Evaluating Nearest-Neighbor Joins on Big Trajectory Data

Yixiang Fang, Reynold Cheng, Wenbin Tang, Silviu Maniu, Xuan Yang

Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong
{yxfang, ckcheng, wbtang, smaniu, xyang2}@cs.hku.hk

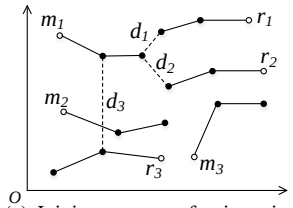
***Abstract*—Trajectory data are abundant and prevalent in systems that monitor the locations of moving objects. In a vehicle location-based service, the positions of vehicles are continuously monitored through GPS; each vehicle is associated with a trajectory that describes its movement history. In species monitoring, animals are attached with sensors, whose positions can be frequently traced by scientists. An interesting avenue to generate and discover new knowledge from these data is by querying them. In this paper, we study the evaluation of the join operator on trajectory data. Given two sets of trajectory data, M and R, our goal is to find, for each entity in M, its k nearest neighbors from R.**

Existing solutions for this query are designed for a single machine. Due to the abundance and size of trajectory data, such solutions are no longer adequate. We hence examine how this query can be evaluated in cloud environments. This problem is not trivial, due to the complexity of the trajectory structure, as well as the fact that both the spatial and temporal dimensions of the data have to be handled. To facilitate this operation, we propose a parallel solution framework using MapReduce. We also develop a novel bounding technique, which enables trajectories to be pruned in parallel. Our approach can be used to parallelize single-machine-based algorithms. We further study a variant of the join operator, which allows query efficiency to be improved. To evaluate the efficiency and the scalability of our approaches, we have performed extensive experiments on large real and synthetic datasets.

I. Introduction

In emerging systems that manage moving objects, a tremendous amount of *trajectory data* is often produced. In a location-based service (LBS), for instance, the positions of mobile phone users or vehicles are constantly captured by GPS receptors and mobile base stations [1], [2]. The location information constitutes a trajectory, which depicts the movement of an entity in the past. In natural habitat monitoring, scientists obtain location information of wild animals by attaching sensors to them. This movement history information, or trajectory data, facilitates the understanding of the animals' behaviours [3]. Figure 1(a) illustrates six trajectories, each of which is constructed by connecting three recorded locations.

Due to the increasing needs of managing trajectory data, the study of *trajectory databases* has recently attracted a lot of research attention [3]. One of the fundamental queries for this database is the *join* [4]–[6]. Given two sets M and R of trajectory objects, a join operator returns entity sets from M and R, that exhibit proximity in space and time.



(a) Joining two sets of trajectories

	k nearest neighbors
m_1	r_1, r_2
m_2	r_3, r_2
m_3	r_2, r_1

(b) Join results

Fig. 1. Illustrating the k-NN join (k=2, $[t_2, t_3]$).

To illustrate this query, let us consider Figure 1(a), where two sets of trajectory objects, namely M=$\{m_1, m_2, m_3\}$ and R=$\{r_1, r_2, r_3\}$, are shown. Each trajectory is constructed by connecting the locations collected at time instants t_1, t_2 and t_3, where $t_1 < t_2 < t_3$. For each trajectory, the small circle represents the position recorded at t_1. The result of joining M and R is demonstrated in Figure 1(b). For each object $m_i \in M$, the two counterparts in R that are the nearest neighbors of m_i in $[t_2, t_3]$ are returned. In this paper, we adapt the *k-nearest neighbor* metric [5], [6], as the joining criterion of M and R. That is, the k objects in R that have the shortest distances from each object in M are returned, by adopting the *closest-point-of-approach* [4]. In this example, within the time interval $[t_2, t_3]$, the 2-NN of m_1 is r_1 and r_2.

The trajectory join query can be used in a wide range of applications, including business analysis, military applications, celestial body relationship analysis, search and rescue missions, and computer gaming [6]–[8]. Let us consider two competitor companies that provide flights in the same geographical area. Let A and B be the sets of flight routes of these two companies. By joining A and B, we can retrieve for each route $a \in A$, the k routes in B that were the closest to a in a specific time interval. These results could be further analyzed by the company that manages A and help her to answer questions like: Is there any plane in B that flew very close to A's flights and cause safety concerns? Is there any route of B that resembles a, and charges a lower fare? In military applications, consider the two sets C and D of trajectories for military units (e.g., soldiers, vehicles and tanks) belonging to two rival countries. By joining C and D, the k entities in D closest to those of C can be evaluated. This is useful for the C's army to study the movement patterns of D, and study whether D is performing inspection on C, or planning a military action.

Despite the usefulness of trajectory joins, evaluating them is

not trivial. A simple solution is to evaluate a k-NN query for every object in M. However, since a trajectory object describes the movement of points in space and time, its data structure can be complex and expensive to handle. The problem is worsened when the sizes of the trajectory object sets to be joined are large. To evaluate joins on large trajectory datasets efficiently, researchers have previously studied fast algorithms and data structures [4]–[6]. However, these approaches run trajectory joins on a single machine only, whose computation, memory, and disk capabilities are limited. As discussed before, extremely large trajectory data have become increasingly common. Two trajectory datasets [1], [2], for instance, consist of over one billion location values. For evaluating joins on these large data, a single machine is no longer sufficient. In this paper, we study efficient trajectory join algorithms in parallel and distributed environments. We choose the MapReduce as the platform for our study, since it provides decent scalability and fault tolerance for very large data processing [9].

Designing trajectory join algorithms on MapReduce is technically challenging. This is because MapReduce is a shared-nothing architecture. Existing single-machine solutions often rely on an index (e.g., R-tree) built on top of the whole dataset (e.g., [5], [6]). As discussed in [10], [11], constructing and maintaining an index in MapReduce can be costly. In this paper, we develop a solution framework that exploits the shared-nothing architecture, without using an index. We first partition the given trajectories of M and R into "sub-trajectories", which are distributed to different computation units. For each partition of sub-trajectories, we develop a bounding technique called the *time-dependent bound* (or *TDB* in short). The TDB is a time-dependent circular region containing the (candidate) objects in R, which can be the k nearest neighbors of objects in M, in the same partition. Based on the TDB, we retrieve R's candidates, and join them with M's sub-trajectories. The join results of the partitions are finally merged.

Our solution can easily adopt single-machine join algorithms in its framework. In the paper, we will study how our approach parallelizes the execution of two single-machine solutions. Moreover, as we will discuss, the TDB is a function of time, and it changes according to the positions of the objects involved. While computing a TDB is not straightforward, we show that it is possible to develop a theoretically efficient algorithm to evaluate the TDB's in different partitions in parallel. The effort of developing TDB is justified by our experiments, which show that TDB significantly reduces the number of candidates to be examined.

We further propose two methods to optimize our join algorithm. First, we enhance the load balancing aspect of our solution, by distributing the trajectory objects to computing units in a more uniform manner. Second, we study a variant of the k-NN join, called (h, k)-NN join, which only returns h objects in M, together with their k-NN in R, under some monotonic aggregate function (e.g., min, max, sum or avg). As we will explain, this represents the h sets of "most important" k-NNs in the join of M and R. We propose a pruning technique for (h, k)-NN join. As shown in our experiments for real and synthetic data, our algorithm for the (h, k)-NN join is much faster than its k-NN join counterpart.

The rest of this paper is organized as follows. In Section II, we review the related work. Section III discusses the background of our solution. In Section IV we study the framework of our solution. In Sections V and VI, we present the detailed solution of the k-NN join. In Section VII, we present an efficient algorithm to evaluate (h, k)-NN join. Section VIII discusses the experimental results. We conclude in Section IX.

II. Related Work

A substantial amount of research on nearest neighbor query for trajectory objects has been performed. In [5], four types of queries have been studied, using R-trees. Our studied query is an extension of one of these four queries, i.e., given a trajectory object and a time interval return the nearest neighbour during this time. The continuous version of nearest neighbour queries has also received significant research attention. In [12], the nearest neighbor of every point on a line segment has been investigated, while [13] studied concurrent continuous spatio-temporal queries and [8] studied the k nearest and reverse k nearest neighbor queries on trajectory objects. The difference between our work and the above is that – for a given query trajectory object – we wish to return k trajectory objects whose distances to the query are minimal *at some particular time instances*, while the above studies focus on returning the k nearest neighbors *at every time instance*.

There are also many studies on join operation for trajectory objects. In [4], an adaptive join algorithm is proposed for closest-point-of-approach join, which is based on sweep line algorithm [14]. Given two trajectory objects, their minimum distance is defined to be achieved at their closest point. Also in [6] a broad class of trajectory join operations are studied, including trajectory distance join and k-NN join.

However, all these join algorithms are designed to be executed on a single machine, and hence are inefficient on large datasets. A natural way to extend them for handling large-scale data is to use parallel computing on a cluster of machines. There exist a few parallel computing paradigms including MapReduce [9], Pregel, Spark and Shark [15]. Out of these, MapReduce is one of the most widely used and performs best for batch processing queries – such as joins – and we study answering k-NN join queries using the MapReduce paradigm here. As we will discuss, the naive way to extend single-machine join algorithms for MapReduce is not scalable and efficient enough for large-scale trajectory objects due to its high computational cost.

Recently, many different kinds of join operations have been studied using MapReduce. For example, in [16] the set-similarity join is answered efficiently using MapReduce, in [11] divide-and-conquer and branch-and-bound algorithms are developed for answering top-k similar pairs using MapReduce, and in [17] the multi-way theta-join query is studied from a cost-effective perspective. In [18] efficient algorithms for k-NN join are presented using MapReduce, but they mainly returns approximate join results for sets of points, while our

k-NN join returns the exact result. [19] design an effective mapping mechanism that exploits pruning rules for distance filtering. However, since they do not deal with the temporal dimension, it is not clear how they can be applied to the data of trajectory objects.

III. Preliminaries

In this section we formally introduce the data model, problem definitions, single-machine solutions, the MapReduce framework, and a basic parallel solution using MapReduce.

A. Data Model

For ease of presentation, we consider in the following trajectory objects – or *trajectories* – in a $d \times d$ 2-D space. Note, however, that our methods can easily be applied for multi-dimensional space. Table I summarizes the symbols used in this paper.

Definition 1: A trajectory tr of an object is a tuple composed of the object's id and a list of locations $(q(t_1), q(t_2), \cdots, q(t_l))$. Each point $q(t)$ is represented by a triple (x, y, t), where x and y are the positions along x and y coordinates, and t is the timestamp of this location.

