DITA: Distributed In-Memory Trajectory Analytics

Zeyuan Shang
Tsinghua Univ., Brown Univ.
zeyuanxy@gmail.com

Guoliang Li
Tsinghua University
liguoliang@tsinghua.edu.cn

Zhifeng Bao
RMIT University
zhifeng.bao@rmit.edu.au

ABSTRACT

Trajectory analytics can benefit many real-world applications, e.g., frequent trajectory based navigation systems, road planning, car pooling, and transportation optimizations. Existing algorithms focus on optimizing this problem in a single machine. However, the amount of trajectories exceeds the storage and processing capability of a single machine, and it calls for large-scale trajectory analytics in distributed environments. The distributed trajectory analytics faces challenges of data locality aware partitioning, load balance, easy-to-use interface, and versatility to support various trajectory similarity functions. To address these challenges, we propose a distributed in-memory trajectory analytics system DITA. We propose an effective partitioning method, global index and local index, to address the data locality problem. We devise cost-based techniques to balance the workload. We develop a filter-verification framework to improve the performance. Moreover, DITA can support most of existing similarity functions to quantify the similarity between trajectories. We integrate our framework seamlessly into Spark SQL, and make it support SQL and DataFrame API interfaces. We have conducted extensive experiments on real world datasets, and experimental results show that DITA outperforms existing distributed trajectory similarity search and join approaches significantly.

ACM Reference Format:
Zeyuan Shang, Guoliang Li, and Zhifeng Bao. 2018. DITA: Distributed In-Memory Trajectory Analytics. In *SIGMOD/PODS '18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3183743

1 INTRODUCTION

With the development of mobile devices and positioning technology, trajectory data can be captured more accurately, where each trajectory is a sequence of geo-locations of a moving object. For example, a Uber car drives a passenger from a source location to a destination location. In every 10 seconds, the GPS embedded in the car reports a geo-location of the car, and the sequence of these locations forms a trajectory. With the increased popularization of online ride-hailing service, the ride-hailing companies collect more and more trajectory data. For instance, there have been 2 billion Uber trips taken up to July 2016 and 62 million Uber trips in July 2016[1]. Most importantly, trajectory analytics can benefit many real-world applications, e.g., frequent trajectory based navigation systems, road planning, car pooling, and transportation optimizations.

Existing algorithms focus on optimizing this problem in a single machine [7–10, 12, 18, 41]. However, the amount of trajectories exceeds the storage and processing capability of a single machine, and existing algorithms cannot be easily extended to efficiently support large-scale trajectory data analytics in distributed environments, because (1) *data locality problem*: since trajectories are distributed, it is challenging to design data partitioning and indexing techniques to reduce heavy data transmission cost; (2) *load balance*: it is challenging to balance the workload to make full use of the computation power of the entire cluster; (3) *easy-to-use interface*: it is challenging to provide full-fledged SQL-like trajectory analytics system; (4) *versatility to support various trajectory similarity functions*: there are various widely adopted similarity functions, classified as the non-metric ones like *DTW*[48], *LCSS*[41] and *EDR*[10], and the metric ones like Fréchet [4]. We observe that existing studies either support one or two of them, or define its own similarity function, while it is critical to support all these similarity functions in one system for various analytics purposes and scenarios.

To bridge the gap between the limited availability of large-scale trajectory analytics techniques and the urgent need for efficient and scalable trajectory analytics in real world, we develop a distributed in-memory system DITA with easy-to-use SQL and DataFrame API interfaces. First, for a trajectory T we propose to select some "representative points" as pivots, and use the pivots to compute a lower bound of the distance between the trajectory represented by those pivots and another trajectory Q. If such a lower bound is already larger than a threshold, then T and Q cannot be similar. We propose a trie-like indexing structure to index the pivots and design effective global index and local index, where the global index finds relevant data partitions that contain possible answers and the local index computes answers in each partition. We propose effective filter-verification algorithms to compute the answers, where the filter step uses a light-weight filter to prune a large number of dissimilar pairs and get a set of candidates and the verification step utilizes effective techniques to verify the candidates. We propose a weighted bi-graph cost model, employ graph orientation mechanism to coordinate the distributed join, and utilize load balancing mechanism to prevent from stragglers. All techniques can also support most of existing trajectory similarity functions.

In summary, we make the following contributions.
(1) We propose a full-fledge distributed in-memory trajectory analytics system DITA, which extends Spark SQL with non-trivial efforts, enables creating index over RDDs, and provides SQL and DataFrame API for trajectory analysis (Section 3).
(2) We judiciously select representative points as pivots for efficient result computation, design a trie-like structure to index the

[1] http://expandedramblings.com/index.php/uber-statistics/

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3183743

[2] Guoliang Li is the corresponding author.

pivot points and develop global and local index to effectively prune dissimilar trajectories. We support most of the widely adopted trajectory similarity functions in our system (Section 4).
(3) We propose a filter-verification framework. In the filter step, we use pivot points to estimate the similarity between two trajectories to prune dissimilar pairs efficiently. In the verification step, we devise effective verification techniques (Section 5).
(4) We devise cost-based optimization techniques to reduce the inter-worker transmission and balance the workload (Section 6).
(5) We conduct a comprehensive evaluation on real world datasets. The results show that DITA outperforms existing distributed trajectoriy search and join approaches significantly (Section 7).

2 PRELIMINARIES

2.1 Problem Formulation

Trajectory. A trajectory is a sequence of points generated from a moving object, defined as below.

Definition 2.1. A trajectory T is a sequence of points $(t_1, \cdots, t_m)$, where each point is a d-dimensional tuple.

We use T_i to denote a trajectory whose id is i, and we use T and T_i interchangeably if no ambiguity. We simply use T^j to denote the prefix of T up to the j-th point, and t_j to denote the j-th point of T.

For simplicity we assume each point is represented as a 2-dimensional tuple (*latitude*, *longitude*). Our method can be easily extended to support multi-dimensional data (e.g., $d \geq 3$).

Distance Function. According to many experimental evaluations on different similarity functions for trajectory and time series data [13, 43, 45], Dynamic Time Warping (DTW) [31] is recognized as the most robust and widely adopted one. Thus, in this paper we use DTW as the default distance function, and we will show how to support other distance functions in Appendix A.

Definition 2.2. Given two trajectories $T = (t_1, \cdots, t_m)$ and $Q = (q_1, \cdots, q_n)$, DTW is computed as below

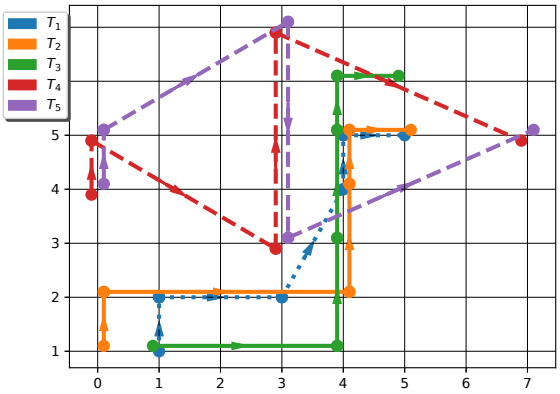
$$\texttt{DTW}(T,Q) = \begin{cases} \sum_{i=1}^{m} \texttt{dist}(t_i, q_1) & \text{if } n = 1 \\ \sum_{j=1}^{n} \texttt{dist}(t_1, q_j) & \text{if } m = 1 \\ \texttt{dist}(t_m, q_n) + \min\big(\texttt{DTW}(T^{m-1}, Q^{n-1}), & \\ \texttt{DTW}(T^{m-1}, Q), \texttt{DTW}(T, Q^{n-1})\big) & \text{otherwise} \end{cases}$$

where T^{m-1} is the prefix trajectory of T by removing the last point, and $\texttt{dist}(t_m, q_n)$ is the point-to-point distance between t_m and q_n (we use Euclidean distance in this paper).

Given two trajectories T and Q, we can utilize dynamic programming to compute $DTW(T, Q)$ with the time complexity $O(mn)$. We use a matrix (w) to store distance values, where $w_{i,j}$ represents $dist(t_i, q_j)$. Considering $T = T_1$ and $Q = T_3$ in Figure 1, the distance matrix is shown in Table 1. According to the definition of DTW, we construct a matrix (v) to store DTW values, where $v_{i,j}$ represents $DTW(T^i, Q^j)$. From the DTW matrix in Table 1, we have $DTW(T_1, T_3) = w_{1,1} + w_{2,1} + w_{3,2} + w_{4,3} + w_{5,4} + w_{5,5} + w_{6,6} = 5.41$.

Trajectory Similarity Search/Join. In this paper, we aim to solve trajectory similarity search and join problems.

Definition 2.3 (Trajectory Similarity). Given two trajectories T and Q, a trajectory-based distance function f(e.g., DTW) and a threshold τ, if $f(T, Q) \leq \tau$, we say that T and Q are similar.



Notation	Trajectory	Pivot Points($K = 2$)
T_1	(1, 1), (1, 2), (3, 2), (4, 4), (4, 5), (5, 5)	(3, 2), (4, 4)
T_2	(0, 1), (0, 2), (4, 2), (4, 4), (4, 5), (5, 5)	(4, 2), (4, 4)
T_3	(1, 1), (4, 1), (4, 3), (4, 5), (4, 6), (5, 6)	(4, 1), (4, 3)
T_4	(0, 4), (0, 5), (3, 3), (3, 7), (7, 5)	(3, 3), (3, 7)
T_5	(0, 4), (0, 5), (3, 7), (3, 3), (7, 5)	(3, 7), (3, 3)

Figure 1: Example Trajectories

(1) Point-to-point Distance

	t_1^3	t_2^3	t_3^3	t_4^3	t_5^3	t_6^3
t_1^1	0	3	3.61	5	5.83	6.40
t_2^1	1	3.16	3.16	4.24	5	5.66
t_3^1	2.24	1.41	1.41	3.16	4.12	4.47
t_4^1	4.24	3	1	1	2	2.24
t_5^1	5	4	2	0	1	1.41
t_6^1	5.66	4.12	2.24	1	1.41	1

(2) DTW

	t_1^3	t_2^3	t_3^3	t_4^3	t_5^3	t_6^3
t_1^1	0	3	6.61	11.61	17.44	23.84
t_2^1	1	3.16	6.16	10.40	15.40	21.06
t_3^1	3.24	2.41	3.83	6.99	11.11	15.59
t_4^1	7.48	5.41	3.41	4.41	6.41	8.65
t_5^1	12.48	9.41	5.41	3.41	4.41	5.83
t_6^1	18.14	13.54	7.65	4.41	4.83	5.41

Table 1: Distance and DTW Matirx for T_1 and T_3

Definition 2.4 (Trajectory Similarity Search). Given a query trajectory Q, a collection of trajectories $\mathcal{T} = \{T_1, T_2, \cdots, T_{|\mathcal{T}|}\}$, a trajectory-based distance function f(e.g., DTW) and a threshold τ, the trajectory similarity search problem finds all trajectories $T \in \mathcal{T}$, such that $f(T, Q) \leq \tau$.

