and the running cost spent on each object is bounded by $O(3^d i)$. Otherwise (i.e., $i = m$), the range query cost is bounded by $O(m3^d + \tau)$. Since $|P_i| \leq \frac{|P_{i-1}|}{2}$, the total cost spent on range queries is bounded by $O\big(\sum_{i=1}^{m} 3^d i|P_i| + |P_m|\tau\big)$, which is $O(\sum_{i=1}^{m} i\frac{n}{2^i} + |P_m|\tau\big)$, i.e., bounded by $O(3^d n + \tau^2)$. Therefore, the τ-DLBP construction cost is $O(n3^d + \tau^2 + dn \log n)$.

4.3 The Partition Incremental Maintenance

After τ-DLBP and TNNS are constructed, the objects in $\mathcal{S}$ may change as new objects arrive and/or existing objects expire. Consequently, it is important to maintain τ-DLBP and TNNS in a dynamic manner. In the following, we first explain how to update τ-DLBP and TNNS when an existing object expires or a new object arrives; we then present the local re-partition.

4.3.1 Expiry of an existing object o_{exp}. To handle object expiration, we make use of a property of τ-DLBP which organizes objects based on their score lower-bounds. The expiration of an object does not reduce the score (or score lower-bound) of any object, i.e., $\forall o \in P_i (i \in [1, m))$, mb_i is still a valid score lower bound even after o_{exp} expires. This means that when an object expires, we only need to (i) update the cube-based data structure and (ii) update TNNS if $o_{exp} \in \mathcal{T}.obj$.

To update the cube-based structure, we remove o_{exp} from P_i and update hypercube $c \in C_i$ that covers o_{exp} by excluding o_{exp} from $c.O$. In addition, $\forall c \in C_j$ corresponding to each partition $P_j (j < i)$ (in front of P_i) whose space covers o_{exp}, we reduce $c.n$ by one. If c (in either C_i or C_j) becomes $c\langle 0, \emptyset\rangle$, we delete c from C_i or C_j.

If $o_{exp} \in \mathcal{T}.obj$, we remove all the NN pairs in $\mathcal{T}$ that contain o_{exp}. In addition, for each of o_{exp}'s RNN o' in $\mathcal{T}.obj$, i.e., $o' \in (RNN(o_{exp}) \cap \mathcal{T}.obj)$, we submit a range query $q(o', g(\tau), P_m)$ to look for objects o'' (if any) in P_m that are within $g(\tau)$ to o'. This range query is to decide whether $\mathsf{D}(o')$ is still bounded by $g(\tau)$. If $q(o', g(\tau), P_m) \neq \emptyset$, we find the object o'' within the range of $g(\tau)$ that has the shortest distance to o'. The new NN pair (o', o'') is then included into TNNS $\mathcal{T}$. If $o_{exp} \notin \mathcal{T}.obj$, we adopt a lazy update strategy and do not process objects in $RNN(o_{exp})$.

4.3.2 Arrival of a new object o. To handle a new object o, we first need to determine its score (or its score lower bound) and assign o to a proper partition P_i, using the similar logic as the score lower-bound estimation step in constructing τ-DLBP. As o could become the new nearest neighbor to an existing object and decreases its score, we also need to process the affected objects. Fortunately, τ-DLBP has a useful property that allows us to efficiently handle such cases. Specially, for a partition P_i, if the range query $q_i(o, mb_i, P_i)$ issued by o does not return any object, then mb_i is still a valid score lower bound for all objects in P_i. Otherwise, if the query results are impacted by o, we need to further process the objects returned by the query. Finally, if o's score is no larger than $g(\tau)$, we update TNNS.

Scanning Partitions to Derive D(o). To calculate $\mathrm{D}(o)$ (or its lower bound) for a new object o, we again use range queries to scan objects partition by partition. Specifically, we conduct a range query $q_j(o, mb_j, P_j)$ per partition to identify objects in P_j that are within mb_j distance to o. We start with $j = 1$, $o.d = +\infty$, and $o.NN = NULL$. We gradually increase the value of j and update $o.NN$ and $o.d$ based on the objects returned by q_j. Let $C_j^{q_j}$ represent the overlap between the search range of q_j and all the hypercubes in C_j, i.e., $C_j^{q_j} = \{c \in C_j | c \cap range(o, mb_j) \neq \emptyset\}$. As the index j of a partition P_j currently evaluated increases its value from 1 gradually and the side-length mb_j of the hypercubes in C_j (that is also the radius mb_j of the range query q_j) keeps decreasing, their overlap $C_j^{q_j}$ becomes smaller. We terminate the scanning when we encounter a partition $P_i (i < m)$ where the overlap is reduced to zero (i.e., $C_i^{q_i} = \emptyset$) or none of the objects in subsequent partitions falls within the overlaps (i.e., $\sum_{c \in C_i^{q_i}} c.n = 0$).

This is because, as stated in Property 4.1, the distance from o to any object in the subsequent partitions is longer than mb_i, and hence we can safely terminate the evaluation of partitions. If $o.d = \mathrm{D}(o, o.NN) \leq mb_i$, where P_i is the last partition we scan, then $o.d$ is the score of o. Otherwise, we use mb_i as score lower-bound of o.

Property 4.1. *Recall that $C_i (i < m)$ is the set of hypercubes with side length mb_i associated with partition P_i. If $C_i^{q_i} = \emptyset$ or $\sum_{c \in C_i^{q_i}} c.n = 0$, $\forall o' \in \cup_{j=i+1}^{m} P_j$, $D(o, o') > LB(P_i) = mb_i$.*

Inserting o to a Proper Partition. To insert an object o into the correct partition, we compare its distance $o.d$ with mb_i values of the partitions. If $o.d < mb_{m-1}$, we insert o into P_m. If $mb_{m-1} \leq o.d < mb_1$, we insert o into a partition P_j where $mb_j \leq o.d < mb_{j-1}$. Otherwise ($o.d \geq mb_1$), we insert o into P_1. This process guarantees that o will be inserted into the partition P_i such that $\mathrm{D}(o)$ is within the range of $[(1 + \sqrt{d})mb_{i-1}, mb_i]$, as stated in Theorem 3. Note, even if object $o.NN$ is not the real nearest neighbor of o (i.e., $o.d \geq \mathrm{D}(o)$), we ensure that o is inserted into the correct partition. Additionally, the hypercube set C_l of the affected partitions is updated.

