Distributed In-Memory Trajectory Similarity Search and Join on Road Network

Haitao Yuan Guoliang Li
Department of Computer Science and Technology, Tsinghua University, Beijing, China
{yht16@mails., liguoliang@}tsinghua.edu.cn

***Abstract*—Many applications, e.g., Uber, collect large-scale trajectory data from moving vehicles on road network. Trajectory data analytics can benefit many real-world applications, such as route planning and transportation optimizations. Two core operations in trajectory data analytics are trajectory similarity search and join, and both of them rely on a trajectory similarity function to measure the similarity between two trajectories. However, existing similarity functions focus on trajectory points distance and neglect the fact the trajectories should be on road network. Obviously aligning trajectories on road network can remove the noise points introduced by system errors. Toward this goal, we define a road-network-aware trajectory similarity function to measure trajectory similarity. To support trajectory similarity search and join, we propose a filtering-refine framework. In the filtering step, we compute a signature of each trajectory such that if two trajectories are similar, they must share a common signature. We utilize the signatures to prune a huge number of dissimilar pairs. In the refine step, we design effective algorithms to verify the candidates that are not pruned in the filtering step. To support large-scale trajectories, we develop a system DISON for Distributed In-Memory Trajectory Similarity Search and Join on Road Network. DISON splits trajectories into disjoint partitions by considering load balance and locality, and designs effective global index to prune irrelevant partitions. Extensive experiments on real datasets showed that our method achieved high effectiveness, efficiency, and scalability and outperformed existing solutions significantly.**

I. Introduction

With the development of mobile Internet and positioning technology, many systems, e.g, Uber, can easily track the trajectories of moving vehicles. For example, Uber collects the geo-locations of a vehicle in every six seconds, and the sequence of these geo-locations form a trajectory of the vehicle. Trajectory data analytics [26], [27], [34], [17], [39], [18], [19] can benefit many real-world applications, such as route planning, frequent trajectory based navigation systems, and transportation optimizations [13], [21], [23]. Two core operations in trajectory data analytics are trajectory similarity search and join, and both of them reply on a trajectory similarity function to measure the similarity between two trajectories. Obviously, these vehicles are running on road networks. However, existing similarity functions neglect the fact the trajectory points should be constrained on road network.

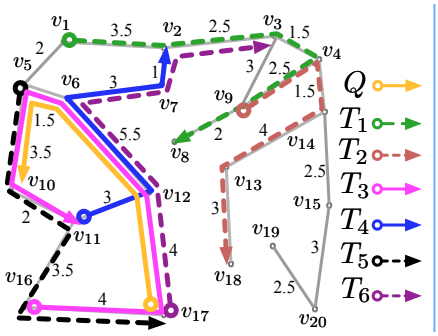
Toward this goal, we define a road-network-aware trajectory similarity function to evaluate trajectory similarity. We first use map matching algorithms [20], [22], [37], [5], [32] to align trajectories on road networks, and each trajectory is transformed into a sequence of road segments. Then, we find Longest Common Road Segment (LCRS) between two trajectories, and define the trajectory similarity function based on LCRS. Our similarity function has two salient features. First, we can remove the noise points introduced by systematic errors by aligning trajectories on road network. Second, most of existing functions only use discrete trajectory points and cannot capture the path between two points, while we use shared road segments to compute more accurate similarity.

To support trajectory similarity search and join, we propose a filtering-refine framework. We first define signatures of trajectories, which are road segments. Then we propose a filtering condition that if two trajectories are similar, they must share a common signature. We utilize this property to prune trajectory pairs that do not share common signatures. In the refine step, we design an effective position-aware algorithm to verify the pairs that are not pruned in the filtering step.

To process large-scale trajectory data, we design a distributed in-memory system on Spark. We first propose an effective trajectory partitioner to split trajectories into different partitions based on the first points and last points of trajectories. The basic idea is that if two trajectories are similar, then the shortest path distance between their first points must be smaller than a bound, and so are the distance between their last points. Then we build a two-layer global index for the partitions, and use the index to prune irrelevant partitions. For each partition, we build a local index and use the filtering-refine framework to find local results. To support trajectory similarity join, we design a global partitioner based on the two collections of trajectories, build a global index and use the index to prune irrelevant partition pairs.

In summary, we make the following contributions.
(1) We propose a road-network-aware trajectory similarity function, which achieves higher quality than existing trajectory similarity functions (see Section II).
(2) We design an efficient filtering-refine framework for trajectory search and join. We propose effective signature-based filtering techniques and refine algorithms (see Section III).
(3) We propose a distributed framework for managing a huge amount of trajectories and implement the framework on Spark. We utilize the road network to divide trajectories into different partitions, build a two-layer global index, and design distributed search and join algorithms (see Section IV).
(4) We conducted a comprehensive evaluation on real world datasets. The results showed that our method outperformed

[1]Guoliang Li is the corresponding author.



Key	Value
$\langle v_1, v_2\rangle$	$\{(T_1, 0)\}$
$\langle v_2, v_3\rangle$	$\{(T_1, 3.5)\}$
$\langle v_9, v_4\rangle$	$\{(T_2, 0)\}$
$\langle v_4, v_{14}\rangle$	$\{(T_2, 2.5)\}$
$\langle v_{16}, v_{17}\rangle$	$\{(T_3, 0)\}$
$\langle v_{17}, v_{12}\rangle$	$\{(T_6, 0), (T_3, 4)\}$
$\langle v_{12}, v_6\rangle$	$\{(T_4, 3), (T_6, 4)\}$
$\langle v_{11}, v_{12}\rangle$	$\{(T_4, 0)\}$
$\langle v_5, v_{10}\rangle$	$\{(T_5, 0)\}$
$\langle v_{10}, v_{11}\rangle$	$\{(T_5, 3.5)\}$

Fig. 1. Example Trajectories on Road Network and Signatures Index($\tau = 0.7$)

existing approaches significantly (see Section V).

II. Preliminaries

A. Road Network and Trajectory

Road Network. A road network is modeled as a weighted, directed graph $G = \langle V, E\rangle$, where V is a vertex set and E is an edge set. A vertex $v_i = \langle x_i, y_i\rangle \in V$ represents the end points of a road segment (e.g., road intersections), where x_i, y_i respectively represent the longitude and latitude of the vertex. An edge $e_k = \langle (v_i, v_j), w_k\rangle \in E$ represents a road segment from vertex v_i to v_j and the corresponding length is w_k. Note that a road segment is usually short, e.g., around 50 meters. For simplicity, we denote e_k as $\langle v_i, v_j\rangle$ if no ambiguity.

Trajectory. Each raw trajectory is a sequence of points obtained from GPS devices and each point is a 2-dimensional tuple with the form of (*longitude,latitude*) typically. Obviously, trajectory points should be on road network, but due to systematic errors, the trajectory points may not be on road network. To address this problem, we align trajectories onto road segments using existing map-matching algorithms [5], [32]. For simplicity, we assume the start point and end points are aligned to vertices on the road networks, and we can easily handle the case that the two points are aligned to road segments. In this way, each trajectory is a path on the road network from the start point to the end point.

Next we formally define a trajectory as below.

Definition 1: **(Trajectory)** Given a road network $G = \langle V, E\rangle$, a trajectory T of a moving object on the road network is a sequence of road segments, $\langle t_1, t_2, \cdots, t_m\rangle$, where $t_k \in E$ is an edge. And the end vertex of t_i is exactly the same as the start vertex of t_{i+1}, where $i \in [1, m-1]$.

Example 1: For simplicity, we assume that each road segment is bidirectional. As shown in the left of Figure 1, there are seven trajectories $\{Q, T_1, T_2, T_3, T_4, T_5, T_6\}$. For example, T_1 contains five edges: $\langle\langle v_1, v_2\rangle, \langle v_2, v_3\rangle, \langle v_3, v_4\rangle, \langle v_4, v_9\rangle, \langle v_9, v_8\rangle\rangle$

B. Trajectory Similarity Function

The trajectory here is essentially a sequence of road segments which is similar to a sequence of letters for a string. Therefore, similar to finding longest common substring between two strings, we find longest common road segments between two trajectories and compute the length of these segments, denoted as $\mathcal{O}(T, Q)$. Then we compute the similarity by normalizing $\mathcal{O}(T, Q)$ to make the similarity between 0 and 1. The normalization method is similar to Jaccard, which is the ratio of intersection size to the union size for two sets. For two trajectories, we take the longest common road segments as the intersection, and the union is the road segments excluding the intersection. The reason of using longest common road segments as intersection is that in real applications, e.g., carpooling, the two trajectories can only share the longest common road segments but not all the common road segments. And the new similarity function LCRS (Longest Common Road Segments) is defined as below.

TABLE I
Overlap Matrix for T_3 and T_5

(a) road segments overlap

	t_1^3	t_2^3	t_3^3	t_4^3	t_5^3	t_6^3
t_1^5	0	0	0	0	3.5	0
t_2^5	0	0	0	0	0	2
t_3^5	0	0	0	0	0	0
t_4^5	4	0	0	0	0	0

(b) $\mathcal{O}(T_3, T_5)$

	t_1^3	t_2^3	t_3^3	t_4^3	t_5^3	t_6^3
t_1^5	0	0	0	0	3.5	3.5
t_2^5	0	0	0	0	3.5	5.5
t_3^5	0	0	0	0	3.5	5.5
t_4^5	4	4	4	4	4	5.5

Definition 2: **(LCRS)** Given two trajectory $T = \langle t_1, \cdots, t_m\rangle$ and $Q = \langle q_1, \cdots, q_n\rangle$, $\mathcal{O}(T, Q)$ is computed as below.

$$\mathcal{O}(T,Q) = \begin{cases} 0 & if\ m = 0\ or\ n = 0 \\ |t_m| + \mathcal{O}(T^{m-1}, Q^{n-1}) & if\ t_m = q_n \\ max\{\ \mathcal{O}(T^{m-1}, Q^n), & \\ \quad \mathcal{O}(T^m, Q^{n-1})\ \} & otherwise \end{cases}$$

where $|t_m|$ is the length of road segment t_m and T^{m-1} is the prefix trajectory of T by removing the last road segment t_m. Then the similarity function LCRS is computed as below by normalizing $\mathcal{O}(T, Q)$ into $[0, 1]$.

$$\texttt{LCRS}(T,Q) = \frac{\mathcal{O}(T,Q)}{|T| + |Q| - \mathcal{O}(T,Q)}$$

where $|T|$ and $|Q|$ respectively represent the length of T and Q on road network.

