Scalable Algorithms for Nearest-Neighbor Joins on Big Trajectory Data (Extended abstract)

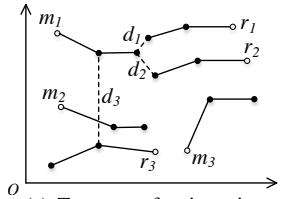
Yixiang Fang[1], Reynold Cheng[1], Wenbin Tang[1], Silviu Maniu[2], Xuan Yang[1]
[1]Department of Computer Science, The University of Hong Kong, Hong Kong
[2]Université Paris-Sud, Université Paris-Saclay, Orsay, France
{yxfang, ckcheng, wbtang, xyang2}@cs.hku.hk, maniu@lri.fr

***Abstract*—Trajectory data are prevalent in systems that monitor the locations of moving objects. In a location-based service, for instance, the positions of vehicles are continuously monitored through GPS; the trajectory of each vehicle describes its movement history. We study joins on two sets of trajectories, generated by two sets M and R of moving objects. For each entity in M, a join returns its k nearest neighbors from R. We examine how this query can be evaluated in cloud environments. This problem is not trivial, due to the complexity of the trajectory, and 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 based on MapReduce. We also develop a novel bounding technique, which enables trajectories to be pruned in parallel. Our approach can be used to parallelize existing single-machine trajectory join algorithms. To evaluate the efficiency and the scalability of our solutions, we have performed extensive experiments on a real dataset.**

I. Introduction

In emerging systems that manage moving objects, a tremendous amount of *trajectory data* [1] 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. 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 [1]. Figure 1(a) gives 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. One of the fundamental queries is the *join*. 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. 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 white dot 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, 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,



(a) 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, $[t_2, t_3]$).

by adopting the *closest-point-of-approach* [2]. In this example, within the time interval $[t_2, t_3]$, the 2-NNs of m_1 are $\{r_1, r_2\}$.

The trajectory join query can be used in a wide range of applications, including business analysis, military applications, celestial body relationship analysis, battlefield analysis, computer gaming, and astronomy areas. For example, the Hubble Space Telescope of NASA generates 140GB of data about movements of stars and asteroids on a weekly basis [2], from which precise trajectory information can be extracted. In the Sloan Digital Sky Survey (SDSS) project, up to 20TB of location data about millions of outer-space objects are collected every night [2]. These huge amounts of data enable the analysis of the behavior of outer-space objects, such as discovery of meteors that were close to the Earth [2]. These tasks, which require the analysis of proximity among trajectory objects, can be facilitated by the trajectory join. For example, given two groups A and B of asteroids the query returns the asteroids from B that have been close to those in A.

Despite the usefulness of trajectory joins, joining two large-scale sets of trajectory objects is not trivial. Here, we briefly present the efficient trajectory join algorithms using MapReduce. We develop a parallel 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 *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 trajectory join algorithms in its framework. We further improve our solutions by using hash functions to distribute trajectory objects to computing units more uniformly, in order to achieve better load balancing. We finally evaluate our solutions on a real dataset using a distributed cluster.

II. Problem definition

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. The timestamps of the first and last points of tr are denoted by $tr.s$ and $tr.e$.

Definition 2: The minimum distance between two (objects with) trajectories tr_i and tr_j is defined as:

$$MinDist(tr_i, tr_j) = \min\{|tr_i.q(t), tr_j.q(t)| \mid t \in \Delta t\}, \quad (1)$$

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

In Figure 1(a), let us consider the minimum distance between two objects m_1 and m_2 with time in $[t_1, t_3]$. We first obtain the "sub-intervals" within $[t_1, t_3]$ such that the line segments from the overlap of their trajectories. These sub-intervals are $[t_1, t_2]$ and $[t_2, t_3]$. For each sub-interval, we compute the minimum distance between the two respective line segments. This can be done by using the method detailed in [2]. Then, their minimum distance is the lowest value of the two minimum distances obtained at $[t_1, t_2]$ and $[t_2, t_3]$.

k-NN join: Given two sets, M and R, of trajectories in the time domain $[T_s, T_e]$, an integer k and a query time interval $[t_s, t_e] \subseteq [T_s, T_e]$, return the k nearest neighbors from R for each object in M during $[t_s, t_e]$.