We denote the timestamps of the first and last points of tr as $tr.s$ and $tr.e$ respectively. We assume that the trajectory object moves along the straight line segment $\overline{q(t_i)q(t_{i+1})}$ between any two consecutive points $q(t_i)$ and $q(t_{i+1})$ with constant speed, in line with previous work [4], [6].

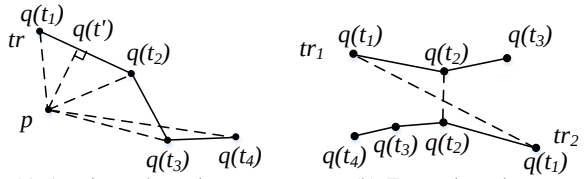
To evaluate k-NN queries on such trajectories, we first define the associated notions of *distance* between trajectories.

Definition 2: The minimum distance between a point p and a line segment $\overline{q(t_i)q(t_{i+1})}$, is defined as:

$$MinDist\left(p, \overline{q\left(t_i\right) q\left(t_{i+1}\right)}\right) = \min_{q \in \overline{q(t_i)q(t_{i+1})}} |p, q|, \quad (1)$$

where q is a point lying on the line segment $\overline{q(t_i)q(t_{i+1})}$, and $|p, q|$ is the Euclidean distance between points p and q.

Without loss of generality, our algorithm can be easily extended for other trajectory models [3] and distance measures such as network distance, Manhattan distance, etc.



(a) A point and a trajectory (b) Two trajectories

Fig. 2. Examples of minimum and maximum distances

Definition 3: The minimum distance between a point p and a trajectory tr with l points is defined as:

$$MinDist(p, tr) = \min_{1 \leqslant i \leqslant l-1} MinDist\left(p, \overline{tr.q\left(t_i\right) tr.q\left(t_{i+1}\right)}\right). \quad (2)$$

Similarly, we can define the maximum distance $MaxDist(p, \overline{q(t_i)q(t_{i+1})})$ between a point p and a line segment $q(t_i)q(t_{i+1})$, and the maximum distance $MaxDist(p, tr)$ between a point p and a trajectory tr.

TABLE I
Summary of Notations

Notation	Meaning
D	a $d \times d$ data space
$M(R)$	a set of trajectory objects
$m(r)$	a trajectory object from $M(R)$
tr	a trajectory
$tr.id$	the id of the trajectory object whose trajectory is tr
$tr.q(t_i)$	a point of tr whose time instance is t_i
$tr.s, tr.e$	the start and end time instances of tr
l	the total number of points in tr
T	the number of temporal partitions
N	the number of spatial partitions
H	the number of trajectory groups after hashing
p_i	the central point of i-th spatial partition
Tr_i^M	trajectories in i-th grid generated by objects from M
C_i^R	a set of candidate trajectories from R for Tr_i^M
G_i^M	a group of trajectories from M whose hash values are i

Example 1: In Figure 2(a), we can easily observe that $MinDist(p,\overline{q(t_1)q(t_2)})$ = $|p, q(t')|$, $MaxDist(p,\overline{q(t_1)q(t_2)})$ = $|p, q(t_2)|$, $MinDist(p,tr)$ = $|p, q(t')|$ and $MaxDist(p,tr)$ = $|p, q(t_4)|$.

Now we have all the ingredients to define what nearest neighbour means in the context of trajectories:

Definition 4: The minimum distance between two trajectories tr_i and tr_j is defined as:

$$MinDist(tr_i, tr_j) = \min_{t \in \Delta t} |tr_i.q(t) - tr_j.q(t)|, \quad (3)$$

where $\Delta t = [tr_i.s, tr_i.e] \cap [tr_j.s, tr_j.e]$.

In other words, the minimum distance between two trajectory objects is the minimum distance between their trajectories. Usually, to compute the minimum distance between two trajectory objects, we have to enumerate and compute the minimum distance between each pair of line segments from their trajectories, for the time intervals which intersect. Since we assume that objects move along straight lines between consecutive points, the distance between each pair of line segments can be formulated as a function of time t [4], i.e., $d(t)^2$=$at^2 + bt + c$, where a, b and c are parameters dependent on their velocities and initial positions. Thus their minimum distance is the minimum value of this function during their intersected time interval. Similarly, we can define the maximum distance $MaxDist(tr_1, tr_2)$ between two trajectories tr_1, tr_2.

Example 2: In Figure 2(b), we can easily observe that $MinDist(tr_1, tr_2)$ = $|tr_1.q(t_2), tr_2.q(t_2)|$ and $MaxDist(tr_1, tr_2)$ = $|tr_1.q(t_1), tr_2.q(t_1)|$.

Definition 5: Given a trajectory object m and a set of trajectory objects R, the k $nearest$ $neighbors$ of m are the k objects from R, whose minimum distances with m are the smallest.

Given a trajectory object m and the k minimum distances d_1^m, $\cdots$, d_k^m to its k nearest neighbors, we can define any monotonic aggregate function f (e.g., maximum, minimum, sum or average of the k distance values [20]) on m. Without loss of generality, in this paper we define $f(m)$ as the

maximum value of these k distance values:

$$f(m) = \max_{1 \le i \le k} d_i^m. \quad (4)$$

B. Problem Definitions

We now formally define the problems studied in this paper, i.e., k-NN and (h, k)-NN trajectory joins:

Problem 1: Given sets M and R of trajectories, an integer k and a query time interval $[t_s, t_e]$, the **k-NN join query** returns k nearest neighbors from R for each trajectory object in M during the query interval.

Problem 2: Given sets M and R, two integers h, k, a monotonic aggregation function f, and a time interval $[t_s, t_e]$, the **(h, k)-NN join query** returns h objects of M, having the smallest values of f on their k nearest neighbors.

A simple way to extend the methods in [18], [19] for our k-NN join is to sample some points from trajectories and then answer the join queries on these points using these algorithms We call this approach **Sampling-based approach**. However, this increases the time complexity significantly. For example, consider two objects with their trajectories, each of which has l points. To compute their minimum distance the time complexity is $O(l)$, since we only need to consider pairs of line segments starting from the first points. But, if we sample l points from each of them, we need to consider $l \times l$ pairs of points and thus the time complexity is $O(l^2)$. Furthermore, the answers may be wrong. Consider an extreme case where the overall time intervals of M and R have no intersection. If we perform k-NN join on them, the result is null. While if we sample two sets of points from them respectively, and then join them using spatial point join algorithms, we can find erroneous k nearest neighbors of each object.

C. Single-machine Solutions

We now discuss two single-machine solutions for solving k-NN joins, namely *brute force* (**BF**) and *sweep line* (**SL**).

Brute force: This basic method simply uses nested loops to solve the join. It first selects all the sub-trajectories appearing in $[t_s, t_e]$. Then it computes the minimum distance between each pair of trajectory objects – one from M and the other one comes from R – and selects the k nearest neighbors for each object of M. This method is very costly, as it potentially needs to process the entire Cartesian product of the sets M and R.

Sweep line: The intuition behind this method is the following: when computing the minimum distance between two trajectory objects, their trajectories must overlap in some time intervals. The sweep line method, e.g., [4], [14], sorts the timestamps of all the points generated by objects from R. For the trajectory of each object from M, it sweeps along the temporal dimension and computes their minimum distances using a dedicated data structure. As the sweep line progresses, the minimum distances to all the objects from R are computed. Finally, the k nearest neighbors are selected. The main difference between BF and sweep SL is that, when two trajectories do not have temporal intersection, SL will not consider them as a potential pair.

D. MapReduce Framework

MapReduce [9] is a popular paradigm for processing large data on share-nothing distributed clusters. It consists of two functions `map` and `reduce`. The `map` function takes a key-value pair and outputs a list of key-value pairs, i.e., $\texttt{map}(k_1, v_1) \rightarrow \textit{list}(k_2, v_2)$. The `reduce` function takes a list of records with the same key as input and outputs a list of key-value pairs, i.e., $\texttt{reduce}(k_2, \textit{list}(v_2)) \rightarrow \textit{list}(k_3, v_3)$. In a MapReduce job, when all the `map` functions have finished, all the intermediate results are grouped and shuffled to the `reduce` functions. By default, each map task processes a split of data with the size equal to the block size of its distributed file system, HDFS. In a MapReduce job, the number of map tasks equals to the number of splits, while number of reduce tasks can be set by the users.

E. A Basic Parallel Solution Using MapReduce (BL)

We now introduce a basic parallel solution of k-NN join using MapReduce, i.e., **BL**, which has two MapReduce jobs. In the first job, it divides objects in M and R into a list of disjoint subsets randomly in the `map()`, and then joins each pair of subsets using a single-machine solution – e.g., BF or SL – in the `reduce()`. Since the k nearest neighbors of an object may be in several subsets of R, a second job where the k nearest neighbors are selected, from the results of the first MapReduce job. The main drawback of BL is its high computational cost, since each pair of trajectories from M and R needs to be enumerated.

IV. Solution Framework

To overcome the drawback of BL, we propose a new solution framework, which allows pruning of trajectories during the querying process. It consists of two phases, namely **preprocessing phase** and **query phase**. The preprocessing phase needs to be conducted for only once, while the query phase is invoked once a k-NN join query arrives. Figure 3 illustrates its workflow.

In the preprocessing phase, we partition trajectories in space using equal-sized grids. In the query phase, we propose a four stage approach to answer a join query:

1) **Sub-trajectory generation.** In this stage, we find all the sub-trajectories appearing in $[t_s, t_e]$. Then we collect relevant statistics from each spatial partition. We also select trajectories, which serve as *anchor trajectories*, if this partition of trajectories is generated by objects from R. We denote the set of sub-trajectories generated by objects from $M(R)$, in the $i(j)$-th grid, as $Tr_i^M(Tr_j^R)$.
2) **Computing bounds.** In this stage, we compute the *time-dependent upper bound* (TDB) of Tr_i^M using the collected statistics and the anchor trajectories.
3) **Finding candidates.** For each partition Tr_i^M, we use its TDB to find a set, C_i^R, of *candidate trajectories* generated by objects from R. The candidate trajectories are sets of trajectories – ideally, minimal in size – which must contain *all* the k nearest neighbors of objects in M which cross the i-th spatial grid.