Theorem 3. *Given an object o to be inserted to a partition P_i, if $o \notin \mathcal{T}.obj$, it is guaranteed $D(o) \in [(1 + \sqrt{d})mb_{i-1}, mb_i]$.*

Proof Sketch: Let P_j be the last partition we have accessed, P_i be the partition o is inserted into, and o' be the object in the first j partitions $\cup_{l=1}^{j} P_l$ having the shortest distance to o. If $\mathrm{D}(o, o') \leq mb_i$, o' must be $NN(o)$ following the logic discussed in Theorem 2. Otherwise, there is at least one hypercube c ($c.n > 0$) of C_{j-1} associated with partition P_{j-1} that overlaps with the search range of q_{j-1}. This implies that at least one object o'' in $\cup_{l=j}^{m} P_l$ fallen inside c. As the maximal distance between o and c is no larger than $(1 + \sqrt{d})mb_{j-1}$, we have $\mathrm{D}(o) \leq \mathrm{D}(o, o'') \leq (1 + \sqrt{d})mb_{j-1} (i = j$ under this case, and $\mathrm{D}(o) \in [(1 + \sqrt{d})mb_{i-1}, mb_i]$ is also held).

Take an example in Figure 4(c). When object o_9 arrives, we scan partitions to find its score. We first submit a range query $q_1(o_9, mb_1 = 2^{-3}, P_1)$ on partition P_1, which returns $\emptyset$. Meanwhile, we have $\sum_{c \in C_1^{q_1}} c.n = 0$, which means that there are no objects in P_1 that fall within the range of q_1. Consequently, we terminate the evaluation, and use mb_1 as the lower bound of $\mathrm{D}(o_9)$. We then insert o_9 into partition P_1 and update C_1 by including a new hypercube.

Impact of o on Existing Objects. As presented above, when evaluating a partition P_j ($j < m$), the corresponding range query $q_j(o, mb_j, P_j)$ locates all the objects in P_j that are within mb_j distance to o. $\forall o' \in q_j(o, mb_j, P_j)$, object o becomes the new NN of o' and mb_j is no longer a valid lower bound of o''s score. We move o' to an appropriate partition using the same logic applied to process o and update the hypercube set C_l of the affected partitions accordingly.

Impact of o on TNNS. If $\mathsf{D}(o) \leq mb_m$, we form a new pair $(o, o.NN)$, and insert it into TNNS. If $\mathsf{D}(o)$ is smaller than $g(\tau)$, we reduce $g(\tau)$ accordingly as $g(\tau)$ refers to the τ-th smallest score of objects in $\mathcal{S}$. We update TNNS so that it only contains objects whose scores are bounded by the new $g(\tau)$. We perform the same operations to handle objects that are impacted by o if their scores are reduced to no larger than $g(\tau)$. In particular, if $g(\tau)$ is reduced to $\frac{mb_m}{2}$, we update mb_m to the new $g(\tau)$ value and update hypercubes C_m associated with P_m accordingly based on the new side-length mb_m.

4.3.3 The Local Re-Partition. After τ-DLBP is constructed, the partitions may need to be adjusted due to the arrival of new objects or expiry of existing objects. This is done through local re-partition of τ-DLBP, which is triggered if any of the following conditions are met: i) $|\mathcal{T}| < \frac{\tau}{2}$; or ii) $\exists i \in [1, m-1]$ such that $|P_i| \leq \sum_{j=i+1}^{m} |P_j|$; or iii)$|\mathcal{T}.obj| < \frac{|P_m|}{2}$.

Local Re-Partition Under Condition i). When some objects expire from $\mathcal{T}.obj$, local re-partitioning under condition i) is triggered, which involves reforming TNNS using the partitions of τ-DLBP. A useful property of τ-DLBP is that for each partition $P_i (i < m-1)$, $\mathsf{LB}(P_i) \geq 2\mathsf{LB}(P_{i+1})$ and $|P_i| \geq \sum_{j=i+1}^{m} |P_j|$. This allows us to efficiently access a small number of objects located at the last few partitions to reform TNNS in most cases.

Specifically, we first consider objects in partition P_m and we perform L *cube-based partition* and *score lower-bound estimation* of τ-DLBP construction based on objects in P_m. If we can find at least τ NN pairs with scores smaller than $\mathsf{LB}(P_{m-1})$, we do not search other partitions, as the scores of objects in other partitions are all no smaller than $\mathsf{LB}(P_{m-1})$. Otherwise, we consider objects in P_m and the partition right before P_m. We merge P_{m-1} into P_m and repeat the above operations. Again, if we can find at least τ NN pairs with scores smaller than $\mathsf{LB}(P_{m-2})$, we do not search other partitions, but instead execute the *Partition Refinement* of τ-DLBP partition construction. Otherwise, we should consider objects in P_{m-2} and so on. Here, we use the notation "m" to refer to the last partition that contains objects of TNNS, even though the actual value of m may change during the local re-partition.

Note that when objects expire, the scores of some objects may become larger. Consequently, lower bounds associated with partitions might be very loose (e.g., the real lower bounds are much larger than the current lower bounds). This could lead to the search not terminating even after searching all partitions. In such cases, we re-construct τ-DLBP from scratch using the τ-DLBP construction presented in Section 4.2.

The algorithm is simple, but we would like to highlight two points. Firstly, after forming the new $\mathcal{T}$, we need to remove objects in P_m with scores/score lower-bounds no smaller than $\mathsf{LB}(P_{m-1})$ into P_{m-1}. Secondly, we use a QC-Tree-liked index $\mathcal{I}'$ to maintain part of objects in $\mathcal{S}$ and use the side-length of hypercubes in the leaf nodes of $\mathcal{I}'$ to estimate the score lower bounds of objects, instead of using L cubes of objects. To keep the scale of $\mathcal{I}'$ small, each object $o \in \mathcal{S}$ is maintained in $\mathcal{I}'$ if it has participated at least once in the latest n local re-partitions of $\mathcal{S}$.