Definition 2.5 (Trajectory Similarity Join). Given two collections of trajectories $\mathcal{T} = \{T_1, T_2, \cdots, T_{|\mathcal{T}|}\}$ and $\mathcal{Q} = \{Q_1, Q_2, \cdots, Q_{|\mathcal{Q}|}\}$, a trajectory distance function f(e.g., DTW) and a threshold τ, the trajectory similarity join problem finds all similar pairs $(T, Q) \in \mathcal{T} \times \mathcal{Q}$, such that $f(T, Q) \leq \tau$.

Example 2.6. Consider $\mathcal{T} = \{T_1, T_2, \cdots, T_5\}$ in Figure 1. We take T_1 as the query trajectory Q and use DTW as the distance function with $\tau = 3$, then the trajectories similar to Q are $\{T_1, T_2\}$.

2.2 Spark SQL Overview

Spark SQL [5] is a module in Apache Spark that enables relational processing (e.g., declarative queries) using Spark's functional programming API [15]. Spark SQL also provides a declarative DataFrame API to bridge between relational and procedural processing. It supports both external data sources (e.g., JSON, Parquet [2] and Avro [1]) and internal data collections (i.e., RDDs). Besides, it uses a highly extensible optimizer *Catalyst*, making it easy to add complex rules, control code generation, and define extension points. However, Spark SQL does not support trajectory analysis. In this paper, we integrate trajectory similarity search and join into Spark SQL, thus making it able to process trajectory queries efficiently with user-friendly front-end (e.g., SQL and DataFrame API).

2.3 Related Work

Trajectory Similarity Measures. Here we only highlight those widely adopted similarity functions for trajectory (or time-series)

data, including *dynamic time warping* (DTW) [48], *longest common subsequence distance* (LCSS) [41], *edit distance on real sequence* (EDR) [10], *edit distance with real penalty* (ERP) [9], DISSIM [18], and Fréchet distances [4]. According to several experimental evaluations on most of these functions [13, 43, 45], DTW is more systematically robust than other distance measures in general and widely adopted.

Trajectory Similarity Search/Join. Existing studies on trajectory similarity search focus on either the threshold based query [7, 8, 48] or the k nearest neighbor (kNN) query [9, 10, 12, 18, 41].

Xie et al. [46] develop a distributed in-memory system to answer KNN queries over trjaectories and support two metric distance functions, Hausdorff and Fréchet. They build global and local indexes based on segments of trajectories, and use bitmap and dual indexing to boost the search performance. Our work is different from [46] in four main aspects. (1) The similarity join studied in our work is not supported in [46]. (2) The performance of the indexing scheme of [46] heavily depends on the choice of distance functions, i.e., it does not support efficient search for non-metric distances such as DTW and LCSS, while our work supports both metric and non-metric distances in an efficient way (as validated in Section 7.2). (3) Our work studies the load balancing mechanism and the optimization of the result verification process, and constructs a cost model for distributed computing, which is ignored in [46]; instead, they simply partition the data evenly. (4) The authors in [46] build indexes on top of segments of trajectories, which require combining segments to calculate the actual distance between trajectories (similar to a non-clustered index). To address this, they utilize roaring bitmap and dual indexing, which increase the space overhead greatly because they have to store dissimilar trajectory IDs in the roaring bitmap and store the second copy of data with dual indexing, thus making it consume much more memory and takes more time to be replicated to every worker node. More importantly, the isolation of index and data reduces the parallelism because the master node has to wait for all the returned bitmaps from local indexes on all workers, merge them together and replicate the merged one to all workers for verification. In contrast, we build a clustered index where trajectories are directly stored and aligned with the index, thus we can validate the similarity on-the-fly.

[17] provides a distributed solution to KNN joins on Map-Reduce. [17] differs from our work in three aspects. (1) Their methods are not in-memory and they use Map-Reduce. (2) The distance functions are different: the distance between two trajectories is defined as the minimal distance between all pairs of points (p_1, p_2), where $p_1 \in T_1$, $p_2 \in T_2$, and p_1 and p_2 are in the same time interval. (3) [17] does not use any index to speed up the performance, and only utilizes a hash function to achieve load balancing.

In centralized settings, depending on the choice of distance functions, various pruning techniques are proposed to boost the search/join performance. Ding et al. [12] propose an R-tree based index for a distance function adapted from the classic Fréchet distance. Bakalov et al. [7, 8] utilize symbolic representations to achieve efficient join with an EDR-like similarity. Vlachos et al. [42] split the trajectories in multidimensional MBRs and store them in an R-tree, then propose the upper and lower bound to prune dissimilar results for two non-metric distances, LCSS and DTW. In [19, 40, 49] vantage point trees are proposed to answer trajectory similarity search/join for metric distance functions (e.g., Fréchet). Ranu et al. [33] utilize a universe set of vantage points as a fast estimation to their self-proposed non-metric EDwP distance function. Our work differs from them in three aspects. (1) Most previous pruning methods are devised for a specific similarity function, which are hard to extend to support other functions, while our work can handle DTW, LCSS, EDR and Fréchet. (2) The indexing method is different. Most of them use a single-level spatial index (e.g., R-tree) to index the trajectories [16, 21, 42]; while we propose a trie-like multi-level index, which accumulates the trajectory distance level by level, thus achieving high pruning power. (3) They are designed for a single machine and it is non-trivial to extend their methods to work in a distributed environment.

Trajectory Analytics. There are some studies on trajectory analytics, including trajectory clustering [20, 24, 26, 32, 36, 37], outlier detection [22, 27], classification [23, 35], simplification [28–30], trajectory storage[11, 44]. Zheng et al. [53] provide a complete review of related works on trajectory analytics.

Distributed/Parallel Spatial/Temporal Analytics. There are some distributed systems for spatial and temporal analytics. SpatialHadoop [14] and Hadoop GIS [3] are two distributed spatial data analytics systems over MapReduce. Clost [38] is a Hadoop-based storage system for spatio-temporal analytics. Some studies focus on distributed spatial join [50, 52]. GeoSpark [51] extends Spark for processing spatial data but only supports two-dimensional data without in-kernel indexing support and programming interface such as SQL or the DataFrame API. Simba [47] extends Spark SQL to support rich spatial queries and analytics through both SQL and the DataFrame API. However, these studies do not support trajectory analytics. Ray et al. [34] implement a parallel multi-core single machine method to join trajectories with spatial objects (e.g., polylines,polygons), rather than trajectory join studied in this paper.

3 OVERVIEW OF THE DITA SYSTEM

Extended SQL. We extend Spark SQL to support trajectory similarity search and join.

(1) Trajectory Similarity Search. Users can utilize the following query to find trajectories in table $\mathcal{T}$ that are similar to the query trajectory Q w.r.t. a function f and a threshold τ.

`SELECT * FROM` $\mathcal{T}$ `WHERE` $f(\mathcal{T}, Q) \leq \tau$

(2) Trajectory Similarity Join. Users can utilize the following query to find trajectory pairs (T, Q) in tables $\mathcal{T}$ and $\mathcal{Q}$ where $T \in \mathcal{T}$ is similar to $Q \in \mathcal{Q}$ w.r.t. a function f and a threshold τ.

`SELECT * FROM` $\mathcal{T}$ `TRA-JOIN` $\mathcal{Q}$ `ON` $f(\mathcal{T}, \mathcal{Q}) \leq \tau$

DataFrame. In addition to the extended SQL syntaxes, users can perform these operations over DataFrame objects using a domain-specific language similar to R. We also extend Spark's DataFrame API to support trajectory similarity search and join.

Index. We extend Spark SQL to support index construction for trajectory similarity search and join. Users can utilize the following query to create a trie-like index (including both global and local index) on table $\mathcal{T}$, which will be introduced in Section 4.

`CREATE INDEX` TrieIndex `ON` $\mathcal{T}$ `USE TRIE`.

Query Processing. Given a SQL query or DataFrame API request, the system firstly transforms it into a logical plan, and then optimizes it with rule-based optimizations (e.g., predicate pushdown, constant folding). Afterwards, the framework generates the most effective physical plan by applying both our cost-based optimizations and Spark SQL internal optimizations. The physical plan is executed on Spark to generate the results.

Query Optimization. We extend the Catalyst optimizer of Spark SQL and introduce a cost-based optimization (CBO) module to optimize trajectory similarity queries. The CBO module leverages the global and local index to optimize complex SQL queries, which will be discussed in Section 6.

4 INDEXING

We first present a method to select several pivot points from a trajectory T to approximately represent it. Then the DTW distance between a trajectory query Q and those pivot points of T is essentially a lower bound of the distance between Q and T. If the lower bound is already larger than a given threshold τ, then T cannot be similar to Q and thus T can be pruned. Next, we devise a trie-like structure to index the pivot points. Finally, we discuss how to implement a distributed index with our methods and propose a two-level (global and local) indexing scheme to reduce the global transmission cost and local computation cost.

4.1 Pivot Point Based DTW Estimation

4.1.1 Accumulative Distance Estimation. Based on Definition 2.2, to compute the DTW of T and Q based on the matrix v, we have to go from $(1,1)$ to (m,n) with a stride of 1. That is for each point $t_i \in T$ corresponding to the i-th row in the DTW matrix, we have to compute its distance with some points in Q and add it to the final DTW value when crossing the i-th row in the DTW matrix. Thus in the i-th row, we need to add a value $\texttt{dist}(t_i, q_j)$ into DTW. If we use the smallest value $\min_{1\le j\le n} \texttt{dist}(t_i, q_j)$, we can get a lower bound of DTW. In addition, in the first row and last row, DTW must include $\texttt{dist}(t_1, q_1)$ and $\texttt{dist}(t_m, q_n)$. Thus based on these two observations, we can get an *accumulated minimum distance*

$$\texttt{AMD}(T,Q) = \texttt{dist}(t_1,q_1) + \texttt{dist}(t_m,q_n) + \sum_{i=2}^{m-1} \min_{1\le j\le n} \texttt{dist}(t_i,q_j).$$

Since $\texttt{DTW}(T,Q) \ge \texttt{AMD}(T,Q)$, if $\texttt{AMD}(T,Q) > \tau$, then $DTW(T,Q) > \tau$, thus T and Q are not similar. The correctness is proved in Lemma 4.1.

Lemma 4.1. *For two trajectories T of length m and Q of length n, if $\texttt{AMD}(T,Q) > \tau$, then T and Q cannot be similar.*

However, it is still time-consuming to calculate the AMD distance since it shares the same time complexity $O(mn)$ with DTW. To speed up the estimation, we select several points as pivot points from T and compute the distance using these pivot points.