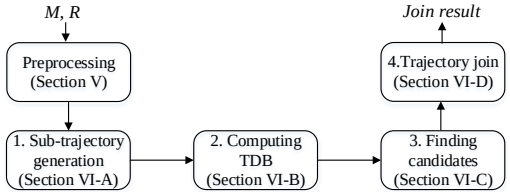


Fig. 3. The workflow of our framework

4) **Trajectory join.** For each partition Tr_i^M, we join it with C_i^R using a single-machine algorithm, (e.g., BF or SL).

We denote the above approach as **GN**. Even though GN achieves greater efficiency by using TDB, it may not able to achieve good load balance, due to the skewness of the distribution of the locations of the trajectory objects. By using uniform partitioning of the space, we may encounter grids which contain many objects while others contain very few objects. To achieve good load balance in the query phase, we improve the load balance of GN under the same framework, by using a load balance strategy, which redistributes all the objects using some hash functions. We denote this new approach as **GL**.

The preprocessing phase and each stage of the query phase are computed using a MapReduce job, in a sequential workflow. The purpose of the `map()` and `reduce()` for each stage is summarized in Table II. We discuss the preprocessing in Section V. We detail the stages of GN and explain how GL improves the load balance in Section VI.

TABLE II
DETAILS OF EACH MAPREDUCE JOB

Phase/Stage	Function	Main work
Preprocessing	Map	conduct temporal partition
	Reduce	conduct spatial partition
Stage 1	Map	generate sub-trajectories
	Reduce	collect statistics and anchor trajectories
Stage 2	Map	compute $MaxDist(p_i, achTr)$
	Reduce	compute the TDB, i.e, $u_i(t)$, of Tr_i^M
Stage 3	Map	find candidates of Tr_i^M
	Reduce	collect the candidates of Tr_i^M
Stage 4	Map	trajectory join
	Reduce	select k nearest neighbors for each object

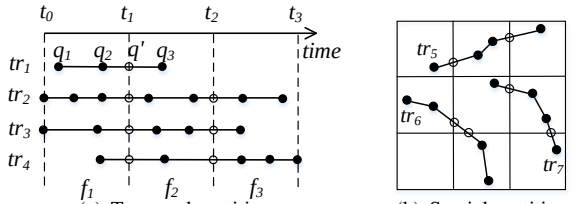
V. THE PREPROCESSING PHASE

The preprocessing phase is mainly used to partition the data, both temporally and spatially. Since the length of the query time interval is usually less than the time interval of the entire data, we propose to partition the trajectories using equal-length time intervals. Spatially, the trajectories are partitioned using equal-sized grids. Note that we only need to conduct temporal partition for BL, since it does not use grid.

1. Temporal partition. Suppose the overall time domain of the dataset is $[T_s, T_e]$ and the number of intervals is T, then we define a list of intervals as: $[T_s, T_s + \Delta t]$, $[T_s + \Delta t, T_s + 2\Delta t]$, $\cdots$, $[T_e\text{–}(T\text{–}1) \cdot \Delta t, T_e]$, where Δt=$(T_e\text{–}T_s)/T$. Each trajectory is then split into a list of sub-trajectories according to these time intervals.

2. Spatial partition. Suppose the number of spatial partitions is N, then the size of each grid is $(d/\sqrt{N})\times(d/\sqrt{N})$. To partition a trajectory tr, we first compute its intersection points with the grids, and then insert them into tr, if they are not already points of tr. Finally, tr is partitioned into a group of sub-trajectories, each of which is in its corresponding grid. To facilitate the following queries, for each sub-trajectory $subTr$, we assign it a key "$M\_p_i.x\_p_i.y$", if it is generated by an object from M; otherwise we assign it a key "$R\_p_i.x\_p_i.y$", where p_i is the central point of the grid containing $subTr$.

We pre-process the trajectories using a MapReduce job, where the temporal and spatial partitions are performed in the `map()` and `reduce()` respectively. Due to space reasons, the detailed implementation is presented in the Appendix. In the output of this MapReduce job, we generate trajectories and output them into the same file if they occur in the same time interval. Hence, we obtain T files $f_1, f_2, \cdots, f_T$, where the time intervals of trajectories in f_i are in $[T_s+i\cdot\Delta t, T_s+(i+1)\cdot\Delta t]$. The time and space complexities of `map()` are $O(l+T)$, since a trajectory is split into at most T sub-trajectories. Both the time and space complexities of `reduce()` are $O((|M|+|R|)(l+N))$.



(a) Temporal partition (b) Spatial partition

Fig. 4. Examples of preprocessing

Example 3: In Figures 4(a) and 4(b), the solid points represent the original points and the hollow points represent the inserted points. Let us take tr_1 from Figure 4(a) as an example. Since q_2 and q_3 are in two consecutive intervals, a new point q' is inserted. Finally, this trajectory is split into two sub-trajectories, where the first one contains the first 3 points while the second one contains the last 2 points. Similarly, in Figure 4(b), the trajectories are partitioned using the grids.

VI. THE QUERY PHASE

In this section, we present details of the four MapReduce stages: sub-trajectory generation, computing TDB, finding the candidate trajectories, and the final trajectory join.

A. Sub-trajectory Generation

When a join query is asked, we first need to locate the files relevant for the join operation and then launch a MapReduce job with these files as an input. In the mapper we retrieve all the sub-trajectories which intersect with $[t_s, t_e]$. In the reducer, we collect some statistics and select anchor trajectories, which are used for computing the TDB for each partition.

We first discuss the sub-trajectory generation stage of GN in Algorithm 1, and then discuss the needed modifications for GL. We first locate files f_c,f_{c+1}, $\cdots$, $f_{c'}$ (line 2), which have time intervals overlapping with $[t_s, t_e]$, where c=$\lfloor t_s/\Delta t \rfloor$ and c'=$\lceil t_e/\Delta t \rceil$.

Map. The input of `map()` is a pair (k_1, v_1), where v_1 is a trajectory and k_1 is its key, i.e., "$M\_p_i.x\_p_i.y$" or

"$R\_p_j.x\_p_j.y$". We extract the sub-trajectory which appears in $[t_s, t_e]$ (line 4), and output it (line 5).

Reduce. We first parse the set label L, which can be either M or R from k_2 (line 7). Then we collect some statistic information and anchor trajectories (line 8-11). We next detail how the statistics and anchor trajectories are collected.

Algorithm 1 Stage 1: Sub-trajectory generation

```
1: procedure MAP-SETUP(t_s, t_e)
2:     locate files f_c, f_{c+1}, ..., f_{c'};
3: procedure MAP(k_1, v_1)
4:     subTr ← v_1.subTraj(t_s, t_e);
5:     k_2 ← k_1; v_2 ← subTr; OUTPUT(k_2, v_2);
6: procedure REDUCE(k_2, v_2)
7:     parse the set label L from k_2; Tr_i^L ← v_2;
8:     compute sT(Tr_i^L), eT(Tr_i^L), maxU(Tr_i^L);
9:     if L = "R" then
10:        achList ← SELECTANCHOR(Tr_i^L, sT(Tr_i^R), k);
11:    output maxU(Tr_i^L), sT(Tr_i^L), eT(Tr_i^L), achList, Tr_i^L;
```

In terms of statistics collected, we first collect the minimum start time and maximum end time of all the trajectories in Tr_i^L, as shown in Equation (5). Moreover, for all the trajectories, we compute their maximum distances to the central point p_i, and collect their maximum value, as shown in Equation (6).

$$sT(Tr_i^L) = \min_{tr \in Tr_i^L} tr.s, \quad eT(Tr_i^L) = \max_{tr \in Tr_i^L} tr.e \tag{5}$$

$$maxU(Tr_i^L) = \max_{tr \in Tr_i^L} MaxDist(p_i, tr) \tag{6}$$

To facilitate our computations, we now introduce a new data structure, namely spatiotemporal-unit (abbreviated as *st-unit*). A **st-unit** is a triple ($dist$, $startT$, $endT$), where $dist$ is a distance value, $startT$ and $endT$ are the start and end time of a time interval $[startT, endT]$. For each partition Tr_i^L, we can form a st-unit u=($maxU(Tr_i^L), sT(Tr_i^L), eT(Tr_i^L)$), where $maxU(Tr_i^L)$ bounds the maximum distances from p_i to all the trajectories during this time interval.

In addition, if L is R, we need to collect some anchor trajectories from Tr_j^R (line 9-10). For any time instance, if there are more than k trajectory objects in the grid of this partition, then we only collect k anchor trajectories; otherwise, all of them are collected. Even though any k trajectories can be selected as anchor trajectories, we found it is better to select trajectories which are closest to the central points of the grids. We propose a heuristic algorithm to select the anchor trajectories. We discuss the details in the Appendix.

Example 4: Figure 5 gives an example of computing $maxU(Tr_i^M)$ of Tr_i^M, which contains 2 trajectories. Figure 6 gives an example of 6 trajectories in Tr_j^R (k=2). For each trajectory tr, we form a st-unit ($MaxDist(p_j, tr)$, $tr.s$, $tr.e$). We can observe that at any time instance, there are at least 2 trajectory objects in the j-th grid. We collect the trajectories which correspond to the wide lines in the figure as anchor trajectories.

GL. To balance the workload for the subsequent jobs in the workflow, we use a hash function to redistribute all the objects in M and R each into H disjoin groups, where H

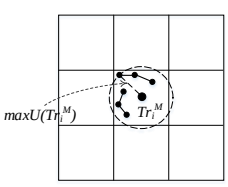


Fig. 5. Statistic collection

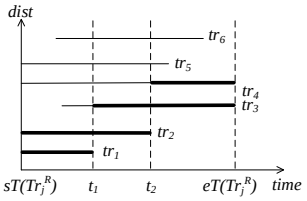


Fig. 6. Anchor trajectory selection

can be set as the integer multiple of the maximum number of parallel map tasks running in a cluster. The details of setting H are discussed in the Appendix In general, any hash function (e.g., [21]) which can partition objects into groups that keep the same distribution of objects as the overall distribution can be adopted here. In our experiments, since the identifiers of trajectory objects in the dataset are uniformly distributed, we simply hash the objects according to their identifiers, i.e., the hash function is a simple modulo function $hash(tr)$=$tr.id\%H$. After hashing, each group has the same expected number of objects. We denote the trajectories of objects from M(R) in the i(j)-th group as G_i^M (G_j^R).