Take Figure 4(c) as an example. When o_1 expires from $\mathcal{S}$, TNNS becomes empty and local re-partition is triggered under condition i). Since there is only one object in P_m ($m = 3$), we first merge the objects in P_2 and P_3, i.e., $\{o_2, o_3, o_4\}$. After performing L cube-based partition and score lower bound estimation, we find that $o_2.NN = o_3$ and $o_3.NN = o_2$ with $o_2.d = o_3.d = 0.08$, and $o_4.NN = o_3$ with $o_4.d = 0.09$. Since all three scores are smaller than $\mathsf{LB}(P_1) = mb_1 = 2^{-3}$, we set $g(\tau) = mb_2 = 0.08$ and TNNS $\mathcal{T} = \{(o_2, o_3)\}$. As 0.08 is the median of $o.d$ value of objects in P_m, no partition refinement is required. Therefore, $\{o_2, o_3, o_4\}$ form the last partition $P_m (m = 2)$.

Local Re-Partition Under Other Conditions. For condition ii) $|P_i| < \sum_{j=i+1}^{m} |P_j|$, we can treat objects in $\cup_{j=i}^{m} P_j$ as a dataset and apply the *partition refinement* step introduced in Section 4.2 to

perform local re-partition. For condition iii) $|\mathcal{T}.obj| < \frac{|P_m|}{2}$, i.e., when partition P_m contains many objects with scores (or score lower-bounds) in the range of $(g(\tau), mb_{m-1})$, we split objects in P_m into P_m and P_{m+1}, using the same logic as *partition refinement* step presented in Section 4.2.

Discussion. In some extreme cases, the local re-partition may lead to a high cost if the top-τ NN pairs with smallest scores keep changing. For example, if many objects of $\mathcal{T}$ expire, it triggers the local re-partition due to condition (i), which in turn increases $g(\tau)$ and leads to objects from other partitions (e.g., P_{m-1}) being merged into partition P_m. This can result in P_m becoming much larger, triggering the local re-partition because of condition (iii) and causing partition P_m to be split into two partitions. If objects in the newly formed $\mathcal{T}$ expire again, P_m again requires expansion by including objects from other partitions. The merge and split of P_m may happen frequently. Although this scenario has a low occurrence probability, we provide a solution by maintaining additional information related to NN pairs whose scores are within the range of $[mb_m, 2mb_m]$ to alleviate the high cost after P_m is split into P_m and P_{m+1}.

4.4 KCP Search Under TNNS and τ-DLBP

When a KCP search is submitted, if there are at least k such pairs, the top-k pairs are selected as the result pairs to complete the search, following the logic presented in Algorithm 1. However, if there are less than k pairs, we need to scan objects in other partitions to find k closest pairs, following the same logic used to perform local re-partition under condition (i). For example, if the top-k pairs in P_m have their scores no greater than $\mathsf{LB}(P_{m-1})$, these k pairs form the answer set. Similarly, if the top-k pairs can be found in $P_m \cup P_{m-1}$ with scores no greater than $\mathsf{LB}(P_{m-2})$, these k pairs form the answer set, and so on. As the algorithm is almost the same as local re-partition under condition i), we skip the details to save space.

In the following, we will briefly discuss the overall cost of supporting KCP search under τ-DLBP. As the number of partitions is bounded by $O(\log \frac{n}{\tau})$, the running cost of processing each update is bounded by $O\big((d + 3^d) \log \frac{n}{\tau} + \tau\big)$. The local re-partition under condition i) is based on τ-DLBP construction. We assume that the last u partitions are involved in the local re-partition. As $|P_i| \geq 2|P_{i+1}|$ is guaranteed, the overall running cost is $O(3^d \sum_{i=m-u+1}^{m} \frac{i|P_i|}{2^i} + \tau^2 + d|P_u| \log |P_u|)$, i.e., bounded by $O(3^d n + \tau^2 + dn \log n)$ (amortized cost of each object is $O(3^d + \tau + \log n)$). Moreover, we can prove that the cost of supporting one KCP search under τ-DLBP is $O(3^d n_k + k^2 + dn_k \log n_k)$, following the logic of local re-partition. Here, n_k refers to the number of objects that are evaluated during the search, and its value is excepted to be $(2^j - 1)\tau$ where j is the minimal integer satisfying $\sum_{i=0}^{j} |P_{m-i}| \geq 2k$.

Based on the above analysis, we understand that parameter τ provides a trade-off between the KCP search efficiency and the update cost of τ-DLBP. Setting a larger τ increases the number of NN pairs maintained by TNNS, which in turn enhances the chances of finding objects in TNNS directly to support KCP search with a lower search cost. However, a larger τ also increases the maintenance cost of τ-DLBP. Thus, the selection of τ value depends on several factors such as the distribution of k, query submission speed (the number of submitted queries per time unit), update frequency, and object distribution. If we have prior knowledge about these factors and the relative importance of update performance and search efficiency for the application, we can create a cost model to determine the optimal τ value. However, if we lack prior knowledge, we recommend setting τ to a constant. In our implementation, we set τ to 100 as we assume zero prior knowledge. Nonetheless, its value can be adjusted easily to accommodate the needs of different applications.

Table 2. Parameter Settings.

Parameter	Values
n	100K, 200K, 500K, **1M**, 2M, 5M, 10M
mean of k (SD= 1)	10, 20, 50, **100**, 200, 500, 1000
s	0.01%, 0.05%, 0.1%, 0.5%, **1%**, 5%, 10% ($\times n$)
τ	10, 50, **100**, 200, 500, 1000

5 PERFORMANCE EVALUATION

In this section, we present the results of our extensive experiments designed to demonstrate the efficiency of our proposed techniques.

5.1 Experiment Settings

Datasets. We conduct our experiments using five datasets, including two real datasets, Stock and Trip, and three synthetic datasets, Multi, Uniform and Normal.

(i) Stock. It consists of 1G stock transactions corresponding to 2,300 stocks from Shanghai/Shenzhen Stock Exchange over a period of 24 months. Each transaction record r is defined as (p, v, t), where p and v refer to the price and volume of the transaction, respectively, and t records the transaction time. Two stocks are considered similar if they share similar prices and similar cumulative volumes over a period of time. KCP search enables us to find potential connections between stocks. In our implementation, records are flowed into/expired from $\mathcal{S}$ based on their transaction time.

(ii) Trip. It contains 1.6G trip records from NYC, collected over a period of 72 months [48]. Each record r in Trip is denoted as (o, d, t), where o/d indicates its origin/destination location, and t refers to its pick-up time. Trip records flow into $\mathcal{S}$ based on their pick up time and expire from $\mathcal{S}$ after another $|\mathcal{S}|$ objects flow into $\mathcal{S}$. KCP search can find highly similar bookings.