Definition 4.2 (Pivot Points). We define T_P as the set of *pivot points* of Q where $T_P \subset T \backslash \{t_1, t_m\}$. We compute the *pivot accumulated minimum distance* as

$$\texttt{PAMD}(T,Q) = \texttt{dist}(t_1,q_1) + \texttt{dist}(t_m,q_n) + \sum_{p\in T_P} \min_{1\le j\le n} \texttt{dist}(p,q_j).$$

If the PAMD distance between T and Q is larger than τ, i.e., $\texttt{PAMD}(T,Q) > \tau$, T and Q cannot be similar based on Lemma 4.3.

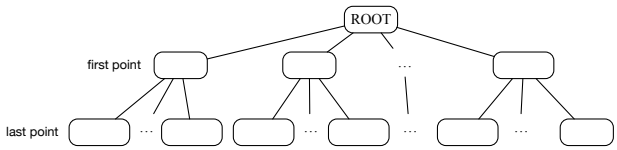


Figure 2: An Example of Partitioning

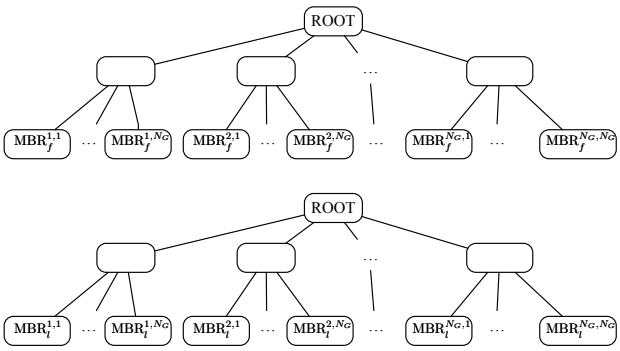


Figure 3: Global Index

Lemma 4.3. *For two trajectories T of length m and Q of length n, if $PAMD(T,Q) > \tau$, then T and Q cannot be similar.*

Using the first point, last point and the pivot points, we reduce the time complexity from $O(mn)$ to $O(nK)$, where K=$|T_P|$ and is much smaller than m.

Example 4.4. Given two trajectories T_1 and T_3 in Figure 1 with the number of pivot points $K = 2$ and $\tau = 3$, we have

$$PAMD(T_1,T_3) = \texttt{dist}(t_1^1,t_1^3) + \texttt{dist}(t_6^1,t_6^3) + \min_{1\le j\le 6} \texttt{dist}(t_3^1,t_j^3) +$$
$$\min_{1\le j\le 6} \texttt{dist}(t_4^1,t_j^3) = \texttt{dist}(t_1^1,t_1^3) + \texttt{dist}(t_6^1,t_6^3) + \texttt{dist}(t_3^1,t_2^3)$$
$$+ \texttt{dist}(t_4^1,t_3^3) = 0 + 1 + 1.41 + 1 = 3.41 > \tau$$

thus T_1 and T_3 cannot be similar.

4.1.2 Pivot Points Selection. Without loss of generality each point is assigned a weight, and our goal is to select K points with the largest weights as pivot points T_P. We fix the number of pivot points for all trajectories as K to make it more convenient for indexing and query processing. Next we propose three strategies for calculating a weight for each point in a trajectory T from different perspectives. Note our index and query processing methods are orthogonal to the choice of pivot selection strategies.

Inflection Point Strategy: for three consecutive points a, b and c in the trajectory, we use $\pi - \angle abc$ as the weight for point b. A large weight denotes that b is an inflection point and should be selected.

Neighbor Distance Strategy: for two consecutive points a, b in the trajectory, we use $\texttt{dist}(a,b)$ as the weight for the point b. A large weight denotes that b is far from a and should be selected.

First/Last Distance Strategy: for a point b in the trajectory T of length m, we use $\max(\texttt{dist}(b,t_1), \texttt{dist}(b,t_m))$ as the weight for the point b. A large weight denotes that b is far from the two endpoints and should be selected.

Consider trajectory T_1 in Figure 1 and the number of pivot points $K = 2$. For *Inflection Point Strategy*, its pivot points are $[(1,2),(4,5)]$; for *Neighbor Distance Strategy*, its pivot points are $[(3,2),(4,4)]$; for *First/Last Distance Strategy*, its pivot points are $[(1,2),(4,5)]$.

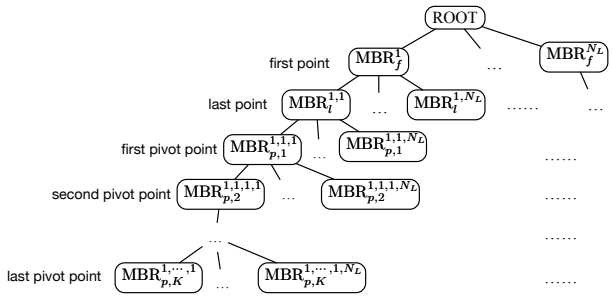


Figure 4: Local Index

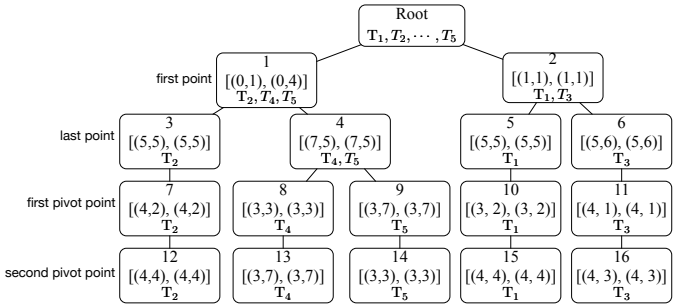


Figure 5: Example Trie-like Index with $K = 2$ pivot points

4.1.3 Pivot Points Based Pruning. Given a query Q, for each trajectory T, we enumerate each pivot point of T, find its nearest point in Q and compute $\texttt{PAMD}(T, Q)$. If $\texttt{PAMD}(T, Q) > \tau$, we can prune T. This method still needs to enumerate every trajectory. To address this issue, we can group the trajectories based on the pivots. Then we can prune a group if all the trajectories in the group are not similar to the query. We will discuss how to group the trajectories and how to build index for trajectories later.

4.2 Distributed Indexing

When building an index on a huge dataset of trajectories, at first we divide them into multiple partitions on different machines for distributed computation. Then DITA employs two levels of indexes: (1) the global index which helps the query trajectory Q find relevant partitions that may contain trajectories similar to Q; (2) the local index that helps Q find candidate trajectories in each relevant partition locally. Algorithm 1 presents the detailed steps.

4.2.1 Partitioning. As shown in Figure 2, we first group all trajectories by their first point into N_G disjoint buckets. Then we further group the trajectories in each bucket by the last point of these trajectories into N_G sub-buckets. Each sub-bucket is taken as a partition and thus there are $N_G * N_G$ partitions. In this way, similar trajectories are more likely to be in the same partition and each partition has roughly the same number of trajectories. We adopt the Sort-Tile-Recursive (STR) partitioning method [25] to partition the points of trajectories, where such a partitioning method can guarantee that each partition has roughly the same number of points, even for highly skewed data.

4.2.2 Global Index. For each partition, we get two minimum bounding rectangle (MBRs), MBR_f and MBR_l. MBR_f (MBR_l) is the MBR covering the first (last) point of all trajectories in that partition. Then we build an R-tree for all MBR_f and an R-tree for all MBR_l across all partitions, as shown in Figure 3.

Algorithm 1: Partition&Index($\mathcal{T}$)

Input: Trajectories $\mathcal{T}$: $T_1, T_2, \cdots, T_{|\mathcal{T}|}$;
Output: Partitions, Global Index, Local Index
// Partitioning
Group trajectories into N_G buckets based on the first points;
for *each bucket* **do**
  Group trajectories into N_G sub-buckets based on the last points; The points in each sub-bucket form a partition;
// Global Index
Build an R-tree RT_f for $N_G * N_G$ MBR_f;
Build an R-tree RT_l for $N_G * N_G$ MBR_l;
// Local Index
for *each partition(sub-bucket)* $\mathcal{T}_i$ **do**
  LocalIndex($\mathcal{T}_i$, root);

Function LocalIndex($\mathcal{T}$, TrieNode)

Input: Trajectories $\mathcal{T}$: $T_1, T_2, \cdots, T_{|\mathcal{T}|}$; TrieNode;
Output: Local Index
if *TrieNode is the root* **then**
  Group $\mathcal{T}$ into N_L buckets based on the first points;
  for *each bucket* $\mathcal{T}_i$ **do**
    Build a node *node* for $\mathcal{T}_i$ with its MBR_f;
    Add *node* as a child of TrieNode;
    LocalIndex($\mathcal{T}_i$, *node*);
else if *TrieNode is in the first level* **then**
  Group $\mathcal{T}$ into N_L buckets based on the last points;
  for *each bucket* $\mathcal{T}_i$ **do**
    Build a node *node* for $\mathcal{T}_i$ with its MBR_l;
    Add *node* as a child of TrieNode;
    LocalIndex($\mathcal{T}_i$, *node*);
else if *TrieNode is in x-th level,* $x < K + 2$ **then**
  Group $\mathcal{T}$ into N_L buckets based on (x−1)-th pivot point;
  for *each bucket* $\mathcal{T}_i$ **do**
    Build a node *node* for $\mathcal{T}_i$ with its MBR_{x-1};
    Add *node* as a child of TrieNode;
    LocalIndex($\mathcal{T}_i$, *node*);
return *LocalIndex*;

Let $\texttt{MinDist}(q, MBR)$ denote the minimal distance from a point q to an MBR (four corners and four sides). Given a query Q of length n, we use an R-tree to find MBR_f^i for each i, where $\texttt{MinDist}(q_1, MBR_f^i) \leq \tau$, and get the set of corresponding partitions C_f. Similarly, we utilize the other R-tree to find MBR_l^i for each i, where $\texttt{MinDist}(q_n, MBR_l^i) \leq \tau$, and get the set of corresponding partitions C_l. For each partition $p \in C_f \cap C_l$, if $\texttt{MinDist}(q_1, MBR_f) + \texttt{MinDist}(q_n, MBR_l) \leq \tau$, this is a relevant partition, where MBR_f and MBR_l are the corresponding first-point MBR and last-point MBR.

Space Complexity. The space complexity is $O(N_G^2)$.

4.2.3 Local Index. For each partition, we build a trie-like index for trajectories based on the pivot points, as shown in Figure 4.

For each trajectory T of length m in a partition, according to Lemma 4.3, we transform it into a sequence T_I of indexing points:

$$(t_1, t_m, t_{P_1}, t_{P_2}, \cdots, t_{P_K})$$

where $1 < P_1 < P_2 < \cdots < P_K < m$ and $(t_{P_1}, t_{P_2}, \cdots, t_{P_K})$ are pre-selected pivot points T_P. We define this sequence of points as the *indexing points* T_I of T.