According to the definition, given two trajectories T and Q, we can utilize *dynamic programming* to compute $\mathcal{O}(T, Q)$ with the time complexity $O(mn)$ and then compute the similarity based on $\mathcal{O}(T, Q)$. In addition, LCRS is symmetrical such that $\texttt{LCRS}(T, Q) = \texttt{LCRS}(Q, T)$.

Example 2: Considering $T = T_3$ and $Q = T_5$ in Figure 1, the directed road segment overlap matrix is shown in Table I(a). According to the definition of LCRS, we construct a matrix (M) to store overlapping road segments length, where M_{ij} represents $\mathcal{O}(T^i, Q^j)$, and we have $\mathcal{O}(T_3, T_5) = |t_5^3| + |t_6^3| = 5.5$ in Table I(b). Finally, we compute $\texttt{LCRS}(T_3, T_5) = \frac{\mathcal{O}(T_3,T_5)}{|T_3|+|T_5|-\mathcal{O}(T_3,T_5)} = \frac{5.5}{20.5+13-5.5} = 0.1964$. T_3 and T_5 share three road segments $\langle v_{16}, v_{17}\rangle$,$\langle v_5, v_{10}\rangle$, and $\langle v_{10}, v_{11}\rangle$, but these road segments cannot be matched at same time because they have different orders on these segments. For example, if $\langle v_{16}, v_{17}\rangle$ is matched, which is the last segment in T_5 as well as the first segment in T_3, then other matched segments between these two trajectories would cause conflicts. Thus we use longest common road segment to address this issue.

C. Trajectory Similarity Search and Join

We next define the trajectory similarity search and join problems. We summarize the notations in Table II.

Definition 3: **(Trajectory Similarity)** Given two trajectories T,Q and threshold τ, if $\texttt{LCRS}(T, Q) \geq \tau$, T and Q are similar.

Definition 4: **(Trajectory Similarity Search)** Given a query trajectory Q, a trajectory set $\mathcal{T} = \{T_1, \cdots, T_{|\mathcal{T}|}\}$ and a threshold τ, the trajectory similarity search problem is to find all trajectories $T \in \mathcal{T}$, such that $\texttt{LCRS}(T, Q) \geq \tau$.

Definition 5: **(Trajectory Similarity Join)** Given two sets of trajectories $\mathcal{T} = \{T_1, \cdots, T_{|\mathcal{T}|}\}$ and $\mathcal{Q} = \{Q_1, \cdots, Q_{|\mathcal{Q}|}\}$, and threshold τ, the trajectory similarity join problem is to find all similar pairs $(T, Q) \in \mathcal{T} \times \mathcal{Q}$, such that $\texttt{LCRS}(T, Q) \geq \tau$.

Remark. Our method can be easily extended to support (1) trajectories with temporal domain, (2) end points on road segments; (3) other functions, as stated in our technical report[2].

D. Related Work

Trajectory Similarity Measures. Most trajectory similarity functions are based on distance aggregation between trajectory points, such as dynamic time warping (DTW) [35], longest common subsequence distance (LCSS) [29], edit distance on real sequence (EDR) [7], edit distance with real penalty (ERP) [6] and DISSIM [10]. The limitation of these similarity functions is that the sample rate of trajectory points has a major influence on the similarity. Moreover, some noise points in a trajectory may cause a big trajectory distance. Obviously, trajectories are essentially continuos, which indicates that using continuous road segments instead of discrete points to represent trajectory is more accurate. Although for existing trajectory similarity functions, we can also first align trajectories to road network and then use the similarity functions on top of the transformed trajectories to compute the similarity, they still use the point distance between trajectories to compute the similarity but not using the shared common roads. Xia et al. [33] propose a spatio-temporal similarity measure for network constrained trajectory, but they just consider the trajectory as a collection of road segments and ignore the order of road segments. Similarly, Shang et al. [25] propose a spatio-temporal function on road network, which respectively aggregates shortest path distance and shortest time difference between vertices of two trajectories as spatial and temporal similarity, but they do not consider the order of vertices. Wang et al. [31] define the Longest Overlapping Road Segments (LORS) to measure similarity between two map-matched trajectories. However, LCRS does not consider the length of the trajectories. We propose the similarity by normalizing the common overlapping road segments into [0,1] and devise effective indexing techniques and algorithms.

Trajectory Similarity Search/Join. There are many studies on trajectory similarity analytics, including trajectory similarity search[3], [4], [8], [31] and similarity join[27], [25]. In addition, to manage and analyze large-scale data more efficiently, there are also some distributed systems for spatial and temporal analytics. Spatial-Hadoop [9] and Hadoop GIS [1] are two distributed spatial data analytics systems over MapReduce. And some studies focus on distributed spatial join [36], [40]. However, these studies do not support trajectory analytics. To address these problems, Xie et al.[34] proposed a distributed in-memory system to answer K-NN queries among trajectories on Spark. However, they partition the trajectory based on segments, which takes more time to merge all candidate trajectories and replicate them to workers for verifying when processing trajectory similarity search. Moreover, they build dual indexing for storing trajectories data, which consumes more memory and takes more time. Another system DITA [26] partitions trajectories based on selected pivot points. However, DITA cannot be used for managing trajectories after map matching because it extracts pivot points from raw trajectory points and builds global and local indexes based on the pivot points. The main difference between our work and DITA is that we support road network and use a new function to evaluate trajectory similarity.

[2]http://dbgroup.cs.tsinghua.edu.cn/ligl/dison.pdf

III. A Filtering-Refine Framework

This section presents a filtering-refine framework to address the trajectory similarity search and join problem. We first propose an effective filtering technique in Section III-A and then present the filtering-refine trajectory search and join algorithms in Section III-B and Section III-C respectively.

A. Prefix and Signature

Given a query trajectory Q, for any data trajectory T that is similar to Q, we have $\frac{\mathcal{O}(T,Q)}{|T|+|Q|-\mathcal{O}(T,Q)} \geq \tau$, and thus $\mathcal{O}(T, Q) \geq \tau|Q|$, i.e., the length of matching segments between T and Q is at least $\tau|Q|$. In other words, the length of non-matching segments of T to Q is at most $(1-\tau)|Q|$. We find a position p such that the length of the first p segments of Q is larger than $(1-\tau)|Q|$ but the length of the first $p-1$ segments is not larger than $(1-\tau)|Q|$. We call these first p segments *the prefix of* Q, denoted by Q^p, and each segment in Q^p is called a *signature*.

Definition 6: **(Prefix and Signature)** Given a trajectory $Q = \{q_1, q_2, \cdots, q_{|Q|}\}$ and a threshold τ, the prefix of Q is the set of the first p segments of Q, such that $\sum_{i=1}^{p} |q_i| > (1-\tau)|Q|$ and $\sum_{i=1}^{p-1} |q_i| \leq (1-\tau)|Q|$. Each segment in the prefix is called a signature.

We can easily prove that if two trajectories have no common signature in their prefixes, then they cannot be similar, as proved in Lemma 1. For example, consider trajectory Q in Figure 1. If the threshold τ is 0.7, $(1-\tau)|Q| = 4.35$ and the prefix of Q is $\{\langle v_{17}, v_{12}\rangle, \langle v_{12}, v_6\rangle\}$. Then we consider trajectory T_5 and compute its prefix, which is $\{\langle v_5, v_{10}\rangle, \langle v_{10}, v_{11}\rangle\}$. As Q and T_5 have no common signature in their prefixes, they cannot be similar[14].

Lemma 1: Given two trajectories T and Q, if they do not share any signature, then they cannot be similar.

Filtering Condition. Given a query trajectory Q, all the data trajectories that do not share signatures with Q can be pruned. In other words, we find the trajectories that share common signatures with Q as candidates.

B. Trajectory Search Algorithm

Indexing. Given a trajectory set $\mathcal{T}$, we build an inverted index for signatures of all trajectories in $\mathcal{T}$. We scan the trajectories

TABLE II
NOTATIONS

Notation	Description
T, Q, T_i, Q_j	trajectory
$\mathcal{T}, \mathcal{Q}$	trajectory set
$\mathcal{T}^{i,j}, \mathcal{T}^i, \mathcal{Q}^j$	trajectory partition
$T[1], Q[1]$	first points of T, Q
$T[-1], Q[-1]$	last points of T, Q
$\texttt{SPD}(u, v)$	shortest-path distance between u, v
N	the number of partitions
$\mathcal{I}^G$	global index
$g_i, g_{i,j}$	road network partition
$\mathcal{I}, \mathcal{I}'$	local inverted index
$L^{\mathcal{T}^{i,j}}, L^{\mathcal{T}^i}, L^{\mathcal{Q}^j}$	longest length of trajectories in partition
$S_f, S_l, S_f^{\mathcal{T}^i}, S_l^{\mathcal{T}^i}, S_f^{\mathcal{Q}^j}, S_l^{\mathcal{Q}^j}$	extended vertex set
$\mathcal{B}_f^{\mathcal{T}^i}, \mathcal{B}_l^{\mathcal{T}^i}, \mathcal{B}_f^{\mathcal{Q}^j}, \mathcal{B}_l^{\mathcal{Q}^j}$	border vertex set

in $\mathcal{T}$ and for each data trajectory $T \in \mathcal{T}$, we generate its signatures. For each signature t in the prefix, we build an inverted list $\mathcal{I}(t)$ and append T on $\mathcal{I}(t)$.

Filtering. Given a query Q, we first compute its prefix and signatures. For each signature g, we get the inverted list of the signature, $\mathcal{I}(g)$, and the trajectories on this list are candidates of query Q.

Verification. For each candidate trajectory T, we verify whether it is actually similar to Q. A naive method computes the real LCRS of Q and T. If $\texttt{LCRS}(Q, T) \geq \tau$, we return T as an answer; we prune it otherwise. As it is expensive to compute LCRS, we propose an effective verification technique.