In the sub-trajectory generation stage of GL, all the steps are the same with Algorithm 1, except we need to redistribute the trajectories using the hash function in the `reduce()`. Specifically, we hash each $tr \in Tr_i^L$ and assign it a key, which is a combination of its set label L and hash value $hash(tr)$. In the output of this MapReduce job, we output trajectories according to their keys, hence obtaining $2\times H$ files, corresponding to $G_1^M, \cdots, G_H^M, G_1^R, \cdots, G_H^R$. Note that in each file, trajectories from a same `reduce()` are collected together and stored in a single line.

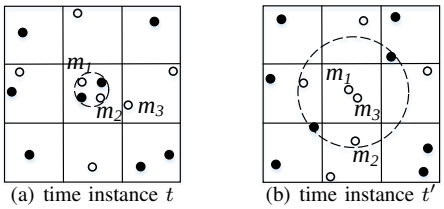
The time and space complexities of `map()` are $O(\log l)$ and $O(l)$ respectively, since we can use binary search to find the sub-trajectory. The computation of $sT(Tr_i^L)$, $eT(Tr_i^L)$ and $maxU(Tr_i^L)$ can be performed linearly by scanning all the trajectories. The time complexity of finding anchor trajectories is $O(|Tr_i^L|^2 l)$, since we need to find one from Tr_j^R each time. Thus, the overall time and space complexities of `reduce()` are $O(|Tr_i^L|^2 l)$ and $O(|Tr_i^L| l)$ respectively.

B. Computing TDB

We first introduce the intuitions behind the time-dependent bound, which is the most important part for the efficiency of the algorithms. For objects whose sub-trajectories are in Tr_i^M, we want to compute an upper bound on the area containing all their k nearest neighbors. However, since the objects may move arbitrarily, the area containing their k nearest neighbors may be large in some time intervals, and small in other time intervals. So it is nontrivial to compute a tight upper bound which holds in *all time intervals*.

Example 5: Figure 7 gives an example (k=2). The hollow points denote the objects from M and the solid points denote the objects from R. Let us consider the upper bound of the central grid partition. At time instance t in Figure 7(a), the k nearest neighbors of objects, i.e., m_1 and m_2, in this partition are in a small area. But at another time instance t' in Figure 7(b), the k nearest neighbors of objects, i.e., m_1 and m_3, in

this partition are in a large area.



(a) time instance t (b) time instance t'

Fig. 7. The upper bound changes with time

To alleviate this issue, we propose to compute a TDB for each partition Tr_i^M, which has the property that its value varies with time. For any object at t, with sub-trajectory $tr \in Tr_i^M(t \in [tr_s, tr_e])$, the distances to its k nearest neighbors must be less than this bound. To compute the TDB of Tr_i^M, we first compute the upper bound distance of the central point of the i-th grid, hereafter denoted as p_i, to its k nearest neighbors from R. It is easy to see that this also constitutes a TDB. Then, since the grid of Tr_i^M bounds the area containing all the objects whose sub-trajectories are in Tr_i^M, we can compute the TDB of Tr_i^M by considering the TDB of p_i and the area of this grid.

For a given time instance, in order to compute the upper bound distances from p_i to its k nearest neighbors, we can use spatial indexes for fast access, e.g., R-trees. As discussed before, however, is is difficult to maintain such indexes using MapReduce [10], [11], which is a share-nothing framework. Another way is to enumerate all the distances between p_i and all the objects in R, and then compute a bound. However, the computational cost is very high, since $|R|$ can be very large.

To address these problems, we propose to choose only k trajectories for each time instance from each partition Tr_j^R, called *anchor trajectories*. Then we can compute the TDB of p_i using these anchor trajectories. In summary, the steps of computing the TDB of Tr_i^M are as follows.

1) We first compute TDB of p_i, i.e., $v_i(t)$, by using the maximum distances from p_i to all the anchor trajectories.
2) We compute the TDB of Tr_i^M by using the TDB of p_i and $maxU(Tr_i^M)$.

In the corresponding MapReduce job, the `map()` computes the maximum distance from each anchor trajectory to each central point of the partitions, and the `reduce()` computes the TDB of Tr_i^M based on the maximum distances. Algorithm 2 gives the details of computing TDB of GN and GL.

Map. The input of `map()` is an anchor trajectory $achTr$. For each partition Tr_i^M, we first find $achTr$'s sub-trajectory in $[sT(Tr_i^M), eT(Tr_i^M)]$ (line 4-5). Then we compute its maximum distance to p_i (line 6), and output a pair, where the key is "$M\_p_i.x\_p_i.y$" and the value is a st-unit (line 7-8).

Reduce. After shuffling, st-units with the same key are sent to the same `reduce()`. We first compute the TDB of p_i using these st-units (line 11), resulting in a list of st-units sorted chronologically. Then we compute the TDB of Tr_i^M (line 12-13). To reduce the number of st-units in the TDB, we merge consecutive st-units if the lengths of their time intervals are too small (line 14). Specifically, we check each st-unit u, and if $|u.endT\text{-}u.startT| \leq \alpha$, where α is a small predefined parameter, then we merge it with its next st-unit u'. We update u=($max(u.dist, u'.dist)$,$u.startT$,$u'.endT$) and delete u' from the list. This process is iterated until the length of time interval of each st-unit is larger than α.

Algorithm 2 Stage 2: Computing TDB

```
1: procedure MAP(k1, v1)
2:     achTr ← v1, start ← 0, end ← ∞;
3:     for i ←1 to N do
4:         [start, end] ← [achTr.s, achTr.e] ∩ [sT(Tr_i^M), eT(Tr_i^M)];
5:         subTr ← achTr.subTraj(start, end);
6:         maxDist ← MAXDIST(subTr, p_i);
7:         k2 ← M_p_i.x_p_i.y, v2 ← (maxDist, start, end);
8:         OUTPUT(k2, v2);
9: procedure REDUCE(k2, v2)
10:    p_i ← k2, unitList ← v2;
11:    boundList ← COMPTDB(unitList, k);
12:    for each unit ∈ boundList do
13:        unit.dist ← unit.dist + maxU(Tr_i^M);
14:    boundList ← COMBINE(boundList, α);
15:    k3 ← k2, v3 ← boundList;
16:    OUTPUT(k3, v3);
```

Given a list of st-units, we now discuss the intuition to compute $v_i(t)$. For any time instance $t \in [sT(Tr_i^M), eT(Tr_i^M)]$, we first locate the st-units, whose time intervals contain t, and then rank these st-units based on their $dist$ values in ascending order. Then we select k st-units, whose $dist$ values are minimal. Since these k st-units correspond to k objects from R, the maximum value of these k $dist$ values is thus an upper bound of p_i at t. To make this process more efficient, we propose an efficient algorithm for computing $v_i(t)$ by dynamically maintaining a balanced binary tree [22], which is detailed in the Appendix.

Figure 8 gives an example (k=2) of computing $v_i(t)$. We can observe that for any time instance $t \in [sT(Tr_i^M), eT(Tr_i^M)]$, there are at least 3 st-units whose time intervals contain t. Then, our goal is to output some st-units as the TDB, which correspond to the border lines as shown in the figure. It is easy to observe that, given a time instance t, if there are many anchor trajectories close to p_i, then its upper bound distance tends to be small.

Since the TDB of p_i is represented by a list of st-units, it can also be rewritten as a piecewise function as follows:

$$v_i(t) = \begin{cases} d_{i,1}, & t \in [sT(Tr_i^M), t_{i,2}) \\ d_{i,2}, & t \in [t_{i,2}, t_{i,3}) \\ \cdots, & t \in [\cdots, \cdots) \\ d_{i,B}, & t \in [t_{i,B}, eT(Tr_i^M)] \end{cases} \quad (7)$$

where B is the number of st-units, $d_{i,b}$ is the upper bound distance, a constant value, when $t \in [t_{i,b}, t_{i,(b+1)})(1 \leq b \leq B)$. We call the time instances $sT(Tr_i^M)$, $t_{i,2}$, $\cdots$, $eT(Tr_i^M)$ *breakpoints*.

Lemma 1: Given a partition of trajectories Tr_i^M and the TDB of the central point p_i, $v_i(t)$, the TDB for all objects having sub-trajectories in Tr_i^M, to their k nearest neighbors

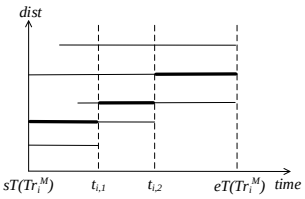


Fig. 8. Computing $v_i(t)$

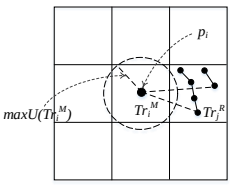


Fig. 9. Computing $u_i(t)$

from R at time instance t is

$$u_i(t) = maxU(Tr_i^M) + v_i(t), t \in [sT(Tr_i^M), eT(Tr_i^M)]. \quad (8)$$

Proof: **(Sketch)** Figure 9 illustrates the geometric intuition. We compute $u_i(t)$ by linking $maxU(Tr_i^M)$ with $v_i(t)$ using triangle inequality. The details are in the Appendix. ■

Since $v_i(t)$ is a piece-wise function, $u_i(t)$ is also a piece-wise function, whose value changes with time. We denote the maximum and minimum values of $u_i(t)$ as $max(u_i(t))$ and $min(u_i(t))$ respectively.

The time and space complexities of `map()` are $O(Nl)$ and $O(l)$ respectively, since we need to enumerate all the central points and anchor trajectories, which can consist of the entirety of R in worst case. The operations on the balanced binary tree including insert, delete and query can be completed in $O(\log|R|)$ and combing st-units can be completed linearly without extra space cost. Thus, the time and space complexities of `reduce()` are $O(|R|log|R|)$ and $O(|R|)$ respectively.

C. Finding Candidates

We now study how to find a set of candidate trajectories for join, C_i^R, for each partition Tr_i^M, i.e., to list all trajectories which may be in the k-NN of trajectories of M crossing Tr_i^M. Given two partitions Tr_j^R and Tr_i^M, we check whether the trajectories of Tr_j^R are the candidates of Tr_i^R in two sequential steps: *partition check* and *trajectory check*.