Then we build a trie-like indexing structure based on the indexing points. We first initialize a dummy root node. Then we group the trajectories in each partition into N_L groups based on the first indexing point. For each group, we compute its MBR; as a result these N_L MBRs are the children of the root. Next, for the trajectories in each MBR, we further group them into N_L sub-groups based on the second indexing point, compute their MBRs and take them as the children. Iteratively we can build a $(K+2)$-level tree structure. For the leaf node, we also keep all the trajectories in the leaf MBR.

Thus in the local index, there are two types of trie nodes: (1) the *internal node* which stores its level, corresponding MBR, child nodes; (2) the *leaf node* which stores its level, corresponding MBR and trajectory data. Our trie index is similar to a B+-tree in that only leaf nodes store the real data, and keys (MBRs in our scenario) are stored in internal nodes. We call the MBRs at the first two levels as *align MBRs*, because they store the first and last point which must be aligned to the first and last point of the query. We call the MBRs at the bottom K levels as *pivot MBRs*.

In Figure 5, we build a trie index using trajectories $T_1, T_2, \cdots, T_5$ in Figure 1, with $N_L = 2$, $K = 2$ and *neighbor distance strategy* to select pivot points. Each node contains an MBR except for the root. For $node_1$, it contains an MBR $[(0,1),(0,4)]$, while $(0,1)$ is the bottom-left point and $(0,4)$ is the top-right point for an MBR.

Space Complexity. The space complexity of the local index is $O(N_L^{K+2} + |\mathcal{T}|)$, where $|\mathcal{T}|$ is the number of trajectories, as we store trajectory IDs in leaf nodes. N_L and K are usually small. For example, on a dataset with 10 million trajectories, N_L is 32 and K is 4. As the first two levels are *align MBRs*, we usually set a larger N_L because (1) in the upper level there are many trajectories and (2) we can get a tighter estimation using the *align MBRs*. As there are fewer trajectories at bottom levels, we set a smaller N_L in them.

5 TRAJECTORY SIMILARITY SEARCH

5.1 Framework

5.1.1 Basic Idea. In DITA, the trajectory similarity search queries are processed in three steps: (1) the master (called the *driver* in Spark) uses the global index to compute relevant partitions that contain trajectories similar to query Q, and sends Q to the corresponding workers (called the *executors* in Spark) of these partitions; (2) in each partition, workers first use the local index to generate candidate trajectories of query Q and then generate the local results by verifying whether they are actually similar to Q; (3) the master collects results and returns them to the user.

5.2 Global Pruning

Given a query Q with its first point q_1 and last point q_n, we find relevant partitions using the global index. We first use the R-tree for MBRs of the first point to find the MBRs $C_f = \{MBR_f | \texttt{MinDist}(q_1, MBR_f) \leq \tau\}$. Then we use the R-tree for MBRs of the last point to compute the MBRs $C_l = \{MBR_l | \texttt{MinDist}(q_n, MBR_l) \leq \tau\}$. Then for each MBR in $C_f \cap C_l$, if $\texttt{MinDist}(q_1, MBR_f) + \texttt{MinDist}(q_n, MBR_l) \leq \tau$, we send the query to the corresponding partition of this MBR, while other partitions can be pruned.

5.3 Local Search

5.3.1 Algorithm Overview. Given a query Q and a relevant partition, we use the trie index to find answers of Q in the partition. For ease of presentation, we first introduce some notations.

Minimal distance from a point to an MBR MinDist (q,MBR). The minimal distance from q to an MBR is the minimal distance from q to four corners and four sides of MBR ($\texttt{MinDist}(q, MBR) = 0$ if $q \in MBR$). We have $\texttt{MinDist}(q, MBR) \leq \texttt{MinDist}(q, p \in MBR)$.

Minimal distance from a trajectory to an MBR MinDist (Q,MBR). The minimal distance from Q to an MBR is the minimal distance of MinDist (Q,MBR) among all points $q \in Q$, i.e.,

$$\texttt{MinDist}(Q, MBR) = \min_{q \in Q} \texttt{MinDist}(q, MBR).$$

Aligned Point Matching. Recall Definition 2.2, since we start from $(1,1)$, the first points of two trajectories must be aligned. Therefore, the first point of Q, q_1, must be aligned to the MBRs in the first level. Thus we aim to find the MBRs from the first level whose minimal distance to q_1 is within τ, i.e., $\texttt{MinDist}(q_1, MBR_f) \leq \tau$. Similarly, since we end at (m,n), and the last point of Q, q_n, must be aligned to the MBRs in the second level. Thus we aim to find the MBRs from the second level whose minimal distance to q_n is within τ, i.e., $\texttt{MinDist}(q_n, MBR_l) \leq \tau$.

Pivot Point Matching. It is possible that the MBRs from the third level to the $(K+2)$-th level can match any point of Q. Therefore, we aim to find the MBRs from the x-th level ($x \in [3, K+2]$) of the Trie-like index whose minimal distance to Q is within τ, i.e., $\texttt{MinDist}(Q, MBR_{x-2}) \leq \tau$, where MBR_{x-2} corresponds to the $(x-2)$-th pivot point.

MBR-based Accumulated Minimum Distance. Consider a node MBR_{x-2} at the x-th level. Suppose its ancestor at i-th level is MBR_{i-2}. We consider the following cases to check whether we can prune the node.

If $x = 1$, we prune it if $\texttt{MinDist}(q_1, MBR_f) > \tau$

If $x = 2$, we prune it if $\texttt{MinDist}(q_1, MBR_f) + \texttt{MinDist}(q_n, MBR_l) > \tau$

If $x \geq 3$, we prune it if $\texttt{MinDist}(q_1, MBR_f) + \texttt{MinDist}(q_n, MBR_l) + \sum_{i=1}^{x-2} \texttt{MinDist}(Q, MBR_i) > \tau$

In other words, if MBR_{x-2} (corresponding to level x) is not pruned, we find its children MBR_{x-1} such that $\texttt{MinDist}(Q, MBR_{x-1}) \leq \tau - \texttt{MinDist}(q_1, MBR_f) - \texttt{MinDist}(q_n, MBR_l) - \sum_{i=1}^{x-2} \texttt{MinDist}(Q, MBR_i)$

Based on the above analysis, we design an index-based search method in Algorithm 2. We traverse the MBR-trie from the root. We first find its children MBR_f such that $d_f = \texttt{MinDist}(q_1, MBR_f) \leq \tau$. Then for each MBR_f, we find its children MBR_l such that $d_l = \texttt{MinDist}(q_n, MBR_l) \leq \tau - d_f$. Next for each MBR_l, we find its children MBR_1 such that $\texttt{MinDist}(Q, MBR_1) \leq \tau - d_f - d_l$. Iteratively we reach the leaf nodes. For each unpruned leaf node, the trajectories in the node are candidates. Then we verify the candidates.

Discussion. Our method can achieve high efficiency due to the following reasons. First, we build a multi-level trie-like index making it easy to calculate the accumulated distance. As along as the distance is gradually accumulated level by level, the threshold keeps decreasing and the pruning power of our method becomes stronger. Second, it is usually time consuming to compute trajectory distance (e.g., DTW) because we have to compute point-to-point distance numerous times (e.g., for DTW it is $O(mn)$). However, with our index we in fact

Algorithm 2: DITA-Search(Q, TrieIndex, τ)

Input: Query Q, TrieIndex, Threshold τ;
Output: Answers $\mathcal{A}$
Candidates C = DITA-Search-Filter(Q, root, τ);
for $T \in C$ **do**
  if *DITA-Search-Verify(Q, T)=true* **then**
    Answers $\mathcal{A} \leftarrow T$;

Function DITA-Search-Filter

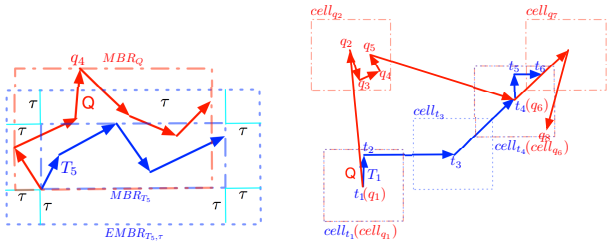
Input: Query Q, TrieNode MBR, Threshold τ;
Output: Candidates C
if *node is LeafNode* **then return** *trajectories in the node* ;
if *the node is the root* **then**
  for *each of its child* MBR_f *in the first level* **do**
    if $\texttt{dist}(q_1, MBR_f) \leq \tau$ **then**
      DITA-Search-Filter(Q,MBR_f,$\tau - \texttt{dist}(q_1, MBR_f)$);
else if *the node is in the first level* **then**
  for *each of its child* MBR_l *in the second level* **do**
    if $\texttt{dist}(q_n, MBR_l) \leq \tau$ **then**
      DITA-Search-Filter(Q,MBR_l,$\tau - \texttt{dist}(q_n, MBR_l)$);
else
  for *each of its child cMBR in the* $x \geq 3$ *level* **do**
    if $\texttt{dist}(Q, cMBR) \leq \tau$ **then**
      DITA-Search-Filter(Q, cMBR, $\tau - \texttt{dist}(Q, MBR)$);

Function DITA-Search-Verify

Input: Query Q, Trajectory T, Threshold τ;
Output: True or False
if MBR_T *not covered by* $EMBR_{Q,\tau}$ *or* MBR_Q *not covered by* $EMBR_{T,\tau}$ **then return** *false*; ;
if $CELL(T, Q) > \tau$ *or* $CELL(Q, T) > \tau$ **then return** *false* ;
Compute DTW(T,Q,τ);
if $DTW(T, Q, \tau) \leq \tau$ **then return** *true*;
else return *false*;

compute the point-to-rectangle distance (i.e., $\texttt{MinDist}(q, MBR)$) between a point and an MBR that covers a lot of points. In other words, we transform a large number of point-to-point distance computation to a one-time point-to-rectangle distance computation, thus reducing the computation workload greatly. Third, our trie index has a small size as justified in Appendix B, making it accessible to be replicated and fully kept in memory.

5.3.2 Optimizing Filtering. We propose effective techniques to optimize the filtering step. Considering an MBR of a pivot point, we compute $\texttt{MinDist}(Q, MBR)$ by enumerating all points in Q. Note that we only need to check a suffix of Q, because DTW has an ordering constraint. That is, in the cost matrix, starting from (i, j), we may only increase i or j or both by 1 in each step until reaching (m, n). Assuming that the first s points $q_1, q_2, \cdots, q_s$ are not within distance of $\tau_1 = \tau - \texttt{dist}(q_1, MBR_f) - \texttt{dist}(q_n, MBR_l)$ to the MBR of the first pivot point t_{P_1}, thus based on the above property t_{P_1} cannot be aligned with the first s points. In other words, t_{P_1} will be aligned with some points after the first s points, then the first s points must be aligned with some points $t_i \in T$ and $i < P_1$ due to



(a) Basic Idea for Lemma 5.4 (b) Basic Idea for Lemma 5.6

Figure 6: Optimizing Verification

the ordering constraint of DTW. Therefore we can discard the first s points because they cannot be aligned with $t_{P_2}, t_{P_3}, \cdots, t_{P_K}$ as well. Based on these observations, we propose to identify a suffix of Q to compute $\texttt{MinDist}(Q, MBR)$.