(iii) Multi. It contains 1G 4-dimensional objects divided into 1,024 equal-sized subsets. Objects in the same subset follow a common normal distribution with mean and standard deviation randomly generated from the ranges of [1000,9000] and [1,10], respectively. We use this dataset to evaluate the effect of data distribution on algorithm performance.

(iv) Uniform and (v) Normal. They contain 1G objects with 2 to 8 dimensions respectively, where 4 is the default dimensionality. Objects in them follow uniform and normal distribution, respectively. Object arrivals and expiration are randomly generated.

Distance functions. For Trip, $(|r_2.o - r_1.o|^2 + |r_2.d - r_1.d|^2 + \alpha|r_2.t - r_1.t|^2)^{1/2}$ is the distance function, where α is a constant related to the average speed of taxis and it is set to 40 in our implementation. For other datasets, Euclidean distance between objects is considered.

Query workload. Each query workload consists of 100 queries. The values of k in each query workload follow normal distribution, with the mean values listed in Table 2 and a standard deviation (SD) of 1.

Other Parameters. We evaluate the performance of different algorithms under parameters n and s. n refers to the size of the streaming data set $\mathcal{S}$; and s refers to the *update rate* of the stream; specifically, $s\% \times n$ new objects flow into $\mathcal{S}$ whenever $\mathcal{S}$ updates, and $s\% \times n$ existing objects expire from $\mathcal{S}$. We use parameter s to simulate the speed of the stream. In addition, we study the impact of τ on τ-DLBP, as it only maintains top-τ NN pairs in TNNS. Table 2 shows the parameter settings, with bold indicating the default values.

Performance Metrics. We use the following five performance metrics. 1) *Overall running time* measures the algorithm's performance when handling query workloads and updates. We report the

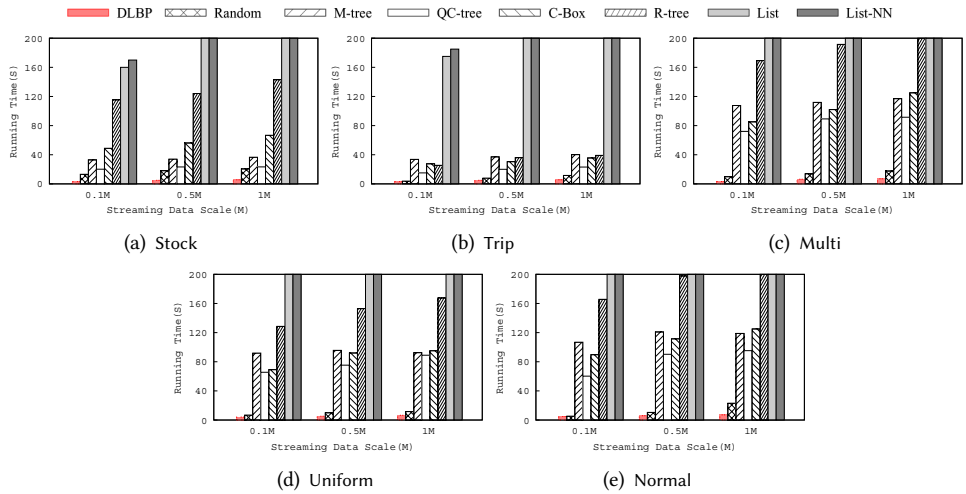


Fig. 5. Overall Performance Comparison under different n (mean of $k = 100$, $s = 1\% \times n$ and $\tau = 100$)

average time required to handle 100 query workloads and 1M updates. 2) *Construction time* evaluates how fast the algorithms can construct their indexes during initialization. 3) *Data throughput* measures the average number of updates per second that the algorithm can process, which indicates how fast the algorithms respond to updates. 4) *Query throughput* measures the average number of queries per second that the algorithm can process, which indicates how fast the algorithms respond to KCP search. 5) *Index size* records the average size of the indexes used by different algorithms.

Competitors. This study presents the first attempt to support KCP search over data streams. We extend several state-of-the-art algorithms to serve as competitors.

(i) C-Box [5]: It maintains all MNNs to support 1CP search. To support KCP search, we maintain all NN pairs instead. In other words, C-Box borrows the algorithm from NNS to support KCP search but uses a different index to maintain streaming data.

(ii) Random [6]: It can support 1CP as it has the closest pair in the last partition. To support KCP search, we assume that it uses the last partition to maintain TNNS, following the logic discussed in Section 4. It borrows the algorithm from TNNS to support KCP search but uses a different approach to partition the streaming data.

(iii) LIST [29]: It supports KCP search under sliding window based on domination relationship among pairs, which is not applicable in our setting. We modify it to maintain n' ($\in [\sqrt{n}, 2\sqrt{n}]$) pairs with the smallest scores, similar to the buffer-based algorithm [1]. LIST pre-processes KCP search and can provide immediate answer if $k \leq n'$, but it cannot support KCP search when $k > n'$.

(iv) LIST-NN [30] and (v) M-Tree [8]: They use inverted-list-based and M-Tree-based structures, respectively, to maintain streaming data and support KCP search via maintaining all NN pairs.

(vi) R-Tree [25]: It does not consider how to support KCP search under data stream. We modify it by borrowing the algorithm from QC-Tree to incrementally maintain all NN pairs to support KCP search.

In addition to these six competitors, we implement QC-Tree and τ-DLBP, two structures proposed in this paper to support KCP search. The former maintains all NN pairs via a tree structure and supports KCP search via algorithm 1. The latter partitions objects based on their score lower bounds and maintains top-τ NN pairs via TNNS. All algorithms are implemented in C++, purely in memory, and experiments are conducted on a 622R CPU with 256GB of memory, running on Win 10.

5.2 Algorithm Comparisons

Overall Performance Comparison. We first conduct experiments to compare the overall performance of all the algorithms across different n values. Figure 5 reports the average running times of the algorithms handling 1M updates and 100 query workloads. We find that τ-DLBP consistently outperforms all competitors across all datasets. For example, under Stock, it is on average $5X$, $8.2X$, $4.5X$, $9X$, $30.3X$, $50X$, and $53X$ times faster than QC-Tree, M-Tree, Random, R-Tree, C-Box, LIST, and LIST-NN, respectively. This result is mainly due to τ-DLBP's ability to avoid maintaining all NN pairs while keeping the number of partitions bounded by $O(\log \frac{n}{\tau})$. This feature allows for processing of every newly arrived or expired object within logarithmic time. In contrast, QC-Tree, M-Tree, R-Tree, C-Box and LIST-NN have to maintain all NN pairs, while Random may result in partitions that are either too small or too large. When there are too many partitions, each newly arrived object has to access many partitions, and when there are too few partitions, the cost of local re-partition or KCP search may be high. LIST, on the other hand, incurs high overall cost because newly arrived objects have to access multiple elements in each inverted list.