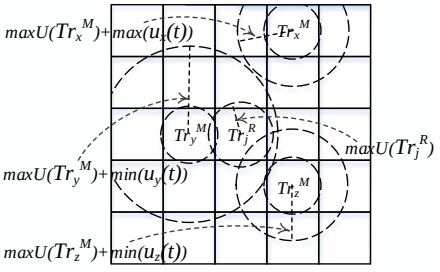


Fig. 10. Join candidate cases

Partition check. Given two trajectory partitions, Tr_i^M and Tr_j^R, we need to check whether the whole set of trajectories in Tr_j^R is the candidate of Tr_i, by checking three cases:

1) none of the them are join candidates of Tr_i^M,
2) all of them are join candidates of Tr_i^M, and
3) a part of them are join candidates of Tr_i^M.

Lemma 2: Given two partitions Tr_i^M and Tr_j^R, if

$$maxU(Tr_i^M) + max(u_i(t)) \leqslant |p_i, p_j| - maxU(Tr_j^R), \quad (9)$$

then all the trajectories in Tr_j^R belong to case 1, else if

$$maxU(Tr_i^M) + min(u_i(t)) \geqslant |p_i, p_j| + maxU(Tr_j^R), \quad (10)$$

then all trajectories in Tr_j^R belong to case 2. Otherwise, the trajectories belong to case 3.

Proof: **(Sketch)** In Figure 10, the cases between Tr_j^R with Tr_x^M, Tr_y^M and Tr_z^M are case 1, 2 and 3 respectively. We can prove each case using triangle inequality in the Appendix. ■

We use the above lemma to check cases 1 and 2 first. If none of them holds, we need to perform individual trajectory check as follows.

Trajectory check. To explain the trajectory check, we first introduce two supporting lemmas.

Lemma 3: Given a partition Tr_i^M and a trajectory object $r \in R$ whose trajectory is tr, the lower bound distance from r to objects which cross the partition Tr_i^M is:

$$w_i(tr) = \max\{0, MinDist(p_i, tr) - maxU(Tr_i^M)\}. \quad (11)$$

Proof: **(Sketch)** We consider the minimum distance between r and an arbitrary object crossing Tr_i^M, and then claim it is larger than $w_i(tr)$. The details are in the Appendix. ■

Lemma 4: Given a partition Tr_i^M and its TDB $u_i(t)$, for any trajectory object $r \in R$, whose sub-trajectory tr appears in $[t_{i,b}, t_{i,(b+1)})$, where $t_{i,b}$ and $t_{i,(b+1)}$ are two consecutive breakpoints of $u_i(t)$, if the $w_i(tr) < u_i(t)$, then tr is among the candidates of Tr_i^M.

The lemma follows directly from the bound in Lemma 3.

In the trajectory check step, we check each trajectory in Tr_j^R, and perform the following steps. We first collect all the breakpoints $t_{i,1}$, $t_{i,2}$, $\cdots$, $t_{i,B}$ of $u_i(t)$. Then, for each trajectory of Tr_j^R, we split it into a list of sub-trajectories according to the breakpoints, each of which appears in one time interval $[t_{i,b}, t_{i,(b+1)}](1 \leq b \leq B)$. For each sub-trajectory tr, we compute its lower bound distance to objects of Tr_i^M, i.e., $w_i(tr)$, using Lemma 3. Finally, by using Lemma 4, we can easily check whether it is a candidate of Tr_i^M.

We now detail the corresponding MapReduce job of GN, and then discuss the needed modifications for GL. The detailed corresponding pseudocode listings of GN are in the Appendix.

Map. The `map()` takes a partition of trajectories Tr_j^R as input. It enumerates each partition Tr_i^M, and checks which case they belong to. If they belong to case 2, then all the trajectories of Tr_j^R are candidates of Tr_i^M. Otherwise, for case 3, we split each trajectory of Tr_j^R into a list sub-trajectories according to the breakpoints of $u_i(t)$, and then check them one by one using Lemma 4.

Reduce. In the `reduce()`, we simply output the candidates of Tr_i^M, i.e., C_i^R.

The output of this MapReduce job is a list of files containing trajectories, one for each key (partition), for a total of N files.

GL. Each map task handles a single group G_j^R, in which each `map()` handles a subset Tr of trajectories from G_j^R, belonging to the same spatial partition. The MapReduce implementation principle is the same as that for GN, except the input of `map()` is Tr. Since different groups have the same expected size, GL achieves better load balance than GN.

We denote the input of `map()` as Tr, i.e., Tr_j^R or a subset of G_j^R. In the `map()`, we first need to enumerate all the partitions and check the cases. If none holds, we need to

consider trajectories one by one. Thus, the overall time and space complexities of `map()` are $O(|Tr|Nl)$ and $O(|Tr|l)$ respectively. Since we only need to output the input directly, the time and space complexities of `reduce()` are $O(1)$ and $O(|C_i^R|l)$ respectively.

D. Trajectory Join

Since we have found the set, C_i^R, of candidates for each partition Tr_i^M, we join it with C_i^R and then output the k nearest neighbors of each object crossing Tr_i^M. But the result is incomplete since an object may cross several grids. So for each object, we need to reselect k nearest neighbors finally.

We now detail the corresponding MapReduce job of GN, and then discuss the needed modifications for GL. The detailed pseudocode listings of GN are presented in the Appendix.

Map. The input of `map()` is a partition Tr_i^M. It joins Tr_i^M with the set C_i^R of corresponding generated candidates, using a single-machine algorithm. In this paper, we use BF or SL as discussed before, but any other single-machine trajectory join algorithms can be incorporated. Finally, we output a list of pairs, where the key is the object id and the value is a list of its k nearest neighbors with their minimum distances.

Reduce. The input of `reduce()` is an object with its k nearest neighbors computed from different partitions. We output k objects whose minimum distances are the smallest.

GL. The principle and implementation of GL is the same as in the finding candidates stage, i.e., each `map()` handles a subset of trajectories from G_i^M.

We denote the input of `map()` as Tr, i.e., Tr_i^M or a subset of G_i^M. Since we need to enumerate each pair of trajectories without extra space cost, the time and space complexities of `map()` are $O(|Tr||C_i^R|l)$. The time and space complexities of `reduce()` are $O(kN)$, since an object may go across at most N grids.

VII. The (h,k)-NN Join Algorithm

In this section, we study a variant of the k-NN join, the (h,k)-NN join and the needed adaptations to our framework.

A. Main Intuition

In the (h,k)-NN join, we wish to return h objects of M, having the smallest values of the function f on their their k nearest neighbors from R. We call these h objects *target objects*. We assume, without loss of generality that the aggregate function is *max*, i.e., Equation (4).

We propose to compute a new time-dependent upper bound, which is tighter than that of k-NN join. Recall that Equation (8) gives the TDB of Tr_i^M, which bounds the minimum distance for all the objects, crossing partition Tr_i^M, to their k nearest neighbors. In this equation, the left summand is $maxU(Tr_i^M)$ and the right summand is $v_i(t)$. Now since we only need to return h objects from M, the intuition is to try to make $maxU(Tr_i^M)$ smaller, so that the upper bound will be tighter and thus we achieve higher efficiency in pruning. Figure 11 shows the geometrical intuition.

To adapt the framework of k-NN join for the (h,k)-NN join, we need to perform the following adaptations. In the sub-trajectory generation stage, for all trajectory objects whose trajectories are in Tr_i^M, we select h nearest objects to p_i. In the computing TDB stage, we use the computed statistics on these h objects from each partition to derive a new TDB. Note that the new TDB only ensures that we can find h target objects with their k nearest neighbors, but it cannot ensure that we can return the k nearest neighbors of every object of M correctly. The finding candidates and trajectory join stages are the same with that of k-NN join. Finally, we select and output h target objects with their k nearest neighbors. The size of the results returned by a (h,k)-NN join query is $h \times k$, which is smaller than that of a k-NN join query, i.e., $|M| \times k$.

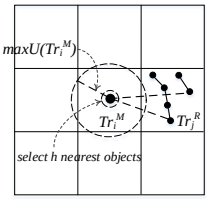


Fig. 11. Main idea

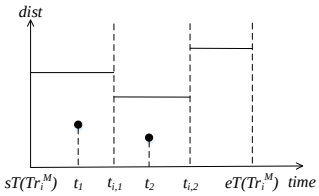


Fig. 12. Computing TDB

B. Implementation

Sub-trajectory generation. The `map()` is the same with that of k-NN join. The first steps of `reduce()` are the same with that of k-NN join. Then, if a partition of trajectories is generated by objects from M, i.e., Tr_i^M, we compute $MinDist(tr, p_i)$ for each $tr \in Tr_i^M$, and select h objects whose minimum distances are the smallest. For each of them, we form a triple (id, $dist$, $time$), where id is its identifier, $time$ is the time instance when the minimum distance $dist$ is achieved. Finally, we output these h triples.

Computing TDB. The `map()` is the same with that of k-NN join. In the `reduce()`, we first compute the TDB of p_i, i.e., $v_i(t)$, as that in k-NN join. By using triangle inequality, we can easily conclude that the distance from the object whose identifier is id to its k nearest neighbors is at most $dist + v_i(time)$. Thus, for each triple (id, $dist$, $time$) collected in previous stage, we update its $dist$ value to $dist + v_i(time)$. Finally, we output a list of h updated triples in the `reduce()`. Figure 12 gives an example, where $v_i(t)$ is a piece-wise function, and the collected triples are represented by black points at t_1 and t_2.

We collect the output triples of this MapReduce job and select h triples with different ids, having minimal $dist$ values. We denote these h triples as tp_1, tp_2, $\cdots$, tp_h. Then the minimum distance from h target objects to their k nearest neighbors is bounded by $\max_{1 \le j \le h} tp_j.dist$. Hence, the TDB of each partition Tr_i^M is:

$$u_i(t) = \max_{1 \le j \le h} tp_j.dist, \quad t \in [sT(Tr_i^M), eT(Tr_i^M)]. \quad (12)$$

In the k-NN join case, the maximum distances from all the objects to the central points are used for computing TDB. While in the (h,k)-NN join case, it only uses the minimum distances of h objects from each partition.

Finding candidates and trajectory join. The `map()` and `reduce()` are the same with that of k-NN join.

Finally, for each object of M, we compute the value of the aggregate function on the distances to its k nearest neighbors. Then we output h objects – which have the h smallest values of the aggregate function – with their k nearest neighbors. The time and space complexities of `map()` and `reduce()` in each stage are the same with that of k-NN join.

VIII. Results

We now present the experiment setting, results on k-NN joins and (h, k)-NN joins in the following three sections.