Consider the MBR for the first pivot point. We need to consider the entire Q and the suffix is $Q_1 = Q$. Then we find the longest prefix of Q, $q_1, q_2, \cdots, q_s$, such that $\texttt{dist}(q_i, MBR) > \tau$ for $1 \leq i \leq s$ and $\texttt{dist}(q_{s+1}, MBR) \leq \tau$. Then $Q_2 = q_{s+1}, \cdots, q_n$ is the suffix of Q_1 by removing the prefix of first s points. Iteratively, for Q_i, we compute the longest prefix of Q_{i-1} where the distance of each point in the prefix to the $(i-1)$-th MBR is larger than τ, and Q_i is the suffix of Q_{i-1} by removing the found prefix.

In this way, we only need to compute $\texttt{MinDist}(Q_i, MBR)$ instead of $\texttt{MinDist}(Q, MBR)$, and the correctness is proved in Lemma 5.1.

Lemma 5.1. *For trajectories T and Q with lengths of m and n, if $\texttt{dist}(q_1, MBR_f) + \texttt{dist}(q_n, MBR_l) + \sum_{i=1}^{K} \texttt{dist}(Q_i, MBR_i) > \tau$, T and Q cannot be similar. We designate the left part of this inequality as ordered pivot accumulated minimum distance (OPAMD).*

Example 5.2. Given a query trajectory $Q = T_4$ in Figure 1, we show how to query the trie index in Figure 5 with threshold $\tau = 3$. Firstly, we compute the distances between T_4's first point $(0, 4)$ and the root's child MBRs $node_1$ (dist = 0) and $node_2$ (dist = 3.16), then the new threshold is 3 for $node_1$ and $node_2$ is pruned. Secondly, we compute the distances between T_4's last point $(7, 5)$ and $node_1$'s child MBRs $node_3$ (dist = 2) and $node_4$ (dist = 0), then the new thresholds are respectively 1, 3. Thirdly, for $node_3$'s child MBR $node_7$, we compute its minimum distance with all points $q \in Q$, which is 1.41. Since the threshold for $node_3$ is 1, we prune $node_7$ now. Then for $node_4$'s child MBR $node_8$, the minimum distance is 0, and we further query $node_8$'s child MBR $node_{13}$, the minimum distance is still 0, thus T_4 is a candidate. For $node_4$'s another child MBR $node_9$, according to Lemma 5.1, as $\tau = 3$ and $\texttt{dist}(q_i, node_9) > \tau$ for $1 \leq i \leq 3$, we drop the first three points of Q and $\texttt{dist}(q_4, node_9) = 0$, and the threshold remains as 3. Further, for $node_9$'s child MBR $node_{14}$, the minimum distance is $\texttt{dist}(q_4, node_{14}) = 4 > \tau$, thus $node_{14}$ is pruned. Finally, we get T_4 as the final candidate.

5.3.3 Optimizing Verification. Given a candidate T of query Q, we verify whether T is similar to Q. It is expensive to directly compute the similarity between T and Q, and we propose efficient verification techniques.

(1) MBR Coverage Filtering. We define the MBR of a trajectory Q, denoted by MBR_Q, as the minimum bounded rectangle that covers the whole trajectory. For each point $t \in T$, if $\texttt{dist}(t, MBR_Q) > \tau$, T cannot be similar to Q, as formalized in Lemma 5.3.

Lemma 5.3. *Given two trajectories T and Q with lengths of m and n which are similar, then for any $t \in T$, $\texttt{dist}(t, MBR_Q) \leq \tau$.*

Note that in Lemma 5.3, if we swap T and Q, it is still correct. For each point $t \in T$, we compute $\texttt{dist}(t, MBR_Q)$ and prune candidate pair (T, Q) if it exceeds τ. However, this may introduce heavy computation if T and Q contain many points, as the time complexity is $O(m)$(or $O(n)$ if we swap T and Q). To solve this, we extend MBR_Q's borders by τ and denote the new rectangle as $EMBR_{Q,\tau}$. Then according to Lemma 5.4, if $EMBR_{Q,\tau}$ cannot fully cover MBR_T or $EMBR_{T,\tau}$ cannot fully cover MBR_Q, we can prune (T, Q). This technique reduces the time complexity from $O(n)$ to $O(1)$ and avoids expensive distance computation, as we only need to check rectangle coverage now. As shown in Figure 6(a), $EMBR_{T_5,\tau}$ cannot cover MBR_Q because $q_4 \in Q$ is not covered, from which we have $\texttt{dist}(q_4, T_5) > \tau$ and $\texttt{DTW}(T_5, Q) \geq \texttt{dist}(q_4, T_5) > \tau$.

Lemma 5.4. *For two trajectories T and Q, if T and Q are similar, $EMBR_{T,\tau}$ fully covers MBR_Q and $EMBR_{Q,\tau}$ fully covers MBR_T.*

Example 5.5. Let trajectory Q = [(0, 4), (0, 5), (3, 7), (3, 9), (3, 11), (3, 3), (7, 5)], whose pivot points are [(3, 7), (3, 3)], given a trajectory T_5 in Figure 1 and $\tau = 3$, we prove that $OPAMD(T_5, Q) \leq \tau$. However, $EMBR_{T_5,\tau}$ cannot fully cover MBR_Q, thus we prune (T_5, Q).

(2) Cell-based Compression. Although the MBR coverage filtering can prune candidate trajectory pairs with non-overlap MBRs, it does not work well on trajectories with overlap MBRs. To address this issue, we define a cell with a pre-defined side length D. For two trajectories T and Q, we compress each trajectory into a list of cells and compute a lower bound for $\texttt{DTW}(T, Q)$ using the two lists of cells with much less computation, thus avoiding the time-consuming DTW computation if the lower bound is beyond τ.

Formally, for each trajectory, we enumerate the points in the trajectory T. For the first point t_1, we create a cell with t_1 as its center with a fixed side length D. Then for each point $t \in T$, if it falls into some cell c, we will increase c's count (the number of covered points) by 1; otherwise we will create a new cell with t as its center. Finally, we will get a list of cells, which can be regarded as a compressed representation of the trajectory, and we denote T's cells as $\texttt{Cell}(T)$. Considering Definition 2.2, we find an easy-to-compute minimum distance for each point in T, sum them up and use the final summation as an estimation for DTW. After we have compressed T into $\texttt{Cell}_T$ and Q into $\texttt{Cell}_Q$, for each cell c_T in $\texttt{Cell}_T$, we compute $\min_{c_Q \in \texttt{Cell}_Q} \texttt{dist}(c_T, c_Q)$ as the minimum distance for all points $t \in c_T$, where $\texttt{dist}(c_T, c_Q)$ is the minimum distance between c_T and c_Q (if they are overlapped, it will be zero). As shown in Figure 6(b), for the second point in Q, we get a cell $\texttt{Cell}_{q_2}$ and four points falling in this cell. Thus we can compute $4 \times (\min_i \texttt{dist}(\texttt{Cell}_{q_2}, \texttt{Cell}_{t_i}))$ as an estimate instead of computing $\texttt{dist}(q, t_i)$ for $q \in \{q_2, q_3, q_4, q_5\}$. By computing this for each cell of Q and summing them up, we can get a lower bound estimate for $\texttt{DTW}(T, Q)$ in a much faster way as stated in Lemma 5.6.

Lemma 5.6. *For two given trajectories T and Q, let*

$$\texttt{Cell}(T, Q) = \sum_{c_T \in \texttt{Cell}_T} (\min_{c_Q \in \texttt{Cell}_Q} \texttt{dist}(c_T, c_Q)) * |c_T|,$$

$$\texttt{Cell}(Q, T) = \sum_{c_Q \in \texttt{Cell}_Q} (\min_{c_T \in \texttt{Cell}_T} \texttt{dist}(c_T, c_Q)) * |c_Q|,$$

where $|c_T|$ is the number of points falling into c_T. Then we have $DTW(T, Q) \geq \texttt{Cell}(T, Q)$ and $DTW(T, Q) \geq \texttt{Cell}(Q, T)$.

Example 5.7. Let a trajectory Q = [(1, 1), (1, 5), (1, 4), (2, 4), (2, 5), (4, 4), (5, 6), (5, 5)], whose pivot points are [(1, 5), (5, 6)], given a trajectory T_1 in Figure 1 and $\tau = 3$, we could prove that $OPAMD(T_1, Q) \leq \tau$. With pre-defined cell size $D = 2$, we could compress T_1 to [t_1, 2; t_3, 1; t_4, 3], where t_1, 2 represents a cell centered at t_1 containing 2 points, i.e., t_1 and t_2. We could compress Q to [q_1, 1; q_2, 4; q_6, 2; q_7, 1]. According to Lemma 5.6, $\texttt{DTW}(T_1, Q) \geq \texttt{Cell}(Q, T_1) = \texttt{dist}(\texttt{Cell}_{q_1}, \texttt{Cell}_{t_1}) * 1 + \texttt{dist}(\texttt{Cell}_{q_2}, \texttt{Cell}_{t_3}) * 4 + \texttt{dist}(\texttt{Cell}_{q_6}, \texttt{Cell}_{t_4}) * 2 + \texttt{dist}(\texttt{Cell}_{q_7}, \texttt{Cell}_{t_4}) * 1 = 0 + 1 * 4 + 0 + 0 = 4 > \tau$, thus we prune (T_1, Q).

(3) Double-Direction Verification. As DTW requires that the first points and the last points must be aligned, we can also start calculating from the last point, opposite to the first point. Thus we can compute from the first point and the last point simultaneously and add the values together to get the final DTW value. If the current sum is beyond τ, we can stop and conclude that they cannot be similar. This method will decrease the search space just as the double-direction search in the breadth-first search.

Overall Verification Algorithm. Computing MBRs and cells is pre-processed during creating the index. Since the MBR coverage filtering is most light-weight, we apply it first; then we apply cell-based pruning because they are easy to compute comparing with DTW. Note that we use a function $DTW(T, Q, \tau)$, which is an optimized version of DTW considering the threshold constraint and employing the double-direction verification.

6 TRAJECTORY SIMILARITY JOIN

6.1 Framework

Partitioning. When joining two tables $\mathcal{T}$ and $\mathcal{Q}$, DITA first builds indexes for them because it does not take too much cost as shown in Appendix B and it can also be reused for future computation. Therefore, in this section, we assume that both $\mathcal{T}$ and $\mathcal{Q}$ have already been indexed. While partitioning, DITA will ensure that the partition size cannot exceed half of the worker's memory such that any two partitions could be kept in memory at the same time.