Secondly, we observe that the running time of τ-DLBP is stable and not sensitive to data distribution. As τ-DLBP organizes objects based on their scores or the lower score bounds, the distribution of the streaming dataset has a limited impact on its performance. In contrast, Random is sensitive to data distribution and may not form a suitable partition when data is skewed distributed. This explains why Random incurs much longer running time under Stock and Normal. Similarly, M-tree, C-Box, QC-Tree and R-tree are sensitive to the change of data distribution too, causing their running times to increase significantly under different scenarios. For example, they all require significantly longer running time under Multi and Normal. These algorithms need to frequently adjust the tree structure to accommodate the changes in data distribution.

Thirdly, we observe that QC-Tree outperforms M-Tree. One possible reason is that when the data dimension is low, the distribution of streaming data is not sparse, and the L cubes used by QC-Tree provide objects with relatively tight boundary. Since there is no overlapping among different L cubes, QC-Tree has a stronger pruning ability. However, when data dimension d increases, the searching radius of QC-Tree also increases, leading to the algorithm accessing more cubes when searching for object' NN. This partly offsets the advantages of QC-Tree discussed earlier. Specifically, we find that when the data dimension d equals to 4, the running time difference between these two algorithms becomes small.

Lastly, we observe that LIST and LIST-NN perform significantly worse than the other algorithms across all datasets and for various n values. While LIST can provide immediate answers to KCP search when $k \leq n'$ by maintaining n' pairs with the smallest scores, it is not suitable for a highly dynamic environment where objects frequently change. The reported running time is based on handling 1M updates and 100 query workloads, which is mainly dominated by the updates. Similarly, LIST-NN is also unsuitable for supporting KCP search under data streams. For example, the running time of both algorithms exceeds 200 seconds in most cases, while τ-DLBP takes less than 10 seconds. Thus, for clearer presentation, we exclude LIST and LIST-NN from the remaining experiments.

Construction Cost Comparison. We evaluate the cost of index construction for all the algorithms across different datasets. The results, as shown in Figures 6(a), 6(e), 6(i), 6(m), and 6(q), indicate that τ-DLBP has the most efficient index construction. This is because τ-DLBP partitions objects based on side-length of their L-Cubes, which reduces the number of NNs that need to be found, resulting in a significant reduction in construction time. In contrast, Random often fails to create proper partition, leading to repeated partitioning and higher construction costs; tree based algorithms have to find NNs for all the objects and hence their construction cost is high.

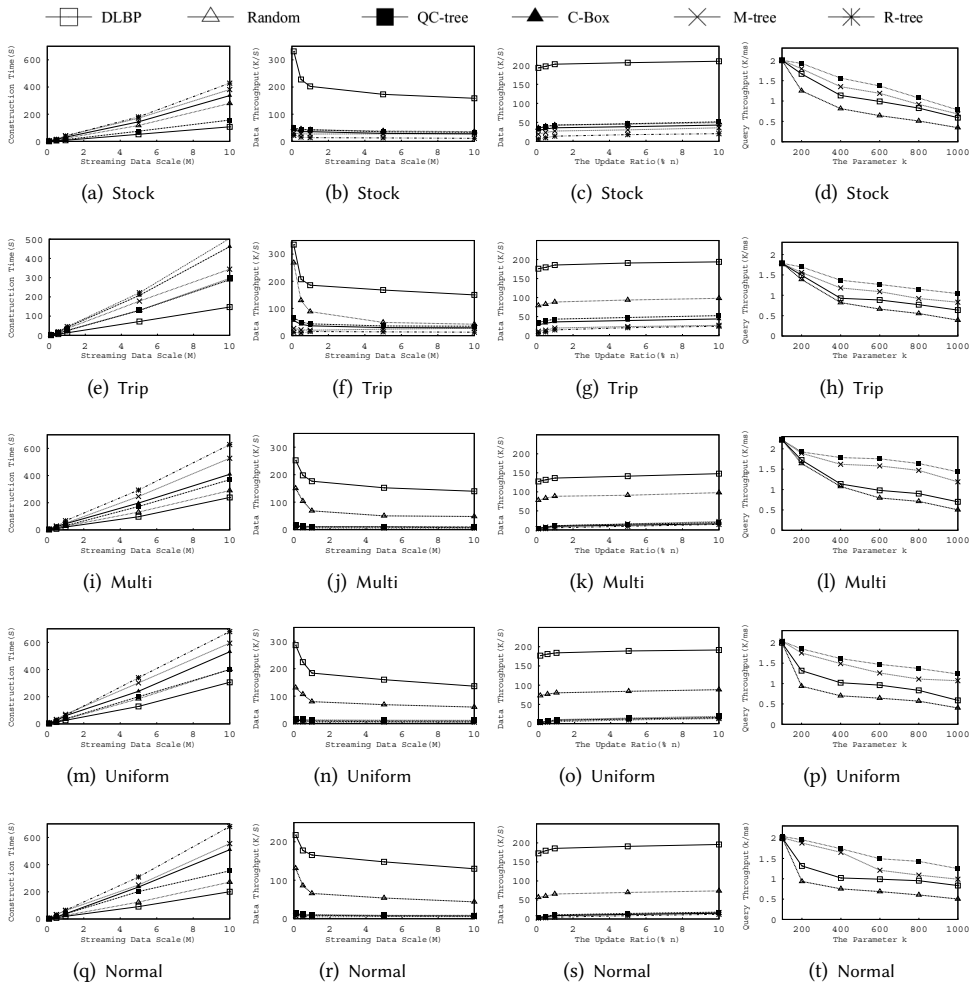
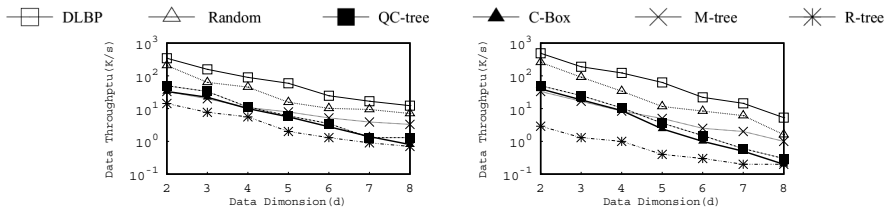