A. Setup

Cluster: We perform experiments on a MapReduce cluster consisting of a master node and 60 slave nodes. Each node has a quad-core Intel i7-3770 3.40GHz processor, 16GB of memory, and 1TB of hard disk. All the nodes are connected via Gigabit Ethernet, and each node has Hadoop-2.2.0 installed. To run our experiments, we used the following Hadoop configuration: 1) the block size of the distributed file system is 128MB, 2) the replication factor is 3, and 3) each node is configured to run 4 map and 4 reduce tasks. By default, we use 60 slave nodes, and the number of reduce tasks in each MapReduce job is set as $60 \times 4 \times 0.95$=228[1].

Data: We conduct our experiments on both synthetic and real datasets. To generate the synthetic data, we use the well-known GTSD data simulator [23]. All the objects are initially distributed within a $10^4 \times 10^4$ 2D domain, and their positions in x and y dimensions follow a Gaussian distribution with parameters $\mathcal{N}(5000, 4000^2)$. The average speed is 30 units per minute and the average time between two consecutive points is 1 minute. Two synthetic datasets i.e., **DS1** and **DS2**, are generated, each of which has two sets i.e., M and R, of trajectory objects. In DS1, each set has 10^4 objects and their positions are monitored for 25 hours in total. In DS2, each set has 10^6 objects, each of which is monitored for 10 hours in total. The total numbers of points in DS1 and DS2 are 30 million and 1.2 billion respectively.

For real data, we use Beijing taxi data [1], which contains the trajectories of 10,357 taxis in the Beijing metropolitan area. Their locations were monitored for a week by on-board GPS devices. After throwing away points that are not in Beijing city, it consists of 12.4 million total coordinate points. The average number of points collected from each taxi is 1,260. The trajectory of each taxi is split into a list of sub-trajectories, so that time interval of each pair of consecutive points is less than 10 minutes. We conduct self-join on the entire dataset, i.e., the sets M and R are the same.

B. Results for k-NN Joins

We first evaluate the sampling-based approach discussed in Section III. Then, we evaluate the algorithms' performance by varying different parameters: N, k, t_q=t_e–t_s, i.e., the length of query time interval, on DS1 and Beijing taxi data. We evaluate the effect of the number of nodes on the DS1 and Beijing datasets. We also evaluate the scalability of the algorithms by varying the sizes of M and R on DS2. The default parameter settings are shown in Table III. For the purpose of reducing the number of st-units in the TDB, we simply set the value of α on each dataset as the average time between any two consecutive points. The start time t_s of the query time interval is chosen randomly. By default, we use SL as the single-machine trajectory join algorithm.

TABLE III
Default parameter settings

Dataset	Default parameters
DS1	T=10, N=400, H=240, α=1 minute, k=10, t_q=6 hours
DS2	T=10, N=400, H=240, α=1 minute, k=10, t_q=2 hours
Beijing taxi	T=10, N=400, H=240, α=3 minute, k=10, t_q=1 day

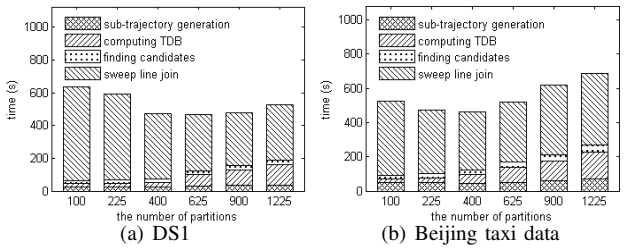
1. Sampling-based approach. We evaluate the quality of k-NN join result returned by the sampling-based approach discussed in Section III. We randomly choose two subsets of trajectories from Beijing taxi data, each of which has 1,000 taxis. We join them using the sampling-based approach. Then for each object, we use the k nearest neighbors returned by SL as the ground truth, and compute the precision and recall of the results returned by the sampling-based approach. Our results show that the average precision and recall values are 0.15 and 0.16 respectively. This shows that the quality of the results returned via sampling is quite poor, mainly because this algorithm overlooks the temporal dimension of the trajectories.

2. Effect of N. Figure 13 shows how the running time of the algorithms is influenced by the value of N. We also illustrate the breakdown of the running time in each of the four stages. When the value of N increases, the data space is split into more, smaller, grids. Since computing the TDB of a partition is based on the information collected from its nearby partitions, more partitions result in a tighter TDB, which increases pruning power and increases the efficiency. But on the other hand, more partitions increase the number of inserted points lying on the borders of grids, and hence it may create more sub-trajectories. Also, the time cost of computing TDB increases as the value of N increases, since we need to deal with more partitions. Let us take N=1,225 in Figure 14(a) as an example. The running time of computing TDB accounts for 22% of the overall running time. The reason is that, when N=1,225, each grid contains less than 9 objects on average, since $|R|$=10^4. All of them are selected as anchor trajectories since k=10. This demonstrates that the time cost of computing TDB is very large if we use all the trajectories from R to compute the TDB, as discussed in Section IV.

Therefore, the overall time cost becomes larger when N is either very small or very large. On balance, in our experiments, the overall time of these algorithms is minimized when N=400 on each dataset. Hence, we set N=400 in subsequent experiments. Among all the 4 stages, the trajectory join stage running time still takes the largest proportion of the time cost, while the sub-trajectory generation and candidate generation stage take a small proportion of the running time.

In addition, we compare the efficiency of BF and SL single-machine algorithms, when used in combination with GL. We denote these 2 algorithms as GL-BF and GL-SL respectively.

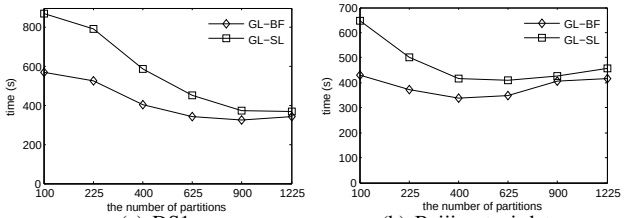
[1] http://wiki.apache.org/hadoop/HowManyMapsAndReduces/



(a) DS1 (b) Beijing taxi data

Fig. 13. Effect of the number of spatial partitions

The running time of the trajectory join stage with different values of N is shown in Figure 14. We can observe that GL-SL is consistently more efficient than GL-BF, which indicates that SL is a better choice for our framework.



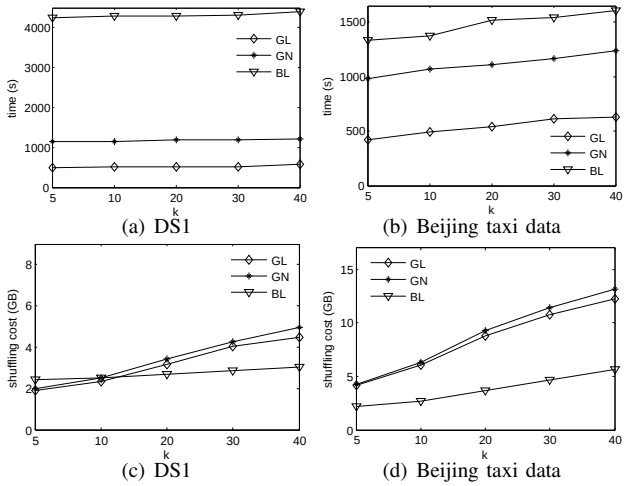
(a) DS1 (b) Beijing taxi data

Fig. 14. Comparison between BF and SL

3. Effect of k. We now investigate how the value of k affects the performance of the k-NN join, in Figure 15. When the value k increases, the running time and shuffling cost of each algorithm increases, because it has to spend more effort on querying more nearest neighbors. BL performs consistently slower than GN and GL, which indicates that our proposed TDB indeed reduces significantly the computational cost. Moreover, GN performs consistently slower than SL. The reason is that – after partitioning the trajectories using grids – some grids contain many trajectories while other grids contain few trajectories. In GN, since we join each partition with its candidates in parallel, the running time of different partitions may vary greatly, hence increasing the overall running time. In GL, the redistribution of objects into groups according to the hash function allows to have a better balance and thus an optimized running time. The results show that GL performs at least twice faster than GN. For example, on the Beijing dataset, we found that uniform partitioning – on GN – leaves some grids having taxis thousands of times larger than others, while GL achieves better load balance using the hash function.

We observed that the shuffling cost of GN and GL is larger than that of BL, and that the largest proportion in this cost is the computation of the TDB. In this stage, for each anchor trajectory, GN and GL need to transmit the maximum distances to all the central points. When data becomes larger, however, the number of such anchor trajectories is a small proportion of the trajectories generated by objects in R, since k is usually much smaller than $|R|$. Hence, the shuffling cost scales quite well, as we will show later in the scalability experiment.

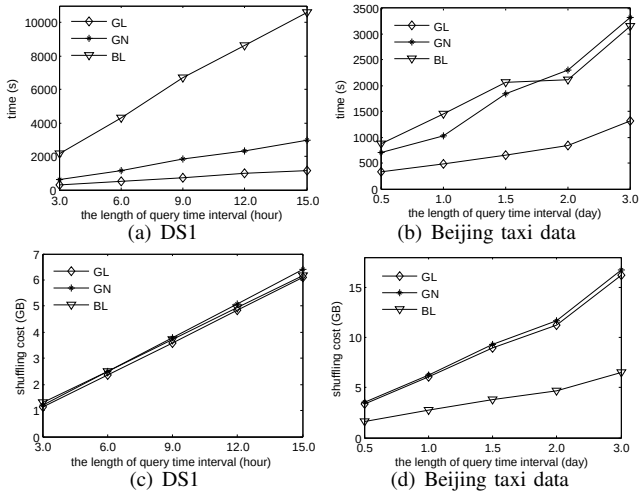
4. Effect of t_q. Figure 16 shows the performance of queries with different t_q. A noticeable effect is the linear dependence of the running time on t_q, as an increase in t_q usually involves more points on the trajectory to process. These results are



(a) DS1 (b) Beijing taxi data (c) DS1 (d) Beijing taxi data

Fig. 15. Effect of k

similar to what we have observed in the case of varying k, i.e., GL performs consistently faster than BL and GN and that the TDB pruning and load balancing are highly effective.