Position-Aware Verification. Consider a candidate T on the inverted list of a segment g. Let $D(T, g)$ and $D(Q, g)$ respectively denote the length of segments before g in T and Q. If $D(T, g) + D(Q, g)$ is larger than a bound, we do not need to verify the candidate T. Next we discuss how to estimate the bound. First, we assume that g is the first matching segment between T and Q. As $\frac{\mathcal{O}(T,Q)}{|T|+|Q|-\mathcal{O}(T,Q)} \geq \tau$, we have $\mathcal{O}(T, Q) \geq \frac{\tau}{1+\tau}(|T| + |Q|)$. Thus the length of their non-matching segments is at most $|T| + |Q| - 2\mathcal{O}(T, Q) = \frac{1-\tau}{1+\tau}(|T|+|Q|)$. Thus if $D(T, g)+D(Q, g) > \frac{1-\tau}{1+\tau}(|T|+|Q|)$, T and Q cannot be similar, and we do not need to compute $\texttt{LCRS}(Q, T)$; otherwise we compute $\texttt{LCRS}(Q, T)$ to verify candidate T. Then, if g is not the first matching segment between T and Q, T must be also on the inverted list of their first matching segment g', and when we access g', we have already considered T and Q, and thus we can skip T when accessing g. Towards this goal, we can maintain a hashset of the candidate of Q. For each candidate T, if T is in the hashset; we skip it; otherwise we use the position-aware method to verify it and put it into the hashset.

Computing $D(T, g)$ and $D(Q, g)$. We can easily compute $D(Q, g)$ when generating signatures of Q. To avoid computing $D(T, g)$ online, we can materialize it with T on the inverted list of g, i.e., instead of keeping T on the inverted list $\mathcal{I}(g)$, we keep $(T, D(T, g))$ on $\mathcal{I}(g)$.

Overview of the Filtering-Refine Algorithm. Algorithm 1 shows the pseudo code of our algorithm.

Example 3: As shown in the right of Figure 1, we first generate signatures of data trajectories and build an inverted index for these signatures with the similarity threshold $\tau = 0.7$. For instance, the inverted list of signature $\langle v_{12}, v_6\rangle$ is $\{(T_4, 3), (T_6, 4)\}$. Then we generate signatures of query trajectory Q. The prefix length of Q is $(1-\tau)|Q| = 4.35$ and the first two segments of Q is larger than 4.35, so the signatures of Q are the first two segments $\{\langle v_{17}, v_{12}\rangle,\langle v_{12}, v_6\rangle\}$. For signature $\langle v_{17}, v_{12}\rangle$, we get inverted list $\{(T_6, 0), (T_3, 4)\}$ and then refine T_6 and T_3 by position-aware verification. After position-aware verification, T_6 and T_3 are all candidates of the query, so we put them into a hashset of candidates. Then we compute $\texttt{LCRS}(Q, T_6) = 0.45$ and $\texttt{LCRS}(Q, T_3) = 0.71$, so we can prune T_6. Next, for signature $\langle v_{12}, v_6\rangle$, we get inverted list $\{(T_4, 3), (T_6, 4)\}$. Similarly, we execute position-aware verification for T_4. And $D(T_4, \langle v_{12}, v_6\rangle) + D(Q, \langle v_{12}, v_6\rangle) = 3 + 4 = 7$ is larger than $\frac{1-\tau}{1+\tau}(|T_4| + |Q|) = 4.76$, so we can prune T_4. As the candidate hashset contains T_6, we can skip the verification of T_6. Finally, we get a similar trajectory set $\{T_3\}$ for the query trajectory Q.

Algorithm 1: Trajectory Similarity Search

Input: Data Trajectories $\mathcal{T}$, Query Q, Threshold τ
Output: Answers $\mathcal{A}$

```
Inverted Index I = IndexBuilding(T, τ) ;
A ← FilteringAndRefine(Q, τ, I);
```

Function IndexBuilding-offline

Input: Data Trajectories $\mathcal{T}$, Threshold τ
Output: Inverted Index $\mathcal{I}$

```
for T ∈ T do
    for t_i ∈ T do
        if D(T, t_i) ≤ (1 − τ)|T| then
            I(t_i) ← (T, D(T, t_i))
```

Function FilteringAndRefine-online

Input: Query Q, Threshold τ, Inverted Index $\mathcal{I}$
Output: Answers $\mathcal{A}$

```
candidate hashset C ← ∅;
for q_i ∈ Q^p do
    if D(Q, q_i) ≤ (1 − τ)|Q| then
        get inverted list I(q_i) for signature q_i;
        for (T, D(T, q_i)) ∈ I(q_i) do
            if T ∉ C then
                if D(T, q_i)+D(Q, q_i) ≤ (1−τ)/(1+τ)(|T| + |Q|)
                then
                    C ← T;
                    R = T∩Q, T^S = T∩R, Q^S = Q∩R;
                    if LCRS(T^S, Q^S) ≥ τ then A ← T;
```

Early Termination in Verification. According to definition of LCRS, the dynamic programming method of computing $\mathcal{O}(T, Q)$ is inefficient since it takes $O(mn)$ time complexity, where m is the numbers of road segments in T and n is the number of road segments in Q. Instead, we can efficiently compute the overlap segment set $R = T \cap Q$ and $|R|$ is an upper bound of $\mathcal{O}(T, Q)$. If $\frac{|R|}{|T|+|Q|-|R|} < \tau$, we can prune the pair as proved in Lemma 2.

Algorithm 2: Trajectory Similarity Join

Input: Trajectories Sets $\mathcal{T}$ and $\mathcal{Q}$, Threshold τ
Output: Answers $\mathcal{A}$

Inverted Index $\mathcal{I} = IndexBuilding(\mathcal{T}, \tau)$;
Inverted Index $\mathcal{I}' = IndexBuilding(\mathcal{Q}, \tau)$;
$\mathcal{A} \leftarrow MergingAndRefine(\tau, \mathcal{I}, \mathcal{I}')$;

Function `MergingAndRefine-online`

Input: Threshold τ, Inverted Indexes $\mathcal{I}$ and $\mathcal{I}'$
Output: Answers $\mathcal{A}$

candidate pairs hashset $\mathcal{C} \leftarrow \emptyset$;
for $\langle q_i, \mathcal{I}'(q_i)\rangle \in \mathcal{I}'$ **do**
  get inverted list $\mathcal{I}(q_i)$;
  for $\langle (Q, D(Q, q_i)), (T, D(T, q_i))\rangle \in \mathcal{I}'(q_i) \times \mathcal{I}(q_i)$ **do**
    if $(Q, T) \notin \mathcal{C} \wedge (T, Q) \notin \mathcal{C}$ **then**
      if $D(Q, q_i) + D(T, q_i) \leq \frac{1-\tau}{1+\tau}(|T| + |Q|)$ **then**
        $\mathcal{C} \leftarrow (Q, T)$;
        if $\texttt{LCRS}(Q, T) \geq \tau$ **then** $\mathcal{A} \leftarrow (Q, T)$
  if $(T, Q) \in \mathcal{C}$ **then** $\mathcal{A} \leftarrow (Q, T)$

Lemma 2: Given two trajectories Q and T, let $R = T \cap Q$. If $\frac{|R|}{|T|+|Q|-|R|} < \tau$, T and Q cannot be similar.

If $\frac{|R|}{|T|+|Q|-|R|} \geq \tau$, we respectively generate sequence $T^S = T \cap R$ and $Q^S = Q \cap R$ for T and Q, and compute $\mathcal{O}(T^S, Q^S)$ using *Dynamic Programming* and have the fact that $\mathcal{O}(T^S, Q^S)$ is equal to $\mathcal{O}(T, Q)$. Generally, the average number of road segment in T^S and Q^S is far less than the average number of road segments in T and Q, which will greatly reduce the space complexity and time complexity. In addition, when computing the value using the matrix, if we find that a value is larger than $\frac{\tau}{1+\tau}(|T| + |Q|)$, we can stop as T and Q share enough common segments and must be similar. For example, taking the candidate trajectory T_6 of Example 3 into consideration, we have $R = \{\langle v_{17}, v_{12}\rangle, \langle v_{12}, v_6\rangle\}$ and $\frac{|R|}{|T_6|+|Q|-|R|} = 0.45 < 0.7$, so we can prune T_6. Similarly, for candidate T_3, we get the intersection and compute LCRS similarity(or $\mathcal{O}(Q^S, {T_3}^S)$). We first compute the lower bound of $\mathcal{O}(Q^S, {T_3}^S)$, which is $\frac{\tau}{1+\tau}(|Q| + |T_3|) = 14.41$. When computing $\mathcal{O}(Q^S, {T_3}^S)$, if a value is beyond 14.41, we stop.

C. Trajectory Join Algorithm

Algorithm 2 shows the pseudo codes of the trajectory join.
Indexing. Given two trajectory sets $\mathcal{T}$ and $\mathcal{Q}$, we respectively build inverted index for signatures of all trajectories in $\mathcal{T}$ and $\mathcal{Q}$, which is similar to indexing in trajectory search algorithm.
Merging. For each signature q, if it has inverted list $\mathcal{I}(q)$ on $\mathcal{T}$ and inverted list $\mathcal{I}'(q)$ on $\mathcal{Q}$, then the pairs $(T, Q) \in \mathcal{I}(q) \times \mathcal{I}'(q)$ are candidate pairs.
Verification. For each candidate trajectory pair $\langle T, Q\rangle$, we first use the position-aware verification to verify it and then compute $\texttt{LCRS}(T, Q)$, which is similar to verification in trajectory search algorithm.

IV. Distributed Framework

This section presents a distributed framework, which includes four main components. (1) *Partitioning.* We first propose a partitioning strategy to split trajectories into different partitions in Section IV-A. (2) *Global Indexing.* We propose a global index and use it to find relevant partitions in Section IV-B. (3) *Trajectory Search.* We devise effective algorithms and indexes for distributed trajectory search in Section IV-C. (4) *Trajectory Join.* We develop effective algorithms for distributed trajectory join in Section IV-D.

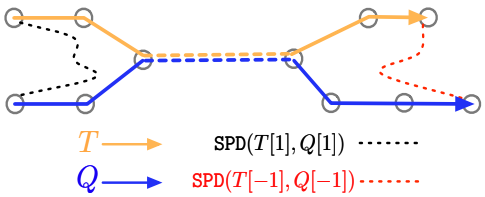


Fig. 2. An Example of Head-Tail Distance

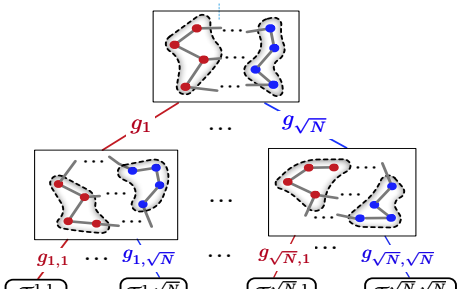


Fig. 3. Two-Layer Tree Structure of Global Index

A. Partitioning

Trajectory partitioning aims to split trajectories into different partitions. To get high-quality partitions, we need to consider several factors. (1) *Locality.* Similar trajectories should be in the same partition (for avoiding data transmission among different partitions, especially for join). (2) *Load Balance.* Each partition has similar size to guarantee load balance. (3) *Complete and Disjoint.* Each trajectory is in one and only one partition, i.e., different partitions have no overlap to save storage space and transmission cost. Towards this goal, we propose a head-tail based trajectory partitioning method.