Fig. 6. Algorithms Comparison under different datasets (n =1M, $s = 1\% \times n$, mean of $k = 100$, and $\tau = 100$)

Data Throughput Comparison. Figures 6(b), 6(f), 6(j), 6(n), and 6(r) show the data throughput results of various algorithms for different n values. τ-DLBP performs the best again, in part because most objects under τ-DLBP can be inserted into a proper partition by accessing a small number of partitions. By contrast, inserting an object into a QC-Tree or C-Box incurs a running cost that is linear to their heights. Furthermore, we find that QC-Tree always outperforms C-Box, as QC-Tree can efficiently adjust its structure to maintain a balance. Also, QC-Tree always outperforms M-tree and R-tree. One main reason is there is no overlapping among different L cubes and QC-Tree has a stronger pruning ability. We also find that Random performs better than QC-Tree in most cases, though it has a similar data throughput than QC-Tree under Stock. This well demonstrates that the performance of Random is not stable as it cannot always find suitable "pivots points", especially when data distribution is skewed.

We also report the data throughput under different s values, with n fixed at 1M, in Figures 6(c), 6(g), 6(k), 6(o), and 6(s). We observe that as s increases, all the algorithms have higher throughput and τ-DLBP again performs the best. The reason is that a larger s leads to lower cost for processing



(a) Uniform (b) Normal

Fig. 7. Update Time vs. d using synthetic datasets

expiration objects. For example, suppose $p(o, o')$ is a NN pair. If o and o' both expire from $\mathcal{S}$ at the same time, we do not need to re-search their NNs.

Query Throughput Comparison. We then evaluate the query throughput of KCP search using different algorithms. As QC-Tree, C-BOX and R-Tree use the same KCP search algorithm, they incur the same query throughput and we only report the results under QC-Tree. Figures 6(d), 6(h), 6(l), 6(p) and 6(t) report the results. As expected, QC-Tree outperforms others since it can quickly find the answer to KCP search by accessing objects that form the top-k NN pairs. On the other hand, M-Tree requires a longer query time than QC-Tree, as it considers top-$2k$ pairs when supporting KCP search. τ-DLBP achieves comparable performance, with a query throughput about 75% of QC-Tree, thanks to its proper partitioning of objects and the ability to support KCP search via accessing a small number of partitions even when $k > \tau$. In contrast, Random requires a much longer query time, highlighting the importance of proper partitioning, as the main difference between Random (after major extension) and τ-DLBP is the way they form partitions.

Data Throughput vs. d. We further evaluate the data throughput of different algorithms when dimansionality d increases, with the results reported in Figure 7. We observe that the data throughput of all algorithms decreases with the increase of d, and τ-DLBP outperforms all the competitors once again. One important reason for this is that as d increases, the score difference among objects becomes smaller, leading to fewer partitions that need to be maintained. We also find that after d reaches 5, M-Tree outperforms QC-Tree. One reason is that when d increases, the search region under QC-Tree becomes larger, and more cubes need to be accessed. In contrast, M-Tree uses a distance-based split criterion to guide the tree construction, which ensures that nearby points are grouped together in the same sub-tree, thereby reducing the search space.

Index Size Comparison. We report the index size of different algorithms in Table 3. It is observed that the size difference among various indexes is small. All the algorithms have to spend extra space in maintaining streaming data. For example, τ-DLBP and Random use a group of hash tables to maintain objects in each partition; QC-Tree, C-Box, and R-Tree have to maintain all NN pairs.

QC-Tree vs. Quad-Tree. To demonstrate the advantage of QC-Tree over Quad-Tree, we compare their performance via an experimental study. We observe consistent comparison trends across all five datasets, though the advantage of QC-Tree over Quad Tree diminishes as d increases. For the sake of brevity, we only report our results on TRIP ($d = 3$) under different n values in Table 4. As observed, QC-Tree is more efficient in terms of construction and incremental maintenance, primarily due to its lower height. As a contrast, Quad-Tree may have a higher height if the object distribution is skewed, which can result in more processing time for range queries during the construction and maintenance of NNS.

The impact of τ. In our last set of experiments, we evaluate the impact of τ on the performance of τ-DLBP. The overall running time required by τ-DLBP for handling 1M updates and 100 query

Table 3. Index size of different algorithms (unit:MB)

Dataset	Stock($d = 2$)	Trip($d = 3$)	Multi($d = 4$)	Normal($d = 4$)	Uniform($d = 4$)
τ-DLBP	49.0	73.2	92.1	91.8	94.2
QC-Tree	53.0	81.0	106.5	111.7	109.2
Random	42.0	64.0	72.5	79.1	76.1
C-Box	51.6	72.0	107.4	105.9	103.1
M-Tree	55.1	80.0	112.1	117.4	119.2
R-Tree	57.1	84.2	109.3	115.2	116.4

Table 4. QC-Tree vs. Quad-Tree under Trip ($d = 3$)

	streaming data scale n	100K	200K	500K	1M	2M	5M	10M
QC-Tree	Construction Time (s)	2.1	4.8	11.8	24.9	60.1	129.1	288.7
	Data Throughput (K/S)	34.6	33.3	33.3	29.3	27.3	25.3	23.9
	Average Height	6.4	7.1	8.5	8.9	9.3	10.6	11.2
Quad-Tree	Construction Time (s)	3.1	6.3	14.1	28.7	64.9	147.4	316.5
	Data Throughput (K/S)	29.7	27.2	26.5	24.3	22.8	20.8	18.4
	Average Height	10.3	10.8	11.5	12.6	13.1	14.3	15.1

Table 5. Overall running time of τ-DLBP vs. τ (unit: s)

τ	10	50	100	200	300	400	400	600	800	1000
Stock	2.77	2.69	2.71	2.66	2.66	2.66	2.68	2.72	2.76	2.80
Trip	3.93	3.84	3.80	3.79	3.79	3.79	3.77	3.82	3.86	3.88
Mutli	4.96	4.84	4.77	4.77	4.77	4.79	4.76	4.71	4.66	4.75
Normal	4.41	4.23	4.02	4.02	4.02	3.99	3.76	3.71	3.86	3.95
Uniform	4.20	4.50	4.10	4.10	4.10	4.40	3.90	3.80	4.00	4.40

workloads under different τ values is reported in Table 5. The result shows that the parameter τ has a relatively small impact on the performance of τ-DLBP. While a small τ reduces the maintenance cost, it increases query time, as discussed in Section 4.4. Nevertheless, its impact on overall performance is small. Notably, even when $\tau = 10$, which is much smaller than the mean of k (= 100), τ-DLBP still outperforms its competitors (whose performance is shown in Figure 5) by a significant margin.