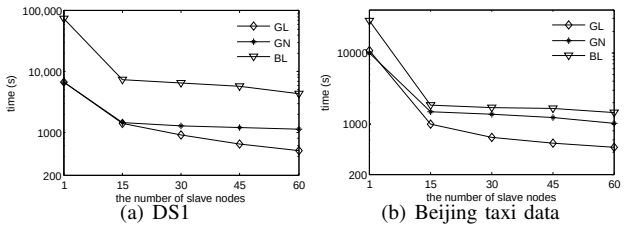


(a) DS1 (b) Beijing taxi data (c) DS1 (d) Beijing taxi data

Fig. 16. Effect of the length of query time interval

5. Effect of number of nodes. Figure 17 shows the performance of queries with various numbers of computing nodes. As can be observed, the trend is similar on DS1 and Beijing dataset. The running time decreases as the number of nodes increases. Comparing the running time on only 1 slave – equivalent to a single-machine execution – is 22 times slower than the running time on 60 machines. This happens because the number of reduce tasks which run in parallel increases with the number of slave nodes, increasing the computing power. Even on a single machine, we can also see that the running time of GN and GL is still smaller than BL. This shows that our proposed bound is very effective also for pruning on a single machine. When the number of machines increases, the gap in running time between the various algorithms decreases. Moreover, the increase in efficiency displays a slow rate of change. We conjecture this is for two reasons: 1) the shuffling cost increases with the number of machines, and 2) the time cost of the I/O operations on the HDFS increases also.

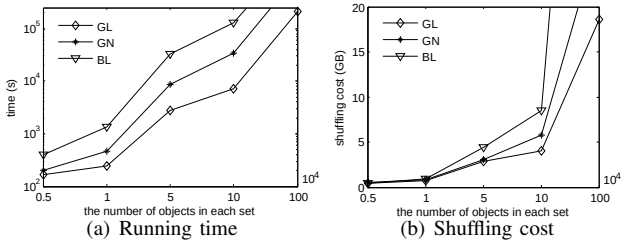
6. Scalability. Figure 18 shows the results of the algo-



(a) DS1 (b) Beijing taxi data

Fig. 17. Effect of the number of slave nodes

rithms' scalability by varying the sizes of M and R (When $|M|$=$|R|$=10^6, we stop GN and BL after running for 3 days). The running time and shuffling cost increases as $|M|$ and $|R|$ increase, due to the fact that more objects are involved in the join operation. The time cost of BL grows quadratically and shuffling cost of BL grows linearly. The running time and shuffling cost of GN and GL grows slower than that of BL. For example, when $|M|$=$|R|$=10^4, the running time of BL is 6 times larger than that of GL, while the shuffling cost remains roughly the same. When $|M|$=$|R|$=10^5, however, the time cost of BL increases to 18 times larger than that of GL, and the shuffling cost also increases to 2 times over GL. This indicates our algorithms have high scalability potential compared to the basic algorithms.

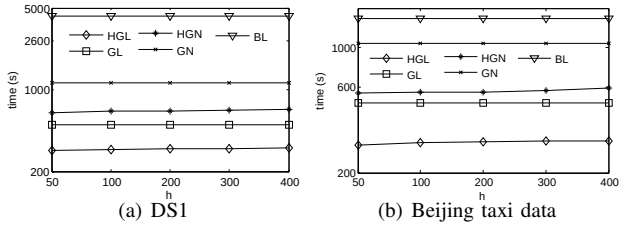


(a) Running time (b) Shuffling cost

Fig. 18. Scalability on DS2

C. Results for (h, k)-NN Joins

We evaluate (h, k)-NN joins by comparing it with the following baseline, which first runs the k-NN joins, then evaluates the function f on objects in M and selects h target objects with their k nearest neighbors finally. We extend GN and GL to HGN and HGL respectively, by using the new TDB proposed in Section VII.



(a) DS1 (b) Beijing taxi data

Fig. 19. Effect of h on (h, k)-NN join

Figure 19 shows the performance gains of our modifications tailored to (h, k)-NN join with various values of h. The default parameter settings on each dataset are the same as in the case of k-NN join. The running time of HGL and HGN increases as the value of h increases. Let us take HGL's performance on Beijing taxi data as an example. The running time increases by 6% when the value of h increases from 50 to 400. As expected, the running time of BL is the largest, since it does not use any pruning technique. HGL and HGN consistently perform at least twice faster than GL and GN respectively. This shows that the new TDB proposed in Section VII is indeed tighter than the TDB designed only for the k-NN join, thus reducing the computational cost.

IX. Conclusions

In this paper, we address the k-NN join on large-scale trajectory data. We present an efficient framework for answering k-NN join queries using MapReduce. It partitions the trajectories using spatial grids, and then computes a time-dependent upper bound (TDB), whose values vary with time, increasing significantly the efficiency of the join operations. We extend k-NN join to (h, k)-NN join, and propose a new tight TDB and an approach under the same solution framework of k-NN join. In the future, we will evaluate our algorithms on more real large-scale data, and study how to answer other complex join queries efficiently using MapReduce.

References

[1] Y. Jing et al, "T-drive: Driving directions based on taxi trajectories," *Association for Computing Machinery*, nov. 2010.
[2] W. Luo, H. Tan, L. Chen, and L. M. Ni, "Finding time period-based most frequent path in big trajectory data," in *SIGMOD*, 2013, pp. 713–724.
[3] Y. Zheng and X. Zhou, *Computing with Spatial Trajectories*. Springer, November 2011.
[4] S. Arumugam and C. Jermaine, "Closest-point-of-approach join for moving object histories." in *ICDE*, 2006, p. 86.
[5] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis, "Nearest neighbor search on moving object trajectories," in *SSTD*, 2005.
[6] Y. Chen and J. M. Patel, "Design and evaluation of trajectory join algorithms." in *GIS*, 2009.
[7] N. A. Bahcall et al, "The spatial correlation function of rich clusters of galaxies," *The Astrophysical Journal*, vol. 270, pp. 20–38, 1983.
[8] R. Benetis et al, "Nearest and reverse nearest neighbor queries for moving objects," *The VLDB Journal*, vol. 15, no. 3, 2006.
[9] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, 2004, pp. 10–10.
[10] A. Akdogan et al, "Voronoi-based geospatial query processing with mapreduce." in *CloudCom*, 2010, pp. 9–16.
[11] Y. Kim and K. Shim, "Parallel top-k similarity join algorithms using mapreduce." in *ICDE*, 2012, pp. 510–521.
[12] Y. Tao, D. Papadias, and Q. Shen, "Continuous nearest neighbor search," in *VLDB*. VLDB Endowment, 2002, pp. 287–298.
[13] M. F. Mokbel et al, "Sina: Scalable incremental processing of continuous queries in spatio-temporal databases," in *SIGMOD*, 2004.
[14] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter, "Scalable sweeping-based spatial join," in *VLDB*, 1998, pp. 570–581.
[15] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: Sql and rich analytics at scale," in *SIGMOD*, 2013, pp. 13–24.
[16] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using mapreduce," in *SIGMOD*, 2010, pp. 495–506.
[17] X. Zhang et al, "Efficient multi-way theta-join processing using mapreduce," *VLDB Endowment*, vol. 5, no. 11, pp. 1184–1195, 2012.
[18] C. Zhang, F. Li, and J. Jestes, "Efficient parallel knn joins for large data in mapreduce," in *EDBT*, 2012, pp. 38–49.
[19] W. Lu et al, "Efficient processing of k nearest neighbor joins using mapreduce," *Proc. VLDB Endow.*, vol. 5, no. 10, pp. 1016–1027, 2012.
[20] W. Zhang, R. Cheng, and B. Kao, "Evaluating multi-way joins over discounted hitting time." in *ICDE*, 2014, pp. 724–735.
[21] M.-W. Lee and S.-w. Hwang, "Robust distributed indexing for locality-skewed workloads," in *CIKM*, 2012, pp. 1342–1351.
[22] R. Bayer, "Symmetric binary b-trees: Data structure and maintenance algorithms," *Acta informatica*, vol. 1, no. 4, pp. 290–306, 1972.
[23] Y. Theodoridis et al, "On the generation of spatiotemporal datasets," *Advances in Spatial Databases*, pp. 147–164, Sep. 1999.

APPENDIX

A. Implementation of Preprocessing Using MapReduce

The detailed implementation of preprocessing using MapReduce is as follows.

Algorithm 3 Preprocessing

```
1: procedure MAP(k_1, v_1)
2:     trList ← TEMPORALPARTITION(v_1);
3:     for each tr ∈ trList do
4:         k_2 ← tr.InterIndex, v_2 ← tr;
5:         OUTPUT(k_2, v_2);
6: procedure REDUCE(k_2, v_2)
7:     for each tr ∈ v_2 do
8:         subTrList ← SPATIALPARTITION(tr);
9:         for each subTr ∈ subTrList do
10:            k_3 ← k_2, v_3 ← (subTr.spatioKey, subTr);
11:            OUTPUT(k_3, v_3);
```

Map. The input of `map()` is a trajectory. Then it is split into a list of sub-trajectories according to the predefined time intervals (line 2). The output is list of $(key, vlaue)$ pairs (line 3-5), where the key is the index of the time interval and the $value$ is a sub-trajectory.

Reduce. After shuffling, trajectories with a same key are sent to a same `reduce()`. We do spatial partition for each trajectory and obtain a ist of sub-trajectories (line 7-8). To facilitate the following queries, for each sub-trajectory $subTr$, we assign it a key (line 9) "$M\_p_i.x\_p_i.y$" if the corresponding object is from M; Otherwise we assign it a key "$R\_p_i.x\_p_i.y$", where p_i is the central point of the grid containing $subTr$.

B. Anchor Trajectory Selection

The algorithm to find the anchor trajectories is shown in Algorithm 4. It initializes a priority queue Q (line 2), in which trajectories are sorted based on the end time in ascending order. k anchor trajectories are selected by calling FINDNEXT($startT$) (line 3-5). Then we dequeue an anchor trajectory from Q (line 7) and update $startT$ (line 8). A new anchor trajectory is selected and enqueued into Q (line 9-10). This process (line 7-10) is iterated until all the anchor trajectories are dequeued. In FINDNEXT($startT$), for each tr, we first compute the gap between its start time and $startT$ (line 14-15). We select the trajectory whose start time is closest to $startT$ (line 16-17). If there exists more than one trajectory having this minimal gap, we select the one whose maximum distance to the central point is the minimum (line 18-20). Finally, the anchor trajectory is selected (line 21-23).