Head-Tail Based Prunning. We have an observation that given two trajectories T and Q, if they are similar, their two first points will not be too far and their two last points will not be too far either. Formally, if the shortest-path distance between their first points, called *head distance*, is larger than a bound, then they cannot be similar; if the shortest-path distance between their last points, called *tail distance*, is larger than a bound, then they cannot be similar. We can utilize this property to split the trajectories and prune irrelevant partitions.

Let $T[1]$ and $T[-1]$ respectively denote the first point and last point of trajectory T. Let $\texttt{SPD}(u, v)$ denote the shortest-path distance between two vertices u and v. Based on Lemmas 3 and 4, given a query trajectory Q, if a data trajectory T is similar to Q, the length of non-shared segments of T to Q is at most $\frac{1-\tau}{1+\tau}(|T| + |Q|)$ and $\frac{1-\tau}{\tau}|Q|$.

Lemma 3: Given a query trajectory Q and a threshold τ, for trajectory T, if it is similar to Q, the length of their non-shared segments is at most $\frac{1-\tau}{1+\tau}(|T| + |Q|)$.

Lemma 4: Given a query trajectory Q and a threshold τ, for any trajectory, if it is similar to Q, the length of their non-shared segments is at most $\frac{1-\tau}{\tau}|Q|$.

Moreover, if T and Q do not share the same point, their head distance is not larger than $\frac{1-\tau}{1+\tau}(|T|+|Q|)$ and $\frac{1-\tau}{\tau}|Q|$, because the length of their non-shared segments is at least the sum of the distance from the first point of T to the first meeting point and the distance from the first point of Q to the first meeting point, which is not smaller than the head distance, as shown in Figure 2. Similarly, their tail distance is not larger than $\frac{1-\tau}{1+\tau}(|T|+|Q|)$ and $\frac{1-\tau}{\tau}|Q|$. Moreover, the sum of the head distance and tail distance is not larger than $\frac{1-\tau}{1+\tau}(|T|+|Q|)$ and $\frac{1-\tau}{\tau}|Q|$ as proved in Lemmas 5 and 6.

Lemma 5: Given a query trajectory Q and a threshold τ, if trajectory T is similar to Q, we have
$\texttt{SPD}(T[1],Q[1])+\texttt{SPD}(T[-1],Q[-1]) \leq \frac{1-\tau}{1+\tau}(|T|+|Q|)$.

Lemma 6: Given a query trajectory Q and a threshold τ, if trajectory T is similar to Q, we have
$\texttt{SPD}(T[1],Q[1])+\texttt{SPD}(T[-1],Q[-1]) \leq \frac{1-\tau}{\tau}|Q|$.

For example, considering Q and T_4 in Figure 1, their LCRS similarity is 0.169 and the sum of their head distance and tail distance is $\texttt{SPD}(v_{17},v_{11})+\texttt{SPD}(v_{10},v_2)=7+9=16$. Suppose the similarity threshold τ is 0.7. We compute $\frac{1-\tau}{1+\tau}(|Q|+|T_4|)=\frac{0.3}{1.7}(14.5+12.5)=4.7647$ and $\frac{1-\tau}{\tau}|Q|=6.2143$. As the sum of head distance and tail distance is larger than the two bounds, they cannot be similar. Thus we can use the head point and tail point to do pruning.

Partitioning. Suppose we want to split the trajectories into N partitions. We first split the trajectories into $\sqrt{N}$ groups based on their first points, and then for each group, we further partition the trajectories in this group into $\sqrt{N}$ sub-groups based on their last points. Each sub-group corresponds to a partition. As each first point or last point must be a vertex on the road network, we partition the trajectories based on the vertices on the road network.

Step 1 - Trajectory partitioning based on first points. Formally, we first assign each vertex with a weight which is the number of trajectories whose first points are exactly the same as the vertex. Then we utilize existing graph partitioning algorithms, e.g., [15], to split the road network into $\sqrt{N}$ groups $g_1, g_2, \cdots, g_{\sqrt{N}}$, where each group nearly has the same vertex weight. Thus we get $\sqrt{N}$ groups and each group contains a set of vertices. We assign each trajectory into a group: a trajectory is in a group if its first point is in the group.

Step 2 - Trajectory partitioning based on last points. Then for each group g_i, we use the similar idea to split the vertex into $\sqrt{N}$ sub-groups, $g_{i,1}, g_{i,2}, \cdots, g_{i,\sqrt{N}}$, based on the last points of trajectories in this group. Each sub-group $g_{i,j}$ corresponds to a partition $\mathcal{T}^{i,j}$ and we get N partitions. Each sub-group also contains a set of vertices. Next we assign each trajectory into a partition: a trajectory is in partition $\mathcal{T}^{i,j}$ if its first point is in group g_i and its last point is in sub-group $g_{i,j}$.

We can see that if two trajectories have similar first points and last points, they will be in the same partition, and this partitioning strategy satisfies the above three properties. The reasons are three-folds.

(1) If the first or last points between two trajectories have large distance, they are assigned to different partitions. According to Lemma 3-6, the first or last points of similar trajectories must be close to each other, so dissimilar trajectories would be assigned into different partitions and thus our strategy satisfies the locality property.

(2) We guarantee each partition has the same vertex weight, where the weight corresponds to the number of trajectories, so each partition has the same number of trajectories and thus our strategy can guarantee load balance.

(3) The graph partitioning algorithms guarantee each road vertex is in one and only one partition. Therefore, in step 1, each trajectory is assigned to one and only one group. Similarly in step 2, for each group, each trajectory is assigned to one and only one sub-group. And each sub-group corresponds to a partition, so each trajectory is in one and only one partition and thus our strategy satisfies the third property.

B. Local and Global Indexing

Global Indexing. The global index is a two-layer tree structure as shown in Figure 3. In the root node, we keep a hashmap, which maps a vertex into a group ID: $v \rightarrow g_i$. In the first level, each node corresponds to a group. For each group, we keep a hashmap, which maps a vertex into a sub-group ID $v \rightarrow g_{i,j}$. In the second level, each node corresponds to a partition. For each partition $\mathcal{T}^{i,j}$, we also keep the longest length of trajectories in this partition, which is denoted as $L^{\mathcal{T}^{i,j}}$. The space size of the root is $\mathcal{O}(|V|)$ to keep the hashmap, where $|V|$ is the number of vertices in the road network. The space of the first level is $\mathcal{O}(\sqrt{N}|V|)$. The space size of the second level is N. Thus the global index size is $\mathcal{O}(\sqrt{N}|V|+N)$, which is independent on the trajectory size and is very small.

Local Indexing. For each partition, we generate the signatures for trajectories in this partition and build an inverted index as described in Section III.

C. Trajectory Search

Given a query Q, we first use the global index to find the relevant partitions and send the query to the relevant partitions. Then, each partition uses the local indexes to find local results and sends back the results to the master. The master collects all the local results and gets the global results. Algorithm 3 shows the pseudo code.

Global Searching. Given a query Q, we can prune the group g_i if the head distances of all vertices in g_i to the first point of Q ($Q[1]$) are larger than $\frac{(1-\tau)}{\tau}|Q|$. Toward this goal, we extend $Q[1]$ on the road network using the Disjtrka algorithm, and find the set S_f of vertices within $\frac{(1-\tau)}{\tau}|Q|$ distance to $Q[1]$. Meanwhile, we record the shortest-path distance $\texttt{SPD}(T[1],Q[1])$ between $Q[1]$ and each vertex $T[1]$ in S_f. Similarly let S_l denote the set of vertices within $\frac{(1-\tau)}{\tau}|Q|$ distance to the last point of Q ($Q[-1]$) and we also record the shortest-path distance $\texttt{SPD}(T[-1],Q[-1])$ for each vertex in S_l. If a node in the first level overlaps with S_f, the node is relevant. Then for each of its children, if it overlaps with S_l, the leave node is a relevant partition (and all other partitions are pruned). Finally, we further verify each relevant partition

Algorithm 3: Distributed Similarity Search

Input: Query Q, Threshold τ, Data trajectories $\mathcal{T}$
Output: Answers $\mathcal{A}$

Partitioning and building global index $\mathcal{I}^G$ for $\mathcal{T}$;
Building local index for each partition;
Candidate Partitions $\mathcal{C} = GlobalPruning(Q, \tau, \mathcal{I}^G)$;
for $\mathcal{T}^{i,j} \in \mathcal{C}$ **do**
  $\mathcal{I}$ = local inverted index in $\mathcal{T}^{i,j}$;
  $\mathcal{A} \leftarrow FilteringAndRefine(Q, \tau, \mathcal{I})$

Function `GlobalPruning`

Input: Query Q, Threshold τ, Global Index $\mathcal{I}^G$
Output: Candidate Partitions $\mathcal{C}$

S_f = vertices by extending $Q[1]$ within $\frac{1-\tau}{\tau}(|Q|)$;
S_l = vertices by extending $Q[-1]$ within $\frac{1-\tau}{\tau}(|Q|)$;
for $\langle v_f, v_l \rangle \in S_f \times S_l$ **do**
  $\mathcal{T}' \leftarrow$ the partition which overlaps with v_f, v_l by $\mathcal{I}^G$;
for $\mathcal{T}^{i,j} \in \mathcal{T}'$ **do**
  $d_f = min_{v_f \in S_f \cap g_i}\{\texttt{SPD}(Q[1], v_f)\}$;
  $d_l = min_{v_l \in S_l \cap g_{i,j}}\{\texttt{SPD}(Q[-1], v_l)\}$;
  if $d_f + d_l \leq \frac{1-\tau}{\tau}|Q| \wedge d_f + d_l \leq \frac{1-\tau}{1+\tau}(|Q| + L^{\mathcal{T}^{i,j}})$ **then** $\mathcal{C} \leftarrow \mathcal{T}^{i,j}$

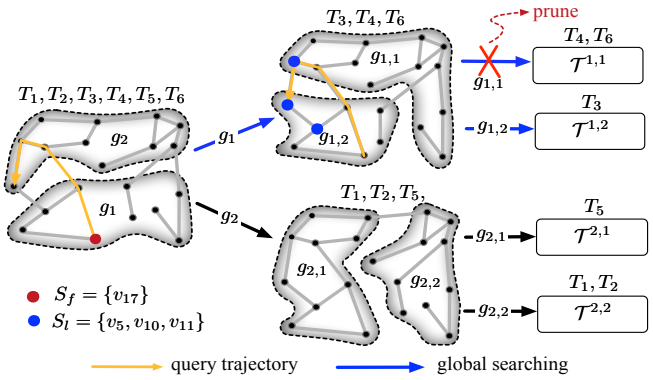


Fig. 4. An Example of Global Searching

based on Lemmas 5 and 6. For each relevant partition $\mathcal{T}^{i,j}$, we compute $d_f = min_{v_f \in S_f \cap g_i}\{\texttt{SPD}(Q[1], v_f)\}$, and $d_l = min_{v_l \in S_l \cap g_{i,j}}\{\texttt{SPD}(Q[-1], v_l)\}$. If $d_f + d_l \leq \frac{1-\tau}{1+\tau}(L^{\mathcal{T}^{i,j}} + |Q|)$ and $d_f + d_l \leq \frac{1-\tau}{\tau}|Q|$, $\mathcal{T}^{i,j}$ may contain results; otherwise $\mathcal{T}^{i,j}$ is pruned.