In Figure 1(a), M=$\{m_1, m_2, m_3\}$, R=$\{r_1, r_2, r_3\}$, and $[T_s, T_e]$=$[t_1, t_3]$. Let k=2 and $[t_s, t_e]$=$[t_2, t_3]$. The results of k-NN join are reported in Figure 1(b).

III. Algorithms

We propose a new parallel solution framework, which consists of two phases, namely **preprocessing phase** and **query phase**. Figure 2 illustrates its workflow. The preprocessing phase needs to be conducted only once, while the query phase is invoked once a k-NN join query arrives. Please refer to [2] for the detailed algorithms and experimental results.

In the *preprocessing phase*, we first partition trajectories along two dimensions: (i) time-wise, using equal-length time intervals, and (ii) spatially, using equal-sized grids.

In the *query phase*, the algorithm proceeds in four stages:

1) **Sub-trajectory extraction.** We extract all the sub-trajectories appearing in $[t_s, t_e]$, and collect relevant statistics from each spatial partition. 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 TDB.** With the collected statistic, we compute a *time-dependent upper bound* (TDB) of Tr_i^M. The TDB is a time dependent spatial circle $c(t)$, such that the k nearest neighbors of S at time t is contained in $c(t)$.
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. Note that C_i^R must contain *all* the k-NNs of objects in M which cross the i-th spatial grid.
4) **Trajectory join.** For each partition Tr_i^M, we join it with C_i^R using a single-machine algorithm [2].

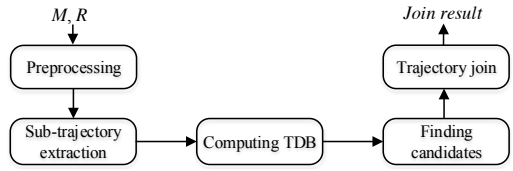


Fig. 2. Algorithm workflow.

We call this approach **GN** (Grid with No load balance). Although GN achieves greater efficiency by using TDB, it may suffer from load balancing issue, due to the skewness of the distribution of objects. We further improve GN by adopting a load balance strategy by assigning grids to machine via some hash functions. We denote this new approach as **GL** (Grid with Load balance). The preprocessing phase and each stage of the query phase are computed using a MapReduce job, in a sequential workflow.

IV. Experiments

Dataset. We use the Beijing taxi dataset, containing the trajectories of 10,357 taxis, collected by GPS devices [1].

Cluster. We use a Hadoop-2.2.0 cluster with a master node and 60 slave nodes. Each node has a quad-core processor and 16GB of memory. All the nodes are connected via Gigabit Ethernet connections.

Queries. By default, we set k=10, $|t_e - t_s|$=1 day. We perform self-joins by varying the values of k and $|t_e - t_s|$. The numbers of temporal and spatial partitions are 10 and 400 respectively.

Results. As a baseline solution, we parallelize the single-machine trajectory join algorithm without using TDB; this baseline is denoted as **BL**. Figure 3 depicts the efficiency under different values of k. When the value k increases, the running time of each algorithm increases. BL performs slower than GN and GL, which indicates that TDB reduces the computational cost significantly. Moreover, GN performs consistently slower than GL, as GL achieves better load balance using a hash function. Figure 4 shows the efficiency under different query interval lengths. We can see that GL performs the best, as it uses both the TDB pruning and load balancing strategy.

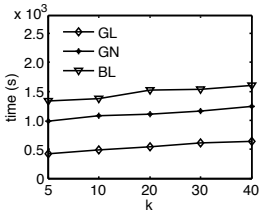


Fig. 3. Varying k.

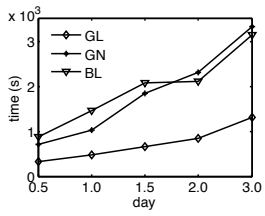


Fig. 4. Varying $|t_e - t_s|$.

Acknowledgment

Yixiang Fang and Reynold Cheng were supported by the Research Grants Council of Hong Kong (RGC Project HKU 17205115) and HKU (Project 201211159083).

References

[1] Y. Zheng et al, *Computing with Spatial Trajectories*. Springer, 2011.

[2] Y. Fang, R. Cheng, W. Tang, S. Maniu, and X. Yang, "Scalable algorithms for nearest-neighbor joins on big trajectory data," in *TKDE*, 2015.