Global Join. Given partitions of table $\mathcal{T}$ and $\mathcal{Q}$, DITA finds all partition pairs (i, j) such that there may exist trajectory $T \in \mathcal{T}_i$ and $Q \in \mathcal{Q}_j$ such that $\texttt{DTW}(T, Q) \leq \tau$, where $\mathcal{T}_i$ denotes the i-th partition of $\mathcal{T}$. Afterwards, DITA computes the join results between $\mathcal{T}_i$ and $\mathcal{Q}_j$ by either sending $\mathcal{T}_i$ to $\mathcal{Q}_j$ or sending $\mathcal{Q}_j$ to $\mathcal{T}_i$.

Local Join. For each pair $\mathcal{T}_i$ and $\mathcal{Q}_j$, without loss of generality, $\mathcal{Q}_j$ is sent to $\mathcal{T}_i$. For each trajectory $Q \in \mathcal{Q}_j$, DITA finds all trajectories $T \in \mathcal{T}_i$ such that $\texttt{DTW}(T, Q) \leq \tau$ by querying the index of $\mathcal{T}_i$.

6.2 Cost Model

Partition-Partition Bi-graph. For each pair $\mathcal{T}_i$ and $\mathcal{Q}_j$, we send them to some workers for local joins; however, it definitely transmits redundant data over the network because not all trajectories in $\mathcal{T}_i$ have candidates in $\mathcal{Q}_j$. To this end, for each trajectory $T \in \mathcal{T}_i$, we check whether T has candidates in $\mathcal{Q}_j$ by querying the global index of $\mathcal{Q}$, then we only send the trajectory $T \in \mathcal{T}_i$ that has candidates in $\mathcal{Q}_j$ rather than the full partition $\mathcal{T}_i$ to $\mathcal{Q}_j$, denoted as $\mathcal{T}_i \rightarrow \mathcal{Q}_j$. Similarly, we only send the trajectory $Q \in \mathcal{Q}_j$ that has candidates in $\mathcal{T}_i$ rather than the full partition $\mathcal{Q}_j$ to $\mathcal{T}_i$, denoted as $\mathcal{Q}_j \rightarrow \mathcal{T}_i$.

Then, we construct a directed bi-graph $\mathcal{G}$ between partitions of $\mathcal{T}$ and $\mathcal{Q}$. For $\mathcal{T}_i \rightarrow \mathcal{Q}_j$, we add an edge from $\mathcal{T}_i$ to $\mathcal{Q}_j$, or vice versa. For $\mathcal{T}_i \rightarrow \mathcal{Q}_j$, some trajectories will be sent from $\mathcal{T}_i$ to $\mathcal{Q}_j$, and we

denote the amount of data transmitted as $trans_{\mathcal{T}_i \to \mathcal{Q}_j}$. After these trajectories have been sent to $\mathcal{Q}_j$, DITA will invoke the local join, and we could use the total number of candidate pairs to estimate the computation workload for this, denoted as $comp_{\mathcal{T}_i \to \mathcal{Q}_j}$. DITA samples $\mathcal{T}$ and $\mathcal{Q}$ to estimate both $trans_{\mathcal{T}_i \to \mathcal{Q}_j}$ and $comp_{\mathcal{T}_i \to \mathcal{Q}_j}$. Therefore, we create a directed bi-graph $\mathcal{G}$, in which each edge has two weights: *trans* and *comp*.

By constructing such a bi-graph, DITA may orientate the directions of edges to achieve the best performance for distributed joins. Since a partition is the basic execution unit in Spark, we will propose a cost model for each partition in the following sections.

Network Cost. For each partition $\mathcal{T}_i$, we define the transmission cost of $\mathcal{T}_i$ as the amount of data in $\mathcal{T}_i$ sent to relevant partitions in $\mathcal{Q}$, i.e., the summation of $trans_{\mathcal{T}_i \to \mathcal{Q}_j}$ if the edge is orientated as from $\mathcal{T}_i$ to $\mathcal{Q}_j$ (out-degree edges for $\mathcal{T}_i$). Thus, we know the transmission cost for partition $\mathcal{T}_i$, $NC_{\mathcal{T}_i} = \sum_{\mathcal{T}_i \to \mathcal{Q}_j} trans_{\mathcal{T}_i \to \mathcal{Q}_j}$ and for partition $\mathcal{Q}_j$, $NC_{\mathcal{Q}_j} = \sum_{\mathcal{Q}_j \to \mathcal{T}_i} trans_{\mathcal{Q}_j \to \mathcal{T}_i}$.

Computation Cost. For each partition $\mathcal{T}_i$, we define the computation cost of $\mathcal{T}_i$ as the total number of candidate pairs during local join between trajectories sent from $\mathcal{Q}$ and partition $\mathcal{T}_i$, i.e., the summation of $comp_{\mathcal{Q}_j \to \mathcal{T}_i}$ if the edge goes from $\mathcal{Q}_j$ to $\mathcal{T}_i$ (in-degree edges for $\mathcal{T}_i$). Thus, we know the computation cost for partition $\mathcal{T}_i$ $CC_{\mathcal{T}_i} = \sum_{\mathcal{Q}_j \to \mathcal{T}_i} comp_{\mathcal{Q}_j \to \mathcal{T}_i}$ and for partition $\mathcal{Q}_j$, $CC_{\mathcal{Q}_j} = \sum_{\mathcal{T}_i \to \mathcal{Q}_j} comp_{\mathcal{T}_i \to \mathcal{Q}_j}$.

Total Cost. We define the total cost for partition $\mathcal{T}_i$ as $TC_{\mathcal{T}_i} = \lambda \cdot NC_{\mathcal{T}_i} + CC_{\mathcal{T}_i}$, where λ is a parameter to tune the transmission and computation speed. Assume the average computation time for a candidate pair is Δ and the network transmission bandwidth is $\mathcal{B}$. Then $\lambda = \frac{1}{\Delta \cdot \mathcal{B}}$.

Goal. Since partitions are executed in parallel, our goal is to reduce the maximum total cost among all partitions for balancing the workload. Thus we aim to select the direction of the edge for each pair $(\mathcal{T}_i, \mathcal{Q}_j)$ in the bi-graph $\mathcal{G}$ to minimize $TC_{global} = \max(\max_i TC_{\mathcal{T}_i}, \max_j TC_{\mathcal{Q}_j})$.

Problem Complexity. This problem is NP-hard. When $\lambda = 0$ in the aforementioned cost model, it becomes a so-called graph balancing or graph orientation problem, which has been proven as NP-Hard [6]. Therefore, we propose an approximation algorithm.

Greedy Algorithm. We employ a greedy algorithm to reduce TC_{global} iteratively. Initially, we decide the direction of edge $(\mathcal{T}_i, \mathcal{Q}_j)$ by comparing $\lambda \cdot trans_{\mathcal{T}_i \to \mathcal{Q}_j} + comp_{\mathcal{T}_i \to \mathcal{Q}_j}$ with $\lambda \cdot trans_{\mathcal{Q}_j \to \mathcal{T}_i} + comp_{\mathcal{Q}_j \to \mathcal{T}_i}$, if the former one is no larger than the latter, we choose $\mathcal{T}_i \to \mathcal{Q}_j$, or vice versa. Then in the rest iterations, we find the graph node (i.e., partition) with the maximum TC, enumerate its edges to find the edge that can reduce TC_{global} at most by changing direction, then change this edge's direction. We repeat this process until TC_{global} cannot be decreased. Finally, we will get an orientation plan Λ for the whole graph, from which DITA knows how to perform a join between $\mathcal{T}_i$ and $\mathcal{Q}_j$.

6.3 Division-based Load Balancing

While graph orientation could improve partition-level balancing, it does not work well in all cases since we only utilize an approximation algorithm. Further, even the optimal orientation cannot handle some cases. For example, if some partitions are inherently with huge *total cost* no matter how we change the directions of edges, orientation does not help. Therefore, we devise a dynamic load balancing mechanism by dividing the workloads of partitions among workers to take full advantage of parallel computing.

Algorithm 3: DITA-Join($\mathcal{T}$, $\mathcal{Q}$, τ)

Input: Table $\mathcal{T}$, Table $\mathcal{Q}$, Threshold τ;
Output: Answers $\mathcal{A}$
// Global Join
Construct the graph $\mathcal{G}$ by sampling $\mathcal{T}$ and $\mathcal{Q}$;
Apply approximation algorithms to select edges;
Apply division algorithm to replicate nodes in the graph $\mathcal{G}$;
for *each edge* $(\mathcal{T}_i \to \mathcal{Q}_j) \in \mathcal{G}$ **do**
  Send all trajectories $T \in \mathcal{T}_i$ to $\mathcal{Q}_j$ if T has candidates in $\mathcal{Q}_j$;
for *each edge* $(\mathcal{Q}_j \to \mathcal{T}_i) \in \mathcal{G}$ **do**
  Send all trajectories $Q \in \mathcal{Q}_j$ to $\mathcal{T}_i$ if Q has candidates in $\mathcal{T}_i$;
// Local Join
for *each trajectory* $T \in \mathcal{T}$ **do**
  Answers $\mathcal{A} \leftarrow$ DITA-Search(T, TrieIndex, τ);

Assume that after graph orientation, we sort partitions of both $\mathcal{T}$ and $\mathcal{Q}$ together in ascending order by their *total cost*, $\mathcal{P}_1$, ... $\mathcal{P}_N$, where N equals the sum of the number of partitions in $\mathcal{T}$ and $\mathcal{Q}$. We divide the workloads of partitions with huge *total cost* into multiple pieces. For example, given a partition $\mathcal{P}_i$, we replicate it to multiple copies, then trajectories supposed to be sent to $\mathcal{P}_i$ can be sent to different replicas, and trajectories sent from $\mathcal{P}_i$ can be sent from different replicas. In other words, we duplicate the node in $\mathcal{G}$ and assign its edges to different duplicates to reduce TC_{global}.

To implement this mechanism in DITA, it is essential to decide what partitions are *to-be-divided* partitions. In practice, since we sort partitions with its *total cost*, we choose up to 98% cost as the threshold for number of divisions for a partition, denoted as $TC_{0.98}$. If any partition with $TC_{\mathcal{P}_i}$ exceeding $TC_{0.98}$, DITA will replicate it to $\frac{TC_{\mathcal{P}_i}}{TC_{0.98}}$ copies to improve the parallelism.