Example 4: Take trajectories and road network in Figure 1 as an example. As shown in Figure 4, we get four partitions. We respectively set query trajectory and similarity threshold as Q and 0.8, and thus we get $S_f = \{v_{17}\}$ and $S_l = \{v_5, v_{10}, v_{11}\}$. Then we use the hashmap in the first point layer of global index to get the candidate nodes which overlap with S_f, and then use the hashmaps in these candidate nodes to get candidate leaf nodes which overlap with S_l. We get candidate partitions $\mathcal{T}^{1,1}, \mathcal{T}^{1,2}$. Finally, we refine each candidate partition. For example, for partition $\mathcal{T}^{1,1}$, we compute $d_f = \texttt{SPD}(v_{17}, v_{17}) = 0$ and $d_l = \texttt{SPD}(v_{10}, v_5) = 3.5$, thus $d_f + d_l = 3.5$, which is larger than the upper bound $\frac{1-\tau}{1+\tau}(|Q| + L^{1,1}) = 3.389$, so we can prune partition $\mathcal{T}^{1,1}$. And the candidate partition is $\mathcal{T}^{1,2}$.

Local Searching. Given a query Q, we use the local search algorithm to compute local results as described in Section III.

Algorithm 4: Distributed Similarity Join

Input: Sets $\mathcal{Q}$ and $\mathcal{T}$, Threshold τ
Output: Answers $\mathcal{A}$

Build global index $\mathcal{I}^G$ by partitioning $\mathcal{Q}$ and $\mathcal{T}$ together;
$\mathcal{C}^P = GlobalJoin(\mathcal{Q}, \mathcal{T}, \tau, G)$;
Build bi-graph $\mathcal{G}$ for pairs in $\mathcal{C}^P$ using Cost Model;
Apply Division-based Load Balancing in $\mathcal{G}$;
for $(\mathcal{T}^i, \mathcal{Q}^j) \in \mathcal{C}^P$ **do**
  if $\mathcal{T}^i \rightarrow \mathcal{Q}^j \in \mathcal{G}$ **then** send trajectories $T \in \mathcal{T}^i$ and inverted index $\mathcal{I}^{\mathcal{T}^i}$ to $\mathcal{Q}^j$ if T has candidates in $\mathcal{Q}^j$;
  if $\mathcal{Q}^j \rightarrow \mathcal{T}^i \in \mathcal{G}$ **then** send trajectories $Q \in \mathcal{Q}^j$ and inverted index $\mathcal{I}^{\mathcal{Q}^j}$ to $\mathcal{T}^i$ if Q has candidates in $\mathcal{T}^i$;
  $\mathcal{A} \leftarrow MergingAndRefine(\tau, \mathcal{I}^{\mathcal{T}^i}, \mathcal{I}^{\mathcal{Q}^j})$

Function `GlobalJoin`

Input: Sets $\mathcal{Q}$ and $\mathcal{T}$, Threshold τ, Global Index $\mathcal{I}^G$
Output: Candidate Partition Pairs $\mathcal{C}^P$

$\mathcal{C}^P = \emptyset$;
for $\mathcal{Q}^j \in \mathcal{Q}$ **do**
  $\mathcal{B}_f^{\mathcal{Q}^j}, \mathcal{B}_l^{\mathcal{Q}^j}$ = two layer border vertices of $\mathcal{Q}^j$;
  $S_f^{\mathcal{Q}^j}, S_l^{\mathcal{Q}^j}$ = set of vertices within $\frac{1-\tau}{\tau}(|L^{Q_j}|)$ distance to vertices in $\mathcal{B}_f^{\mathcal{Q}^j}, \mathcal{B}_l^{\mathcal{Q}^j}$;
  for $\langle v_f, v_l \rangle \in S_f^{\mathcal{Q}^j} \times S_l^{\mathcal{Q}^j}$ **do**
    $\mathcal{T}' =$ partitions of $\mathcal{T}$ using v_f, v_l search in $\mathcal{I}^G$;
    for $\mathcal{T}^i \in \mathcal{T}'$ **do**
      $\mathcal{B}_f^{\mathcal{T}^i}, \mathcal{B}_l^{\mathcal{T}^i}$ = two layer border vertices of $\mathcal{T}^i$;
      $d_f = min_{v_f \in S_f^{\mathcal{Q}^j} \cap \mathcal{B}_f^{\mathcal{T}^i}, v_f' \in \mathcal{B}_f^{\mathcal{Q}^j}}\{\texttt{SPD}(v_f, v_f')\}$;
      $d_l = min_{v_l \in S_l^{\mathcal{Q}^j} \cap \mathcal{B}_l^{\mathcal{T}^i}, v_l' \in \mathcal{B}_l^{\mathcal{Q}^j}}\{\texttt{SPD}(v_l, v_l')\}$;
      if $d_f + d_l \leq \frac{1-\tau}{\tau}L^{\mathcal{Q}^j} \wedge d_f + d_l \leq \frac{1-\tau}{\tau}L^{\mathcal{T}^i}$ **then** $\mathcal{C}^P \leftarrow (\mathcal{T}^i, \mathcal{Q}^j)$

D. Trajectory Join

Given two trajectory sets $\mathcal{T}$ and $\mathcal{Q}$, we first construct a global partitioner and use the partitioner to split them into different partitions. We still first partition the vertices in the road network, where the weight of each vertex is the number of trajectories in $\mathcal{T} \cup \mathcal{Q}$ whose first points or last points are the vertex. Then we have two trajectories partitions $\mathcal{T}^i$ and $\mathcal{Q}^i$ in the i-th partition, where $\mathcal{T}^i$ denotes the i-th partition of $\mathcal{T}$ and $\mathcal{Q}^i$ denotes the i-th partition of $\mathcal{Q}$. Thus we have a common two-layer structure tree global index for $\mathcal{T}$ and $\mathcal{Q}$, and the leaf node corresponding to the i-th partition stores $L^{\mathcal{T}^i}$ and $L^{\mathcal{Q}^i}$, which respectively are the longest length of trajectories in $\mathcal{T}^i$ and $\mathcal{Q}^i$. We then find all partition paris $(\mathcal{T}^i, \mathcal{Q}^j)$ such that there exist trajectory $T \in \mathcal{T}^i$ and $Q \in \mathcal{Q}^j$ such that $\texttt{LCRS}(T, Q) \geq \tau$ based on global index. Afterward, we compute the join result between $\mathcal{T}^i$ and $\mathcal{Q}^j$ by either sending $\mathcal{T}^i$ to $\mathcal{Q}^j$ or sending $\mathcal{Q}^j$ to $\mathcal{T}^i$. Without loss of generality, suppose $|\mathcal{Q}^j| < |\mathcal{T}^i|$ and $\mathcal{Q}^j$ is sent to $\mathcal{T}^i$ for each pair $(\mathcal{T}^i, \mathcal{Q}^j)$. Finally, we find all trajectories pairs $(Q, T) \in (\mathcal{T}^i, \mathcal{Q}^j)$ such that $\texttt{LCRS}(T, Q) \geq \tau$ by querying local indexes, which is similar to the trajectory join procedure in Section III-C.

Global Join. We build a global index $\mathcal{I}^G$ for partitions of $\mathcal{T}$ and $\mathcal{Q}$. Without loss of generality, we find all candidate

TABLE III
TRAJECTORY DATASETS

Road Networks	# of vertices	# of edges
BRN	1,285,215	2,690,296
NRN(original)	21,259	29,901
NRN(new)	496,173	1,009,582

partitions from $\mathcal{T}$ for each partition $\mathcal{Q}^j \in \mathcal{Q}$ by searching every trajectory $Q \in \mathcal{Q}^j$ on the global index. However it is inefficient to enumerate every trajectory $Q \in \mathcal{Q}^j$ and compute its extended vertex sets to get candidate partitions. To address this issue, we introduce border vertices, which are a small subset of vertices, and we only extend the border vertices.

Definition 7: **(Border Vertex)** For each partition of the road network, if a vertex has an edge to vertex in other partitions, it is called a border vertex of the partition [41].

For example, considering $\mathcal{T}^{1,1}$ in Figure 4. In the first point layer, the corresponding partition of road network is g_1 and border vertices set of g_1 is $\{v_{11}, v_{12}, v_{13}, v_{15}\}$. Similarly, in the last point layer, the partition of road network is $g_{1,1}$ and the corresponding border vertex set is $\{v_5, v_6, v_{14}\}$.

Let $\mathcal{B}_f^{\mathcal{Q}^j}$ and $\mathcal{B}_l^{\mathcal{Q}^j}$ respectively denote border vertex set of $\mathcal{Q}^j$ in the first and last point layer. Let $S_f^{\mathcal{Q}^j}$ and $S_l^{\mathcal{Q}^j}$ respectively denote the set of vertices within $\frac{1-\tau}{\tau} L^{\mathcal{Q}^j}$ distance to vertices in $\mathcal{B}_f^{\mathcal{Q}^j}$ and $\mathcal{B}_l^{\mathcal{Q}^j}$. Similarly, we can get border vertices sets $\mathcal{B}_f^{\mathcal{T}^i}$ and $\mathcal{B}_l^{\mathcal{T}^i}$ for each partition $\mathcal{T}^i$ in $\mathcal{T}$. In global join, for partition $\mathcal{Q}^j$, we first use vertices in $S_f^{\mathcal{Q}^j}$ and $S_l^{\mathcal{Q}^j}$ to get candidate partitions $\mathcal{T}'$. Then for partition $\mathcal{T}^i$ in $\mathcal{T}'$, we refine the partition pair $(\mathcal{T}^i, \mathcal{Q}^j)$ based on Lemma 7. Algorithm 4 shows the pseudo code.