6 CONCLUSION AND FUTURE WORKS

In this paper, we propose a novel concept named NNS for supporting KCP search over stream data. It is able to return the result pairs via accessing $O(k)$ objects. Furthermore, we propose TNNS and a group of algorithms for supporting KCP search under TNNS. We have conducted extensive experiments to evaluate the performance of our proposed algorithms on several datasets with different distributions. The results demonstrate the superior performance of our proposed algorithms.

In the near future, we would like to study approximate search to support KCP search in streaming data with higher dimensionalities. We plan to propose a τ-DLBP-alike structure to organize objects, but apply an approximate algorithm with error guarantee to calculate approximate scores/score lower bounds of objects, which can provide a flexible trade-off between query quality and update efficiency to cater for different needs of real applications and alleviate the curse of dimensionality.

ACKNOWLEDGMENTS

The work is partially supported by the National Natural Science Foundation of Liao Ning (2022-MS-302, 2022-MS-303 and 2022-BS-218), the National Key Research and Development Program of China (2020YFB1707901), National Natural Science Foundation of China (Nos. U22A2025, 62072088, 62232007, 61991404), Ten Thousand Talent Program (No.ZX20200035), and Science and Technology Projects in Liaoning Province (No. 2023JH2/101300182).

REFERENCES

[1] Michiel H. M. Smid. Closest-point problems in computational geometry. In *Handbook of Computational Geometry*, pages 877–935. North Holland / Elsevier, 2000.

[2] Timothy M. Chan. Dynamic generalized closest pair: Revisiting eppstein's technique. In *3rd Symposium on Simplicity in Algorithms, SOSA 2020, Salt Lake City, UT, USA, January 6-7, 2020*, pages 33–37. SIAM, 2020.

[3] Sanguthevar Rajasekaran, Subrata Saha, and Xingyu Cai. Novel exact and approximate algorithms for the closest pair problem. In *ICDM 2017, New Orleans, LA, USA, November 18-21, 2017*, pages 1045–1050. IEEE Computer Society, 2017.

[4] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD 2000, May 16-18, 2000, Dallas, Texas, USA*, pages 189–200. ACM, 2000.

[5] Sergei Bespamyatnikh. An optimal algorithm for closest-pair maintenance. *Discret. Comput. Geom.*, 19(2):175–195, 1998.

[6] Mordecai J. Golin, Rajeev Raman, Christian Schwarz, and Michiel H. M. Smid. Randomized data structures for the dynamic closest-pair problem. *SIAM J. Comput.*, 27(4):1036–1072, 1998.

[7] Christian Schwarz, Michiel H. M. Smid, and Jack Snoeyink. An optimal algorithm for the on-line closest-pair problem. *Algorithmica*, 12(1):18–29, 1994.

[8] Yunjun Gao, Lu Chen, Xinhan Li, Bin Yao, and Gang Chen. Efficient k-closest pair queries in general metric spaces. *VLDB J.*, 24(3):415–439, 2015.

[9] Anil K. Jain, M. Narasimha Murty, and Patrick J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[10] David Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. *ACM J. Exp. Algorithmics*, 5:1, 2000.

[11] Alexandros Nanopoulos, Yannis Theodoridis, and Yannis Manolopoulos. C^2p: Clustering based on closest pairs. In *VLDB 2001, September 11-14, 2001, Roma, Italy*, pages 331–340. Morgan Kaufmann, 2001.

[12] Philipp Kranen, Ira Assent, Corinna Baldauf, and Thomas Seidl. Self-adaptive anytime stream clustering. In *ICDM 2009, Miami, Florida, USA, 6-9 December 2009*, pages 249–258. IEEE Computer Society, 2009.

[13] Dingming Wu, Erjia Xiao, Yi Zhu, Christian S. Jensen, and Kezhong Lu. Efficient retrieval of the top-kk most relevant event-partner pairs. *IEEE Trans. Knowl. Data Eng.*, 35(3):2529–2543, 2023.

[14] Pankaj K. Agarwal, Haim Kaplan, and Micha Sharir. Kinetic and dynamic data structures for closest pair and all nearest neighbors. *ACM Trans. Algorithms*, 5(1):4:1–4:37, 2008.

[15] Jianzhong Qi, Rui Zhang, Christian S. Jensen, Kotagiri Ramamohanarao, and Jiayuan HE. Continuous spatial query processing: A survey of safe region based techniques. *ACM Comput. Surv.*, 51(3), may 2018.

[16] Yuandong Wang, Hongzhi Yin, Lian Wu, Tong Chen, and Chunyang Liu. Secure your ride: Real-time matching success rate prediction for passenger-driver pairs. *IEEE Trans. Knowl. Data Eng.*, 35(3):3059–3071, 2023.

[17] Arneish Prateek, Arijit Khan, Akshit Goyal, and Sayan Ranu. Mining top-k pairs of correlated subgraphs in a large network. *Proc. VLDB Endow.*, 13(9):1511–1524, 2020.

[18] Jeffrey D. Ullman and Jonathan R. Ullman. Some pairs problems. In *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR@SIGMOD 2016, San Francisco, CA, USA, July 1, 2016*, page 8. ACM, 2016.

[19] Fangwei Wu, Xike Xie, and Jieming Shi. Top-k closest pair queries over spatial knowledge graph. In *Database Systems for Advanced Applications - 26th International Conference, DASFAA 2021, Taipei, Taiwan, April 11-14, 2021, Proceedings, Part I*, volume 12681 of *Lecture Notes in Computer Science*, pages 625–640. Springer, 2021.