6.4 Overall Algorithm

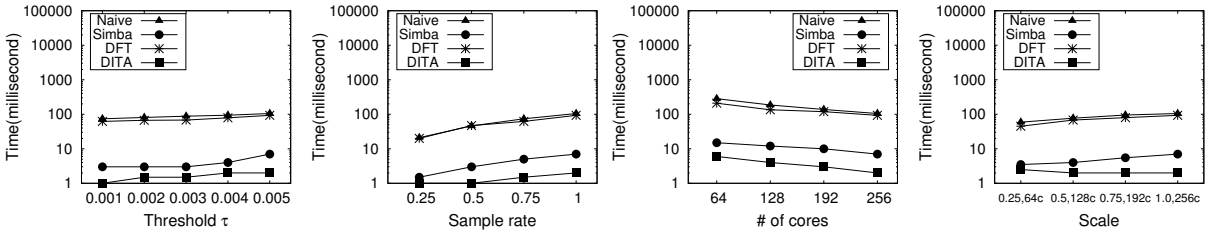
Algorithm 3 illustrates our global and local join methods. DITA creates the graph model by sampling $\mathcal{T}$ and $\mathcal{Q}$ to compute the weight of edge, then employs two aforementioned load balancing methods to modify the graph to achieve load balancing. Then DITA creates local join tasks according to the modified graph, from which each partition sends trajectories to other partitions if there exists an edge in the graph. For local join, for each trajectory sent to current partition, DITA utilizes the local trie index to search candidates.

7 EXPERIMENTS

7.1 Experimental Setup

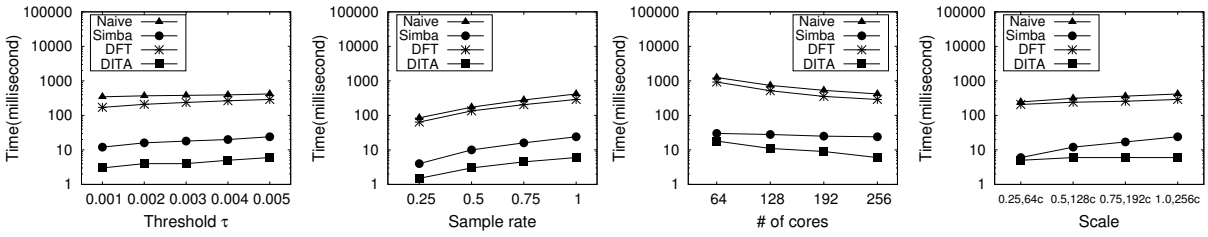
Dataset. Table 2 showed the statistics of datasets we used: Beijing, Chengdu, OSM(search), OSM(join). Beijing and Chengdu were collected from the GPS device on taxis in Beijing and Chengdu[3]. As there were no huge public trajectory datasets, OSM were synthesized from publicly available GPS traces (of various objects) from OpenStreeMap[4], which contained 8.7 million trajectories with 2.7 billion points (110 GB). We generated OSM(search) by dividing long

[3] http://more.datatang.com/en
[4] https://www.openstreetmap.org/



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

Figure 7: Comparison with Baselines on Beijing (Search) with the DTW Distance



(a) Varying τ: Chengdu (b) Scalability: Chengdu (c) Scale-up: Chengdu (d) Scale-out: Chengdu

Figure 8: Comparison with Baselines on Chengdu (Search) with the DTW Distance

Table 2: Datasets

Datasets	Cardinality	AvgLen	MinLen	MaxLen	Size
Beijing	11,114,613	22.2	7	112	10.4GB
Chengdu	15,316,372	37.4	10	209	28GB
OSM(search)	141,236,563	113.9	9	3000	703 GB
OSM(join)	65,764,358	119.5	9	3000	312 GB

Table 3: Parameters (Default value is highlighted)

Parameter	Value
Threshold τ	0.001, 0.002, 0.003, 0.004, **0.005**
N_G	32, **64(Beijing)**, **128(Chengdu)**, **256 (OSM)**
N_L	16, **32**, 64
Pivot Selection	InflectionPoint,**Neighbor**, First/LastDistance
Pivot Size K	2, 3, **4(Beijing)**, **5(Chengdu, OSM)**, 6
# of Cores	64, 128, 192, **256**
Dataset Size	0.25, 0.5, 0.75, **1.0**

trajectories (length > 3000) into several shorter ones; and (2) generating some trajectories by following the same distributions with the real ones. Since distributed join required much more memory than search, we generated OSM(join) by sampling OSM(search).

Baseline Methods. We compared with three baselines, including the naive method Naive without index, Simba which was extended from the in-memory spatial analytics system [47], and DFT which was extended from the most recent work on distributed in-memory trajectory search [46]. Simba supported range and kNN search on spatial points using R-tree on Spark. To make Simba support trajectory-based similarity search and join, we first indexed the first points of trajectories using Simba, and then used Simba to find trajectories whose first point was within a distance of τ from the query trajectory's first point as the candidates. Finally we verified the candidates to get the final answers. We extended DFT to support threshold-based search on DTW as stated in [46]. All the source codes were gotten from the authors.

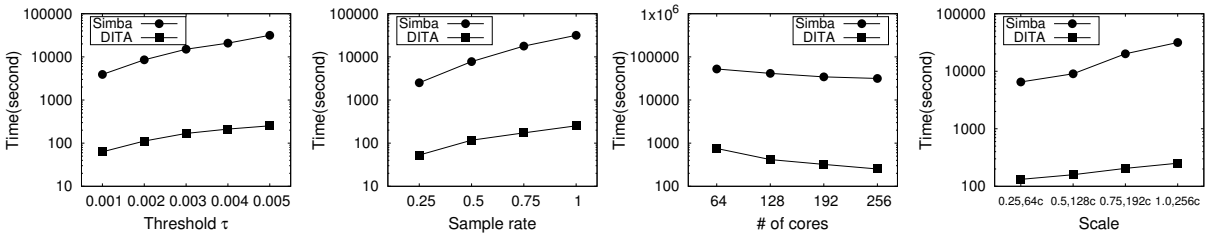
Parameters. Table 3 showed the parameters used in our method. When we varied a parameter, other parameters were set to default values (highlighted in bold). After a deep user study of trajectory similarity thresholds on Google Earth, we select $0.001, 0.002, \cdots, 0.005$ as thresholds, where 0.001 is roughly 111 meters.

Machines. All experiments were conducted on a cluster consisting of 64 nodes with a 8-core Intel Xeon E5-2670 v3 2.30GHz and 24GB RAM. Each node was connected to a Gigabit Ethernet switch and ran Ubuntu 14.04.2 LTS with Hadoop 2.6.0 and Spark 1.6.0.

7.2 Comparison with Baselines

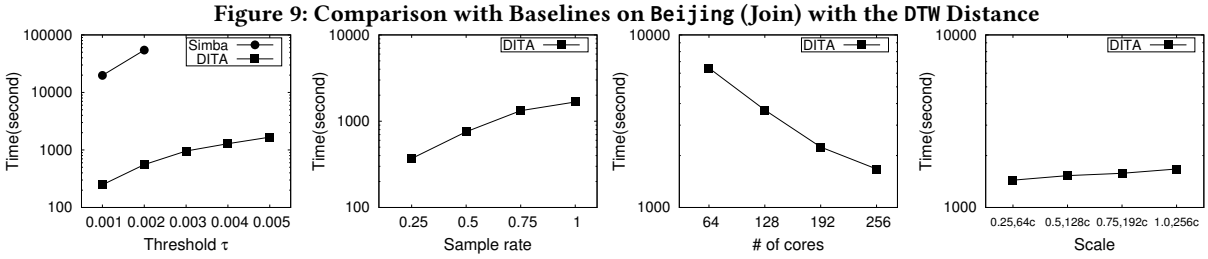
7.2.1 Trajectory Similarity Search. We first compared different methods for trajectory similarity search. For each dataset, we randomly sampled 1,000 queries from the dataset and reported the average running time in Figures 7 and 8.

Varying the Threshold. Figures 7(a) and 8(a) showed the performance of all methods when varying the threshold. We made the following observations. (1) With the increase of threshold, it took more time for all methods as a larger threshold led to more results. (2) Our method significantly outperformed baseline methods, even by one or two orders of magnitude. For example, when $\tau = 0.005$ on Chengdu, Naive took 418 milliseconds, Simba took 24 milliseconds, DFT took 289 milliseconds while DITA took 6 milliseconds. The reasons were four-fold: *i*) Simba, DFT and DITA employed a two-level (global and local) indexing scheme to reduce the transmission cost and filtered out dissimilar pairs while Naive did not; *ii*) DITA adopted an efficient trie-like index which was much better than R-tree used in Simba and DFT, because DITA was specially designed for trajectories and could compute accumulative distance level by level; *iii*) for DFT, it queried the index to get the bitmap for filtered trajectories, collected the bitmap at the master node, then searched the data with the collected bitmap to verify the similarity, thus making a barrier between indexing and verification, while both Simba and DITA queried the index and verified the similarity in the same iteration. Thus, DFT had less parallelism than Simba and DITA. Besides, DFT did not optimize the verification process, while Simba and DFT did. *iv*) DITA optimized the verification process with the MBR coverage filtering, cell-based pruning and double-direction verification (which are proposed in Section 5.3.3), while Naive and Simba only utilized the double-direction verification. (3)



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

Figure 9: Comparison with Baselines on Beijing (Join) with the DTW Distance



(a) Varying τ: Chengdu (b) Scalability: Chengdu (c) Scale-up: Chengdu (d) Scale-out: Chengdu

Figure 10: Comparison with Baselines on Chengdu (Join) with the DTW Distance

Naive, Simba and DFT were much more sensitive to the threshold as compared with DITA, as only DITA used accumulative distance estimation and efficient verification process. For example, when $\tau = 0.003$ on Beijing, Naive, Simba, DFT, DITA took 88, 3, 68, 1.5 milliseconds respectively; when τ was 0.005, they took 105, 7, 93 and 2 milliseconds respectively.

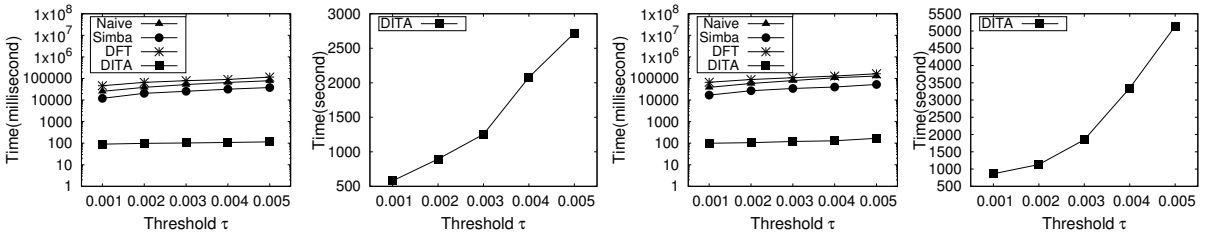
Scalability. We evaluated the scalability of all methods as shown in Figures 7(b) and 8(b) with varying dataset size. With the increase of data size, the gap between DITA and DFT, Simba, Naive tended to be bigger, since large dataset sizes introduced much more computation workloads and transmission costs. DITA achieved a better scalability with its efficient partitioning scheme, well-devised indexing, and optimized verification. For instance, if we increased the dataset size from 0.5 to 1.0 on Beijing, Naive spent from 42 to 105 milliseconds, Simba spent from 3 to 7 milliseconds, DFT spent from 47 to 93 milliseconds, while DITA only spent from 1 to 2 milliseconds.