Lemma 7: Given two trajectory partitions $\mathcal{T}^i, \mathcal{Q}^j$ and a threshold τ, if $\mathcal{T}^i$ and $\mathcal{Q}^j$ have similar trajectories, we have $min\{\texttt{SPD}(v_f, v'_f)\} + min\{\texttt{SPD}(v_l, v'_l)\} \leq \frac{1-\tau}{\tau} L^{\mathcal{Q}^j}$,
where $v_f \in \mathcal{B}_f^{\mathcal{T}^i}, v'_f \in \mathcal{B}_f^{\mathcal{Q}^j}, v_l \in \mathcal{B}_l^{\mathcal{T}^i}, v'_l \in \mathcal{B}_l^{\mathcal{Q}^j}$.

Shuffle Decision. For each partition pair candidate $(\mathcal{T}^i, \mathcal{Q}^j)$, we send them to some workers for local joins. We implement the *cost model* in DITA [26] to determine the data shuffle direction between $\mathcal{T}^i$ and $\mathcal{Q}^j$. The *cost model* constructs a directed bi-graph between partitions of $\mathcal{T}$ and $\mathcal{Q}$ and proposes a greedy algorithm to determine directions of edges in the graph to achieve the best performance by reducing total costs. Moreover, it is a bottleneck when the local join between $\mathcal{T}$ and $\mathcal{Q}$ inherently causes huge total costs no matter how the data shuffle between these two partitions. To maintain load balancing, we implement *Division-based Load Balancing* from DITA [26], which take full advantage of parallel computing by dividing the workloads of partitions among workers.

Local Join. Given candidate partition pair $(\mathcal{T}^i, \mathcal{Q}^j)$, we use the local join algorithm to compute local join results as described in Section III.

V. EXPERIMENTS

We conducted experiments to evaluate our techniques. Our experimental goals includes: 1) evaluate the effectiveness our similarity function, 2) evaluate the search efficiency of our filtering-refine algorithm in a centralized setting, and 3) evaluate the efficiency and scalability of our distributed algorithms.

TABLE IV
PARAMETERS (DEFAULT VALUE IS HIGHLIGHTED)

Parameter	Value
threshold τ	0.5, 0.6, 0.7, **0.8(search)**, **0.9(join)**
N	625, **1296(join)**, 2041, **4096(search)**, 6561
# of cores	54, 108, 162, **216**
sample rate	**0.25(join)**, 0.5, 0.75, **1.0(search)**

A. Experimental Setup

Datasets. We used two road networks: Beijing Road Network[3] and Nanjing Road Network[4]. Beijing Road Network contained 1,285,215 vertices and 2,690,296 edges and the length of edge was from tens of meters to several hundred meters. Nanjing Road Network contained 496,173 vertices and 1,009,582 edges. We used taxi trajectories in Beijing and Nanjing[5]. We first aligned the trajectories on road network with the map matching algorithm [20]. Table **??** showed trajectories, where *Avg# of road segments* was the average road segments number of trajectories after map-matching on road network.

Baseline methods. We compared our centralized framework with five baselines: VP-Tree (vantage point tree) [11], MBE (the Minimal Bounding Envelop) [30], Torch [31], TP (Two-Phase algorithm) [25] and the centralized implementation of DITA [26]. We also compared DISON with three distributed baselines DFT [34], DITA and distributed implementation of Torch. Note that some baselines(e.g. MBE, VP-Tree, TP, DFT and DITA) used distance functions (e.g. Fréchet, DTW), which was different from our similarity function. Therefore, we converted similarity threshold τ to distance threshold η by the formulation $\eta = \lambda \ln \frac{1}{\tau}$, where η represents the distance threshold and λ was a constant value to guarantee that they found the same number of results to make a fair comparison. Besides, we adapted Torch to support similarity search on LCRS. When comparing with Torch in the distributed setting, we utilized our partition strategy and global pruning algorithm.

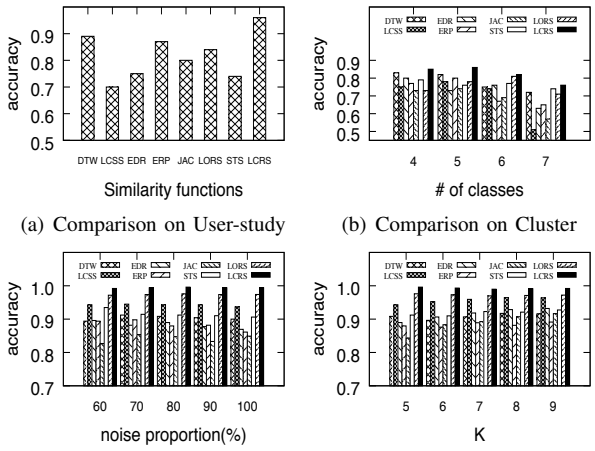
Parameters. The parameters used were shown in Table IV. The default similarity threshold for search was set as 0.8 while that for join was set as 0.9. The reason was that similarity join required much more time to get results than search. For partition number N, we respectively selected 4096 and 1296 as default number for similarity search and join. We sampled 25% of the data for similarity join, because the baselines were rather slow on large datasets.

Cluster setup. All distributed experiments were conducted on a cluster with 1 master node and 6 slave nodes. The master node had an 8-core Intel(R) Xeon(R) E5420 @ 2.50GHz processor and 40GB main memory reserved for Spark. Every slave node consisted of 40-core Intel(R) Xeon(R) CPU E5-2630 v4@2.20GHz and 124GB main memory. And each slave node was connected to a Gigabit Ethernet switch running Ubuntu 14.04.1 with Hadoop 2.7 and Spark 2.0.0.

B. Effectiveness of LCRS

To evaluate the effectiveness of our function LCRS, we designed three methods to compare LCRS with other seven

[3]https://pan.baidu.com/s/1snbqswd
[4]https://figshare.com/articles/Urban_Road_Network_Data
[5]http://more.datatang.com/en



(a) Comparison on User-study (b) Comparison on Cluster

(c) Varying Noise Proportion (d) Varying K

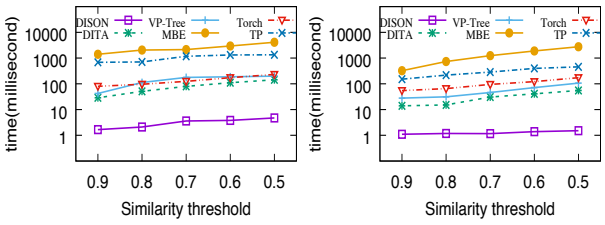
Fig. 5. Effectiveness Comparison with Other Functions

existing trajectory similarity functions DTW[35], LCSS[29], EDR[7], ERP[6], Jaccard on shared road segments(denoted as JAC)[33], spatio-temporal similarity STS [25] and LORS [31].

User Study. As there was no benchmark, we sampled 100 trajectories from Beijing dataset and manually found the most similar trajectory for each trajectory. As shown in Figure 5(a), LCRS could correctly find the most similar trajectory for 95% of trajectories while the accuracy of LCSS was only 70%.

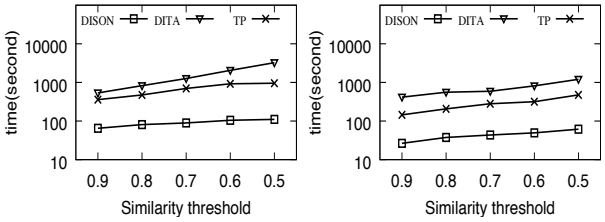
Noise-based method. We used the method in [24] to compare accuracy in noisy data between different functions. We constructed two dataset, $\mathcal{D}_1$ and $\mathcal{D}_2$, where $\mathcal{D}_1$ had 10,000 trajectories sampled from Beijing and $\mathcal{D}_2$ generated 10,000 trajectories by adding noise for each trajectory in $\mathcal{D}_1$. Specifically, we used one sample point of a trajectory as centre to construct a circle with 40 meter radius and then leveraged uniform sampling on the circle to get noisy point, which substituted the sample point. The reason of 40 meters radius was that the positioning accuracy of GPS was in the range 0-40 meters at the 95% confidence level according to [2]. Then we sampled 1,000 trajectories from $\mathcal{D}_1$ as query trajectories and utilized different functions to process KNN search in $\mathcal{D}_1$ and $\mathcal{D}_2$. Finally, we compared query results and counted the proportion of same query results between $\mathcal{D}_1$ and $\mathcal{D}_2$, which reflected the accuracy in noisy data. Notably, we added the noise by changing the position of trajectory points and the number of points was determined by a given noise proportion. Therefore, we explored the influence of noise proportion (the proportion of noisy points in a trajectory) and parameter K for KNN search, where the default value of noise ratio and K were set as 80% and 5. And the corresponding results were shown in Figure 5(c) and 5(d). In summary, LCRS and LORS were more effective than other functions for computing the similarity over trajectories, while LCRS was better than LORS, because LCRS utilized the length to normalize the distance.

Clustering-based method. We utilized another objective evaluation method proposed in [16], which leverages nearest neighbor (1NN) classifier [12], [28] to group labelled data into disjoint parts. To get labelled data, we generated 7 trajectories on Beijing Road Network and guaranteed any two trajectories were far away from each other (e.g. they have no shared road segments and the shortest path distance between their first points or last points were larger than 5,000 meters.), and thus we got 7 classes of trajectories. Then for each class we generated 1,000 trajectories with the form of points sequence by sampling and adding noise, which was similar to adding noise in the noise-based method. Trajectories sampled from the same class were marked with the same class, so each class consisted of 1,000 trajectories. As shown in Figure 5(b), we respectively selected 4, 5, 6 and 7 classes and computed the accuracy of clustering results using different functions. LCRS outperformed other distance functions by 5% to 20%.