[20] Dingming Wu, Yi Zhu, and Christian S. Jensen. In good company: Efficient retrieval of the top-k most relevant event-partner pairs. In *Database Systems for Advanced Applications - 24th International Conference, DASFAA 2019, Chiang Mai, Thailand, April 22-25, 2019, Proceedings, Part II*, volume 11447 of *Lecture Notes in Computer Science*, pages 519–535. Springer, 2019.

[21] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Trans. Database Syst.*, 35(3):20:1–20:46, 2010.

[22] Bolong Zheng, Xi Zhao, Lianggui Weng, Quoc Viet Hung Nguyen, Hang Liu, and Christian S. Jensen. PM-LSH: a fast and accurate in-memory framework for high-dimensional approximate NN and closest pair search. *VLDB J.*, 31(6):1339–1363, 2022.

[23] Michiel H. M. Smid. Maintaining the minimal distance of a point set in polylogarithmic time. In *Proceedings of the Second Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 28-30 January 1991, San Francisco, California, USA*, pages 1–6. ACM/SIAM, 1991.

[24] Sanjiv Kapoor and Michiel H. M. Smid. New techniques for exact and approximate dynamic closest-point problems. *SIAM J. Comput.*, 25(4):775–796, 1996.

[25] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. Algorithms for processing k-closest-pair queries in spatial databases. *Data Knowl. Eng.*, 49(1):67–104, 2004.

[26] Yifan Zhu, Lu Chen, Yunjun Gao, and Christian S. Jensen. Pivot selection algorithms in metric spaces: a survey and experimental study. *VLDB J.*, 31(1):23–47, 2022.
[27] Lu Chen, Yunjun Gao, Xuan Song, Zheng Li, Yifan Zhu, Xiaoye Miao, and Christian S. Jensen. Indexing metric spaces for exact similarity search. *ACM Comput. Surv.*, 55(6):128:1–128:39, 2023.
[28] Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. A parallel batch-dynamic data structure for the closest pair problem. In *37th International Symposium on Computational Geometry, SoCG 2021, June 7-11, 2021, Buffalo, NY, USA (Virtual Conference)*, volume 189 of *LIPIcs*, pages 60:1–60:16, 2021.
[29] Zhitao Shen, Muhammad Aamir Cheema, Xuemin Lin, Wenjie Zhang, and Haixun Wang. Efficiently monitoring top-k pairs over sliding windows. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 798–809, 2012.
[30] Muhammad Aamir Cheema, Xuemin Lin, Haixun Wang, and Wenjie Zhang. A unified framework for answering k closest pairs queries and variants. *IEEE Trans. Knowl. Data Eng.*, 26(11):2610–2624, 2014.
[31] Abdullah Mueen, Eamonn J. Keogh, Qiang Zhu, Sydney Cash, and M. Brandon Westover. Exact discovery of time series motifs. In *Proceedings of the SIAM International Conference on Data Mining, SDM 2009, April 30 - May 2, 2009, Sparks, Nevada, USA*, pages 473–484. SIAM, 2009.
[32] Jianzhong Qi, Rui Zhang, Christian S. Jensen, Kotagiri Ramamohanarao, and Jiayuan He. Continuous spatial query processing: A survey of safe region based techniques. *ACM Comput. Surv.*, 51(3):64:1–64:39, 2018.
[33] Kyriakos Mouratidis, Marios Hadjieleftheriou, and Dimitris Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 634–645. ACM, 2005.
[34] Baihua Zheng, Wang-Chien Lee, and Dik Lun Lee. Search continuous nearest neighbors on the air. In *1st Annual International Conference on Mobile and Ubiquitous Systems (MobiQuitous 2004), Networking and Services, 22-25 August 2004, Cambridge, MA, USA*, pages 236–245. IEEE Computer Society, 2004.
[35] Baihua Zheng, Wang-Chien Lee, and Dik Lun Lee. On searching continuous k nearest neighbors in wireless data broadcast systems. *IEEE Trans. Mob. Comput.*, 6(7):748–761, 2007.
[36] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of spatial queries in wireless broadcast environments. *IEEE Trans. Mob. Comput.*, 8(10):1297–1311, 2009.
[37] Rui Zhu, Bin Wang, Xiaochun Yang, Baihua Zheng, and Guoren Wang. SAP: improving continuous top-k queries over streaming data. *IEEE Trans. Knowl. Data Eng.*, 29(6):1310–1328, 2017.
[38] Yunjun Gao and Baihua Zheng. Continuous obstructed nearest neighbor queries in spatial databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 577–590. ACM, 2009.
[39] Yunjun Gao, Baihua Zheng, Wang-Chien Lee, and Gencai Chen. Continuous visible nearest neighbor queries. In *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, pages 144–155. ACM, 2009.
[40] Yunjun Gao, Baihua Zheng, Gencai Chen, Qing Li, and Xiaofa Guo. Continuous visible nearest neighbor query processing in spatial databases. *VLDB J.*, 20(3):371–396, 2011.
[41] Yunjun Gao, Baihua Zheng, Gang Chen, Chun Chen, and Qing Li. Continuous nearest-neighbor search in the presence of obstacles. *ACM Trans. Database Syst.*, 36(2):9:1–9:43, 2011.
[42] Kyriakos Mouratidis, Dimitris Papadias, Spiridon Bakiras, and Yufei Tao. A threshold-based algorithm for continuous monitoring of k nearest neighbors. *IEEE Trans. Knowl. Data Eng.*, 17(11):1451–1464, 2005.
[43] Jun Zhang, Manli Zhu, Dimitris Papadias, Yufei Tao, and Dik Lun Lee. Location-based spatial queries. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 443–454. ACM, 2003.
[44] Sarana Nutanong, Rui Zhang, Egemen Tanin, and Lars Kulik. The v*-diagram: a query-dependent approach to moving KNN queries. *Proc. VLDB Endow.*, 1(1):1095–1106, 2008.
[45] Haibo Hu, Jianliang Xu, and Dik Lun Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 479–490. ACM, 2005.
[46] Chuanwen Li, Yu Gu, Jianzhong Qi, Ge Yu, Rui Zhang, and Wang Yi. Processing moving knn queries using influential neighbor sets. *Proc. VLDB Endow.*, 8(2):113–124, 2014.
[47] Stefan Berchtold, Christian Böhm, Daniel A. Keim, and Hans-Peter Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona, USA*, pages 78–86. ACM Press, 1997.
[48] https://www1.nyc.gov/site/tlc/about/tlc-trip-record data.page.

Received January 2023; revised April 2023; accepted May 2023