Algorithm 4 Selecting anchor trajectories

```
1: procedure SELECTANCHOR(Tr_j^R, startT, k)
2:     init Q, achList;
3:     while Q.size < k do
4:         tr ← FINDNEXT(Tr_j^R, startT);
5:         Q.add(tr);
6:     while Q.size > 0 do
7:         achTr ← Q.pop; achList.add(achTr);
8:         startT ← achTr.e;
9:         tr ← FINDNEXT(Tr_j^R, startT);
10:        if tr!=null then Q.add(tr);
       return achList;
11: procedure FINDNEXT(Tr_j^R, startT)
12:    minT ← ∞, achTr ← null;
13:    for tr ∈ Tr_j^R do
14:        t' ← tr.s − startT;
15:        if t' < 0 then t' ← 0;
16:        if t' < minT then
17:            achTr ← tr, minT ← t';
18:        else if t' = minT then
19:            d ← MaxDist(achTr, p_j), d' ← MaxDist(tr, p_j);
20:            if d < d' then achTr ← tr;
21:    if achTr!=null then
22:        Tr_j^R.remove(achTr);
23:        achTr ← achTr.subTraj(startT + minT, achTr.e);
       return achTr;
```

C. Computing the Value of H

Suppose the block size of HDFS is S, the storage cost for a point in a trajectory is s, the maximum number of map tasks that can be run in parallel in a cluster as Mp, which can be set as the number of slave nodes times the maximum number of map tasks run in a single node. Suppose the average number of points in a unit time interval is n.

For a specific k-NN join query with time interval length t_q, the expected total storage space for points generated by objects from M, which should be involved in the query processing, is $|M| \times t_q \times n \times s$. So the minimum number of blocks to store these data in HDFS is $\frac{|M| \times t_q \times n \times s}{S}$.

In a MapReduce job, the number of map tasks equals to the number of blocks of the input data. Since only Mp map tasks can be run in parallel, to achieve good load balance for handling set M, we can set the number of groups of M for hashing, i.e., H^M, as follows:

$$H = \left\lceil \frac{|M| \times t_q \times n \times s}{S \times Mp} \right\rceil \times Mp. \tag{13}$$

Similarly, we can compute H^R for points generated by objects from R. For simplicity, in this paper, we set H=$\max\{H^M, H^R\}$.

D. Computing the TDB of p_i

We propose a binary search tree (BST) algorithm to compute the TDB of a central point. We first introduce a new data structure, namely, spatiotemporal-event (abbreviated as $st\text{-}event$). A **st-event** is a triple $(time, dist, operator)$, where $time$ is a time instance, $dist$ is a distance value and $operator$ is an operation, e.g., *add* or *remove*. For each st-unit u, we can create two st-events: e_1=($u.startT$, $u.dist$, *add*) and e_2=($u.endT$, $u.dist$, *remove*). The balanced binary tree [22] we used is TreeMap, which an implementation of the balanced binary tree. In TreeMap, the key is a $dist$ value and the value is a counter, which counts the times of keys. We dynamically maintain a TreeMap of st-events according to their $operators$, and find the st-units that we need, as shown in Algorithm 5.

We first create a list of st-events using the st-units, then sort the list by $time$ in ascending order (line 13-19). We sweep from the earliest time instance (line 4-5). If the next st-event has a strictly larger $time$ value, we query the k-th $dist$ from $treeMap$, form a st-unit and add it to the bound list (lines

6-9). Then, we continue updating the $treeMap$ using the st-events according to their $operators$ (lines 10-11). Finally, we obtain a list of sorted st-units, i.e., $v_i(t)$.

Algorithm 5 Computing the TDB of the central point

```
1: procedure CompTDB(stunitList, k)
2:     treeMap ← initTreeMAP<DIST, COUNTER>;
3:     eventList ← createStEvent(stunitList);
4:     startT ← eventList[0].time; TDB ← initList;
5:     for each event ∈ eventList do
6:         if event.time > startT then
7:             kthDist ← treeMap.findKth(k);
8:             TDB.addStUnit(startT, event.time, kthDist);
9:             startT ← event.time;
10:        if event.operator="add" then treeMap[event.dist] += 1;
11:        else treeMap[event.dist] -= 1;
12:    return TDB;
13: procedure createStEvent(stunitList)
14:    eventList ← null;
15:    for each unit ∈ stunitList do
16:        eventList.addEvent(unit.startT, unit.dist, "add");
17:        eventList.addEvent(unit.endT, unit.dist, "remove");
18:    sortByTime(eventList);
19:    return eventList;
```

E. Proof of Lemma 1

Proof: Figure 9 illustrates the geometric intuition of the lemma. Consider an arbitrary time instance $t \in [st(Tr_i^M), et(Tr_i^M))$. Suppose the k nearest neighbors of p_i are $r_j (1 \leqslant j \leqslant k) \in R$. Then $|p_i, r_j| \leqslant v_i(t)$.

Now let us consider an arbitrary object m at t, whose sub-trajectory $subTr$ is in Tr_i^M. By using triangle inequality, the distance from m to r_j is

$$|m, r_j| \leqslant |m, p_i| + |p_i, r_j| \tag{14}$$

Since $|m, p_i| \leqslant MaxDist(p_i, subTr) \leqslant maxU(Tr_i^M)$, we have

$$|m, r_j| \leqslant maxU(Tr_i^M) + v_i(t) \tag{15}$$

The above equation implies that the distances from m to these k objects are bounded by $maxU(Tr_i^M) + v_i(t)$. Hence, for all objects having sub-trajectories in Tr_i^M at t, the upper bound distance to its k nearest neighbors from R is $u_i(t)$=$maxU(Tr_i^M) + v_i(t)$. ■

F. Proof of Lemma 2

Proof: Consider two arbitrary trajectories, one from Tr_i^M and one from Tr_j^R. Then consider two points p_i^M and p_j^R on the two trajectories, occurring at the same time instance. Using triangle inequality, we have:

$$\begin{aligned} |p_i^M, p_j^R| &\geqslant |p_i, p_j^R| - |p_i, p_i^M| \\ &\geqslant |p_i, p_j| - |p_j, p_j^R| - |p_i, p_i^M| \\ &\geqslant |p_i, p_j| - maxU(Tr_j^R) - maxU(Tr_i^M) \end{aligned} \tag{16}$$

$$\begin{aligned} |p_i^M, p_j^R| &\leqslant |p_i, p_j^R| + |p_i, p_i^M| \\ &\leqslant |p_i, p_j| + |p_j, p_j^R| + |p_i, p_i^M| \\ &\leqslant |p_i, p_j| + maxU(Tr_j^R) + maxU(Tr_i^M) \end{aligned} \tag{17}$$

If their minimum distance is larger than $max(u_i(t))$, then none of the trajectories in Tr_j^R are candidates. If their maximum distance is larger than $min(u_i(t))$, then all of the trajectories in Tr_j^R are candidates. For other cases, a part of them are the candidates. Hence, Lemma 2 holds. ■

G. Proof of Lemma 3

Proof: Consider an arbitrary object m whose sub-trajectory is in Tr_i^M, and the time instance $t_{min} \in [tr.s, tr.e)$ when the minimum distance between m and r occurs. Denote m and r's positions at t_{min} as p_i^m and p_j^r respectively. Using the triangle inequality, we have:

$$\begin{aligned} |p_i^m, p_j^r| &\geqslant |p_i, p_j^r| - |p_i, p_i^m| \\ &\geqslant MinDist(p_i, tr) - maxU(Tr_i^M) \end{aligned} \tag{18}$$

For other time instance $t \in [tr_s, tr_e]$, their distance must be larger than $|p_i^m, p_j^r|$. Also, we know that $|p_i^m, p_j^r| \geqslant 0$. Hence, the lower bound distance from r to objects whose trajectories are in Tr_i^M is $\max\{0, MinDist(p_i, tr) - maxU(Tr_i^M)\}$. ■

H. Implementation of Finding Candidates Using MapReduce

Algorithm 6 gives the detailed implementation of this MapReduce job in GN.

Map. The `map()` takes a partition of trajectories Tr_j^R as input. It enumerates each partition Tr_i^M, and checks which case they belong to (line 3-4). If they belong to case 2, then all the trajectories of Tr_j^R are candidates of Tr_i^M (line 6-8). Otherwise, for case 3, we split each trajectory of Tr_j^R into a list sub-trajectories according to the breakpoints of $u_i(t)$, and then check them one by one using Lemma 4 (line 9-14).

Reduce. In the `reduce()`, we simply output the candidates of Tr_i^M, i.e., C_i^R.

Algorithm 6 Stage 3: Finding candidates

```
1: procedure map(k1, v1)
2:     Tr_j^R ← v1;
3:     for i ←1 to N do
4:         case ← check(p_i, u_i(t), maxU(Tr_i^M), maxU(Tr_j^R));
5:         k2 ← p_i.x_p_i.y;
6:         if case=2 then //Partition check
7:             for tr ∈ Tr do
8:                 v2 ← tr; output(k2, v2);
9:         else if case=3 then //Trajectory check
10:            for tr ∈ Tr do
11:                subTrList ← split(tr, u_i(t));
12:                for subTr ∈ subTrList do
13:                    if subTr is a candidate then //Use Lemma 4
14:                        v2 ← tr; output(k2, v2);
15: procedure reduce(k2, v2)
16:    output(k2, v2);
```

I. Implementation of Trajectory Join Using MapReduce

Algorithm 7 gives the detailed implementation of this MapReduce job in GN.

Map. The input of `map()` is a partition Tr_i^M (line 2). It joins Tr_i^M with the set C_i^R of corresponding generated candidates, using a single-machine algorithm (line 3-4). In this paper, we use BF or SL as discussed before, but any other

single-machine trajectory join algorithms can be incorporated. Finally, we output a list of pairs (line 5-6), where the key is the object id and the value is a list of its k nearest neighbors with their minimum distances.

Reduce. The input of `reduce()` is an object with its k nearest neighbors computed from different partitions. We output k objects whose minimum distances are the smallest (line 8-9).

Algorithm 7 Stage 4: Trajectory join

```
1: procedure MAP(k_1, v_1)
2:     Tr_i^M ← v_1;
3:     read C_i^R from Distributed Cache (or HDFS);
4:     kNNList ← JOIN(Tr_i^M, C_i^R, k); // Use BF or SL
5:     for each tr ∈ Tr_i^M do
6:         OUTPUT(tr.id, kNN); //k nearest neighbors
7: procedure REDUCE(k_2, v_2)
8:     kNN ← reselect k nearest neighbors from v_2;
9:     OUTPUT(tr.id, kNN); //final k nearest neighbors
```