(a) Beijing(1.5M) (b) Nanjing(1.5M)

Fig. 6. Efficiency Comparison with Centralized Baselines(Search)



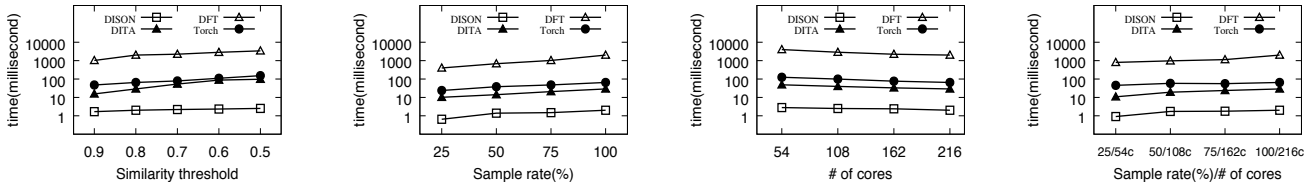
(a) Beijing(500K) (b) Nanjing(500K)

Fig. 7. Efficiency Comparison with Centralized Baselines(Join)

C. Comparison with Baselines

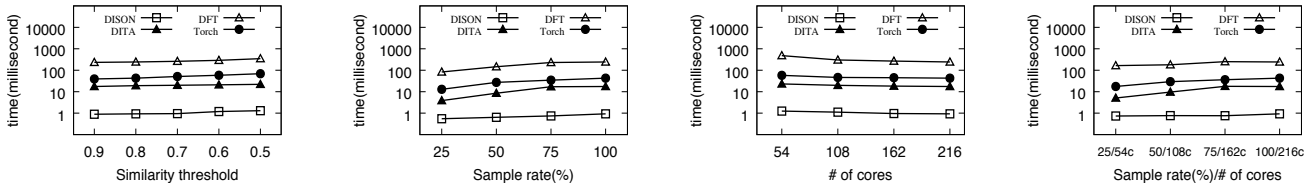
1) Centralized Trajectory Similarity Search: We first compared the performance of different centralized methods for trajectory similarity search on a single machine in our cluster using two small datasets with 1.5M trajectories, which were called Beijing(1.5M) and Nanjing(1.5M). For each dataset, we randomly sampled 1,000 queries and compared the average query latency. As shown in Figure 6, we made the following observations: (1) All methods took more time for smaller similarity thresholds. The intuition was that there were more relevant trajectories. (2) DISON outperformed other methods both on Beijing(1.5M) and Nanjing(1.5M) by 1-3 orders of magnitude. For example, when the similarity threshold was set as 0.8 on Beijing(1.5M), the latency was 2.1 milliseconds for DISON while the latency of DITA, VP-Tree, MBE, Torch and TP were respectively 51.1, 112.6, 2051.7, 32.3 and 706.3 milliseconds. The reasons were that DISON utilized effective LCRS to compute similarity and we reduced the computational complexity by replacing trajectories with intersection sequence when verifying candidate trajectories. In addition, our pruning algorithm with prefix and signatures can effectively prune most irrelevant trajectories. (3) All methods performed better on Nanjing(1.5M) than Beijing(1.5M), because there were more trajectories similar to query trajectories on Beijing(1.5M) than those on Nanjing(1.5M).

2) Centralized Trajectory Similarity Join: We also compared DISON with centralized baselines for trajectory joins on



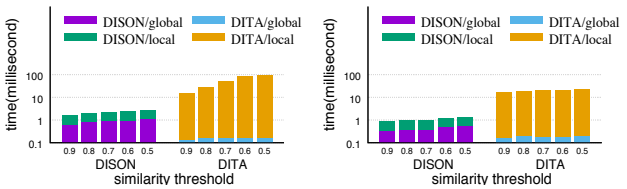
(a) Varying similarity: Beijing (b) Scalability: Beijing (c) Scale-up: Beijing (d) Scale-out: Beijing

Fig. 8. Comparison with Baselines on Beijing (Search)



(a) Varying similarity:Nanjing (b) Scalability: Nanjing (c) Scale-up: Nanjing (d) Scale-out: Nanjing

Fig. 9. Comparison with Baselines on Nanjing (Search)



(a) Search on Beijing (b) Search on Nanjing

Fig. 10. Query Latency in Global/Local Stage

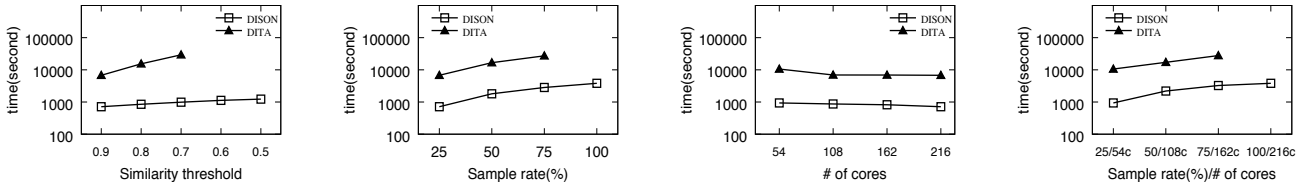
a single machine using two small datasets with 500K trajectories, which were called Beijing(500K) and Nanjing(500K). Since there were no effective join algorithms for `Torch`, `VP-Tree` and `MBE`, we only compared our methods with other two baselines. And for each dataset, we processed trajectory self-join and reported query latency. As shown in Figure 7, `DISON` outperformed `DITA` and `TP` on Beijing(500K) and Nanjing(500K). The reason were two-folds. First, `DITA` processed trajectory join between two trajectories sets by executing trajectory search for each trajectory in one set. This would cause duplicate computation especially for self-join. Second, `TP` generated candidate trajectories set for each trajectory and then refined trajectories in each candidate set. Although `TP` avoided repeated verification by removing refined trajectories from the candidate set, the candidate generation procedure still caused replicated calculation. In contrast, our methods utilized inverted index and prefix pruning to generate as little candidate pairs as possible. Meanwhile, we keep a candidate pair hashset to avoid generating replicated candidate pairs.

3) Distributed Similarity Search: We compared different distributed methods for trajectory similarity search on Beijing and Nanjing, and we randomly sampled 1,000 queries and evaluated the average query latency. Specifically, we compared these methods in four aspects: (1) The query efficiency by varying thresholds. (2) The scalability by varying dataset sizes. (3) *Scale-up* by varying number of cores. (4) *Scale-out* by varying both dataset sizes and the number of cores.

Efficiency. Based on results in Figure 8(a) and Figure 9(a), we made the following observations. (1) `DISON` outperformed other distributed methods on Beijing and Nanjing by 1-3 orders of magnitude. The reasons were three-folds. First, `DISON` used effective signature-based techniques to prune many dissimilar results. Second, `DFT` required two iterations for executing the whole search procedure. In the first iteration, `DFT` collected candidate trajectory IDs by pruning dissimilar trajectories. And in the second iteration, it rebuilt and verified the trajectories based on the dual index. Therefore, `DFT` took more time than `DISON`, `DITA` and `Torch`, which only included one iteration. Third, `DITA` used the minimal distance between pivot points and MBRs for global pruning, which was too loose to filter irrelevant partitions. In addition, the efficiency of local index mainly depended on the selections of pivot points. If the number of pivot points was small or there were many unused pivot points, the local pruning power would be decreased and the number of candidate trajectories would be more. In contrast, when the number of pivot points was more, it would take more time to get candidate trajectories. Although we extended `Torch` to support distributed trajectory search with our partition strategy and global pruning algorithms, the local pruning method in `Torch` cannot prune as many dissimilar trajectories as `DISON`. (2) The increase of latency in `DISON` was slower than `Torch`, `DITA` and `DFT` when the similarity threshold decreased, because we designed more effective pruning algorithms to filter dissimilar trajectories, and thus the increased number of candidate trajectories was less than `Torch`, `DITA` and `DFT`. As shown in Figure 10, `DISON` took less time in local stage, which proved the effectiveness of global pruning. As `DISON` had larger pruning power, the local stage latency of `DISON` was more stable than `DITA`.

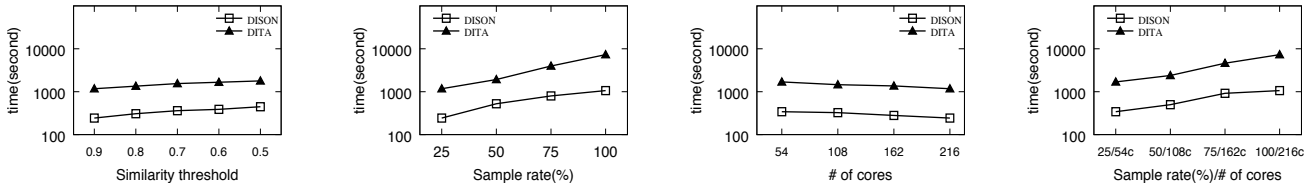
Scalability. According to the results in Figure 8(b) and Figure 9(b), we found that the performance gap between `DISON` and baselines increased with increasing data sizes due to our effective signature-based techniques. In addition, the query latency of `DISON` was stable for different sample rates. Especially, it only took 0.55, 0.64 and 0.74 milliseconds on Nanjing for the corresponding sample rate of 25%, 50% and 75%. The reason was that our well-designed algorithms can filter most dissimilar trajectories and the cost of query latency in `DISON` was mainly caused by global and local pruning, which was almost constant for different data size.

Scale-up. As shown in Figure 8(c) and Figure 9(c), it was obvious that the performance would be improved for all



(a) Varying similarity: Beijing (b) Scalability: Beijing (c) Scale-up: Beijing (d) Scale-out: Beijing

Fig. 11. Comparison with Baseline on Beijing (Join)



(a) Varying similarity:Nanjing (b) Scalability: Nanjing (c) Scale-up: Nanjing (d) Scale-out: Nanjing

Fig. 12. Comparison with Baseline on Nanjing (Join)

methods when we increased the number of cluster cores. Another observation was that DISON still outperformed other baselines. On the one hand, DFT cannot guarantee the balance of partitioning, which would alleviate the data skew problem when the workers reduced parallelism by decreasing the number of cluster cores. On the other hand, DISON, Torch and DITA aimed to partition similar trajectories into the same partitions. Therefore, the number of partitions for *Local Search* was less than DFT and the Spark jobs were correspondingly less. Moreover, each core would be assigned to more jobs when reducing cluster cores and the latency of each Spark job in DITA was proved to be bigger than that in DISON, which caused the faster drop of performance in DITA. For example, if we decreased the number of cores from 108 to 54 on Beijing, the cost of DITA was respectively 39.66 and 48.54 milliseconds while DISON only took 2.50 and 2.81 milliseconds. In addition, Torch needed to validate more trajectories than DISON in each candidate partition, which increased the latency of each Spark Job in Torch.

Scale-out. As shown in Figure 8(d) and Figure 9(d), almost all methods took more time when we increased data size and cluster cores. The reason was that all methods were implemented based on master-slave mode distributed system Spark [38], computing (e.g. global pruning, collecting results from slave nodes) on master node cannot be improved by increasing cluster cores. In contrast, the master node would be the bottleneck of the performance if the trajectory data size was more than a certain threshold. Importantly, the gap between DISON and baselines was also increased, which indicated that our method was more stable than baselines. That is, if given sufficient computing resources, DISON could process huge amount of trajectories more efficiently than baselines.

4) Trajectory Similarity Join: As DFT and Torch cannot support trajectory similarity join, we only compared DISON with DITA on Beijing and Nanjing. For each dataset, we processed trajectory self-join and reported query latency.

Efficiency. Figure 11(a) and Figure 12(a) showed the results. Unfortunately, DITA cannot finish join processing on Beijing within 10 hours when the threshold was less than 0.7, so we ignored the results of DITA. And we had the following observations: (1) Both DISON and DITA took more time to process trajectory similarity join with the decrease of similarity thresholds. It was clear that the less the similarity threshold was, the more the join results were. Therefore, global join would generate more candidate partition pairs to execute local join, and the local join would verify more candidate trajectory pairs if the threshold was smaller. (2) DISON outperformed DITA by 1-3 orders of magnitude both on Beijing and Nanjing. For example, when the threshold was set as 0.9, DISON only took 714.27 seconds Beijing while DITA took 6760.56 seconds. On the one hand, the pruning power of DITA was proved to be weaker than that of DISON, because we designed effective algorithm to select candidate partition pairs, which reduced Spark jobs executed on workers. (3) The gap between DISON and DITA increased with the decreasing of threshold. For instance, the threshold was 0.7, DISON took $1,229$ seconds on Beijing and DITA took $29,307$ seconds.

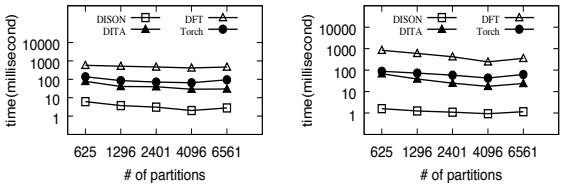
Scalability. As shown in Figure 11(b) and Figure 12(b), DISON was significantly better than DITA. Moreover, DITA seriously depended on the selection of pivot points, because few pivot points would reduce the pruning power while more pivot points would increase the pruning latency. For instance, Beijing was more complicated than Nanjing, because the trajectory data on Beijing required to use more pivots points to represent them for DITA, so DITA cannot process similarity join on Beijing efficiently.

Scale-up. As shown in Figure 11(c) and Figure 12(c), DISON and DITA gained better performance with the increase of the number of cores, because more workers indicated that there were more running jobs at the same time.

Scale-out. As shown in Figure 11(d) and Figure 12(d), DISON was better than DITA, and DITA could not efficiently handle larger dataset (especially complicated data) even given more workers, because DISON designed more effective algorithms to get candidate partition pairs.

D. Impacts of N

We compared the query latency of searching on Beijing and Nanjing between DISON and baselines by varying the number of partitions. As shown in Figure 13, both the performance on Beijing and Nanjing tended to be better for all methods by



(a) Search on Beijing (b) Search on Nanjing

Fig. 13. Varying the Number of Partitions

increasing the number of partitions. The reason was that the data size in each partition was decreased and thus the parallelism of computing on workers was improved by increasing the number of partitions. However, the degree of performance improvement slowly declined with the increase of partitions, when the amount of partitions exceeds a certain threshold. Because too many partitions means too many tasks and each task was too tiny, the bottleneck of the overall performance would be the cost of task initialization and results collection.

VI. Conclusions

We proposed a road-network-aware trajectory similarity function. We designed a filtering-refine framework to solve trajectory similarity search and join problem. We designed a scalable distributed framework for processing large-scale trajectories. We proposed an effective partitioning strategy, built global index for these partitions and local inverted index for each partition. We designed effective global pruning and local search algorithms to filter irrelative partitions and dissimilar trajectories. Extensive experiments on real datasets verified the effectiveness of our similarity function and efficiency and scalability of our centralized and distributed methods.

Acknowledgement

This work was supported by 973 Program of China (2015CB358700), NSF of China (61632016, 61472198, 61521002, 61661166012), Huawei, and TAL education.

References

[1] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. H. Saltz. Hadoop-gis: A high performance spatial data warehousing system over mapreduce. *PVLDB*, 6(11):1009–1020, 2013.

[2] M. A.Quddus, W. Y.Ochieng, and R. B.Nolandb. Current map-matching algorithms for transport applications: State-of-the art and future research directions. *Transportation Research Part C*, 15(5):3121–328, 2007.

[3] P. Bakalov, M. Hadjieleftheriou, E. J. Keogh, and V. J. Tsotras. Efficient trajectory joins using symbolic representations. In *MDM*, pages 86–93, 2005.

[4] P. Bakalov, M. Hadjieleftheriou, and V. J. Tsotras. Time relaxed spatiotemporal trajectory joins. In *GIS*, pages 182–191, 2005.

[5] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On map-matching vehicle tracking data. In *VLDB*, pages 853–864, 2005.

[6] L. Chen and R. T. Ng. On the marriage of lp-norms and edit distance. In *VLDB*, pages 792–803, 2004.

[7] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD*, pages 491–502, 2005.

[8] H. Ding, G. Trajcevski, and P. Scheuermann. Efficient similarity join of large sets of moving object trajectories. In *TIME*, pages 79–87, 2008.

[9] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE*, pages 1352–1363, 2015.

[10] E. Frentzos, K. Gratsias, and Y. Theodoridis. Index-based most similar trajectory search. In *ICDE*, pages 816–825, 2007.

[11] A. W. Fu, P. M. Chan, Y. Cheung, and Y. S. Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *VLDB J.*, 9(2):154–173, 2000.

[12] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques, 3rd edition*. Morgan Kaufmann, 2011.

[13] J. Jiang, C. Xu, J. Xu, M. Xu, N. Zheng, and K. Kong. Route planning for locations based on trajectory segments. In *SIGSPATIAL*.

[14] Y. Jiang, G. Li, J. Feng, and W. Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.

[15] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *PS*, page 29, 1995.

[16] E. J. Keogh and S. Kasetty. On the need for time series data mining benchmarks: A survey and empirical demonstration. *Data Min. Knowl. Discov.*, 7(4):349–371, 2003.

[17] J. Lee, J. Han, and X. Li. Trajectory outlier detection: A partition-and-detect framework. In *ICDE*, pages 140–149, 2008.

[18] J. Lee, J. Han, and K. Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD*, pages 593–604, 2007.

[19] Z. Li, J. Lee, X. Li, and J. Han. Incremental clustering for trajectories. In *DASFAA*, pages 32–46, 2010.

[20] Y. Lou, C. Zhang, Y. Zheng, X. Xie, W. Wang, and Y. Huang. Map-matching for low-sampling-rate GPS trajectories. In *SIGSPATIAL*, pages 352–361, 2009.

[21] W. Luo, H. Tan, L. Chen, and L. M. Ni. Finding time period-based most frequent path in big trajectory data.

[22] P. Newson and J. Krumm. Hidden markov map matching through noise and sparseness. In *SIGSPATIAL*, pages 336–343, 2009.

[23] M. Qiu and D. Pi. Mining frequent trajectory patterns in road network based on similar trajectory. In *IDEAL*, pages 46–57, 2016.

[24] S. Ranu, D. P, A. D. Telang, P. Deshpande, and S. Raghavan. Indexing and matching trajectories under inconsistent sampling rates. In *ICDE*, pages 999–1010, 2015.

[25] S. Shang, L. Chen, Z. Wei, C. S. Jensen, K. Zheng, and P. Kalnis. Trajectory similarity join in spatial networks. *PVLDB*, 10(11):1178–1189, 2017.

[26] Z. Shang, G. Li, and Z. Bao. Dita: Distributed in-memory trajectory analytics. In *SIGMOD*, 2018.

[27] N. Ta, G. Li, Y. Xie, C. Li, S. Hao, and J. Feng. Signature-based trajectory similarity join. *IEEE Trans. Knowl. Data Eng.*, 29(4):870–883, 2017.

[28] P. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.

[29] M. Vlachos, D. Gunopulos, and G. Kollios. Discovering similar multidimensional trajectories. In *ICDE*, pages 673–684, 2002.

[30] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. J. Keogh. Indexing multi-dimensional time-series with support for multiple distance measures. In *KDD*, pages 216–225, 2003.

[31] S. Wang, Z. Bao, J. S. Culpepper, Z. Xie, Q. Liu, and X. Qin. Torch: A Search Engine for Trajectory Data. In *SIGIR*, pages 535–544. ACM, 2018.

[32] C. Wenk, R. Salas, and D. Pfoser. Addressing the need for map-matching speed: Localizing globalb curve-matching algorithms. In *SSDBM*, pages 379–388, 2006.

[33] Y. Xia, G. Wang, X. Zhang, G. B. Kim, and H. Bae. Spatio-temporal similarity measure for network constrained trajectory data. *Int. J. Computational Intelligence Systems*, 4(5):1070–1079, 2011.

[34] D. Xie, F. Li, and J. M. Phillips. Distributed trajectory similarity search. *PVLDB*, 10(11):1478–1489, 2017.

[35] B. Yi, H. V. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE*, pages 201–208, 1998.

[36] S. You, J. Zhang, and L. Gruenwald. Large-scale spatial join query processing in cloud. In *ICDE*, pages 34–41, 2015.

[37] J. Yuan, Y. Zheng, C. Zhang, X. Xie, and G. Sun. An interactive-voting based map matching algorithm. In *MDM*, pages 43–52, 2010.

[38] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX*, pages 15–28, 2012.

[39] D. Zhang, N. Li, Z. Zhou, C. Chen, L. Sun, and S. Li. ibat: detecting anomalous taxi trajectories from GPS traces. In *UbiComp*, pages 99–108, 2011.

[40] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu. SJMR: parallelizing spatial join with mapreduce on clusters. In *CLUSTER*, pages 1–8, 2009.

[41] R. Zhong, G. Li, K. Tan, L. Zhou, and Z. Gong. G-tree: An efficient and scalable index for spatial search on road networks. *IEEE TKDE*, 27(8):2175–2189, 2015.