Scale-up. We varied the number of cores and had the following observations from Figures 7(c) and 8(c). (1) With the increase of the number of cores, all methods gained performance improvement since there were more workers running in parallel. (2) DITA achieved the biggest performance gain, because DITA partitioned data by their first and last points such that similar trajectories tended to reside together and the size of each partition was nearly equal; then with more workers, there would not be severe data skew or imbalance problem. Simba only partitioned data by the first or last point, which may cause unbalanced workloads of partitions. Although DFT partitioned segments of trajectories, it had less parallelism due to the barrier between the process of index probing and verification. For example, when there were 128 cores on Chengdu, Naive took 739 milliseconds, Simba took 28 milliseconds, DFT took 518 milliseconds, while DITA took 11 milliseconds; if we increased it to 256 cores, they spent 418, 24, 289 and 6 milliseconds respectively.

Scale-out. We varied both the data size and the number of cores as shown in Figures 7(d) and 8(d)), where "0.25,64c" denoted that the dataset ratio was 0.25 and the number of cores was 64. We could observe that: (1) DITA scaled-out nearly linearly on both datasets while Naive, Simba, DFT could not handle bigger dataset well even with more workers. The reasons were in two folds: (*i*) DITA adopted efficient partition scheme to achieve data locality and load balancing; (*ii*) similarity search did not need to transmit much data between workers while similarity join required sending candidate trajectories through the network, thus the performance of search was directly related with local computation performance. (2) On the Beijing dataset, the latency decreased when we increased from "0.25,64c" to "0.5,128c", because on a small dataset, the query latency was tiny and the overheads (e.g., sending answers from workers to master) were relatively more severe.

7.2.2 Trajectory Similarity Join. As Naive was too slow to complete for similarity join, we would omit it. Note that DFT [46] only presented how to support trajectory search but did not discuss how to support join. We extended its search algorithm to support trajectory similarity join; however, it consumed too much memory on big datasets. The reason was as below. For each query, [46] constructed a bitmap to store the ids of dissimilar trajectories, although they used compressed roaring bitmap to reduce the space usage. On our smallest dataset Beijing (a 11-million dataset), for each (trajectory search) query, it took 0.2 MB on average. For trajectory join, there were 11-million queries, thus it took roughly 2.2 TB memory to store all the bitmaps for all queries. Thus DFT cannot support trajectory join for large datasets. We also tried the MapReduce-based distributed join [17] on Beijing, but it was not completed in 24 hours ([17] only conducted experiments on a dataset with one-million trajectories while our datasets were 1-2 orders of magnitude larger than it). Also note that their distance function was self-defined, and it was hard to extend their techniques to support those widely adopted trajectory distance functions. Thus, we compared Simba and DITA for join, as shown in Figures 9 and 10. As Simba was not completed in 24 hours on Chengdu when the threshold was larger than 0.002, we only showed partial results for Simba.

Varying the Threshold. Figures 9(a) and 10(a) evaluated the performance of two methods when varying threshold. We made the following observations. (1) DITA significantly outperformed Simba by one or two orders of magnitude especially when τ was big. For example, when $\tau = 0.005$ on Beijing, Simba took 31594 seconds



(a) Search Time on OSM(search) (DTW) (b) Join Time on OSM(join) (DTW) (c) Search Time on OSM (Fréchet) (d) Join Time on OSM(join) (Fréchet)

Figure 11: Evaluation on Large Datasets OSM(search) and OSM(join) with the DTW and Fréchet Distances

and DITA only spent 252 seconds. This was because that: *i*) DITA employed efficient trie indexing scheme which generated much less candidates than Simba; *ii*) DITA optimized time-consuming verification step while Simba did not; *iii*) DITA adopted graph orientation mechanism to reduce the transmission and computation cost globally, and the load balancing mechanism to prevent from stragglers while Simba did not. *iv*) during execution, Simba sent each partition to its relevant partitions while DITA sent each trajectory to its relevant partitions, thus DITA sent much less data. (2) Both methods took more time on Chengdu than Beijing, because Chengdu had more trajectories and thereby many more answers.

Scalability. We evaluated the scalability of the two methods in Figures 9(b) and 10(b) with varying the dataset size. As these two figures suggested, DITA achieved nearly linear scalability, and the performance gap between DITA and Simba tended to be bigger when the dataset size was increasing. This was because DITA employed more efficient indexing scheme, adopted graph orientation mechanism to reduce transmission and computation cost, and utilized load balancing mechanism to prevent from stragglers. For example, when we increased the dataset size from 0.5 to 1.0 on Beijing, Simba took from 7761 seconds to 31594 seconds, while DITA took from 117 seconds to 252 seconds.

Scale-up. We varied the number of cores and had the following observations from Figures 9(c) and 10(c). (1) With the increase of the number of cores, both methods gained performance improvement since there were more workers, but not as much as similarity search, because for search, all workers were running in parallel with little inter-worker communication, while for join, workers were supposed to send candidate trajectories to each other. (2) DITA achieved higher performance gain with the increasing number of workers, because DITA employed graph orientation mechanism to reduce cost and balancing mechanism to distribute workloads more evenly. For example, when we had 128 cores on Beijing, Simba spent 41453 seconds and DITA spent 416 seconds; if we increased it to 256 cores, Simba took 31594 seconds and DITA took 252 seconds.

Scale-out. We varied the dataset size and number of cores as shown in Figures 9(d) and 10(d). We observed that DITA scaled-out well on both datasets while Simba could not handle bigger dataset even with more workers. The reasons were four-folds. (1) with more cores and bigger data size, there would be more network transmission between workers, thus requiring efficient global indexing methods to prune irrelevant partitions, and in this case DITA achieved better performance; besides, DITA employed graph orientation mechanism to further reduce network transmission cost; (2) DITA generated much less candidates than that of Simba, leading to that the local computation workload for DITA was also smaller than Simba; (3) with more cores (workers), the imbalance between workers would be more intense, and DITA was optimized with load balancing mechanism while Simba did not; (4) Simba processed join by matching partition to partition, while DITA matched each trajectory to a partition, thus DITA sent much less data.

7.3 Evaluation on Large Datasets

We evaluated the performance of all methods for trajectory similarity search and join on the large datasets OSM by varying the threshold, and the result is shown in Figure 11. Since all methods except DITA were too slow to complete in 24 hours on the OSM(join) dataset, we only reported the results of DITA for distributed trajectory join. We had five observations. (1) DITA significantly outperformed other methods for distributed search. For example, existing works took more than 10 seconds on search while DITA only took 0.1 second. (2) The join performance of DITA was more sensitive to the threshold value, as more overheads were incurred for join than search. (3) Join on OSM(join) was only three times slower than join on Chengdu, even though OSM(join) was one order of magnitude larger than Chengdu. This was because OSM(join) was a dataset of worldwide trajectories and had smaller numbers of candidates and results, while Chengdu only contained citywide trajectories. (4) For both search and join, the efficiency on Fréchet distance function was slower than that of DTW because DTW was tighter than Fréchet with the same threshold. (5) Our method was able to scale well on huge datasets while existing approaches cannot support huge datasets. This was attributed to our effective global and local indexes, effective filtering algorithms and verification techniques.

8 CONCLUSION

In this paper we proposed DITA, a distributed in-memory trajectory analytics system on Spark. DITA provided the trajectory similarity search and join operations with a user-friendly SQL and DataFrame API, and can work with most trajectory similarity functions such as non-metric ones (DTW, EDR, LCSS) and non-metric ones (Fréchet distance). DITA built effective global index and local index to prune irrelevant partitions and trajectories. Furthermore, DITA optimized the verification step to reduce the computation cost, and adopted load balancing mechanisms to balance the workload. Extensive experiments showed that DITA outperformed state-of-the-art distributed approaches significantly. In future, we plan to support KNN-based search and join in DITA, and implement an extension of DITA by considering road networks.

Acknowledgement. Guoliang Li was supported by the 973 Program of China (2015CB358700), NSF of China (61632016,61472198, 61521002,61661166012), and TAL education. Zhifeng Bao was supported by ARC (DP170102726, DP180102050), NSF of China (61728204, 91646204), and Google Faculty Award.

REFERENCES

[1] Apache avro. https://avro.apache.org/.
[2] Apache parquet. https://parquet.apache.org/.
[3] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop gis: a high performance spatial data warehousing system over mapreduce. *PVLDB*, 6:1009–1020, 2013.
[4] H. Alt and M. Godau. Computing the fréchet distance between two polygonal curves. *Int. J. Comput. Geometry Appl.*, 5:75–91, 1995.
[5] M. Armbrust, R. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: relational data processing in spark. In *SIGMOD*, pages 1383–1394, 2015.
[6] Y. Asahiro, E. Miyano, H. Ono, and K. Zenmyo. Graph orientation algorithms to minimize the maximum outdegree. *Int. J. Found. Comput. Sci.*, 18:197–215, 2006.
[7] P. Bakalov, M. Hadjieleftheriou, E. J. Keogh, and V. J. Tsotras. Efficient trajectory joins using symbolic representations. In *Mobile Data Management*, pages 86–93, 2005.
[8] P. Bakalov, M. Hadjieleftheriou, and V. J. Tsotras. Time relaxed spatiotemporal trajectory joins. In *GIS*, pages 182–191, 2005.
[9] L. Chen and R. T. Ng. On the marriage of lp-norms and edit distance. In *VLDB*, pages 792–803, 2004.
[10] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD*, pages 491–502, 2005.
[11] P. Cudré-Mauroux, E. Wu, and S. Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *ICDE*, pages 109–120, 2010.
[12] H. Ding, G. Trajcevski, and P. Scheuermann. Efficient similarity join of large sets of moving object trajectories. In *TIME*, pages 79–87, 2008.
[13] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. J. Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *PVLDB*, 1:1542–1552, 2008.
[14] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE*, pages 1352–1363, 2015.
[15] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: fast data analysis using coarse-grained distributed memory. In *SIGMOD*, pages 689–692, 2012.
[16] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, 1994.
[17] Y. Fang, R. Cheng, W. Tang, S. Maniu, and X. S. Yang. Scalable algorithms for nearest-neighbor joins on big trajectory data. *ICDE*, pages 1528–1529, 2016.
[18] E. Frentzos, K. Gratsias, and Y. Theodoridis. Index-based most similar trajectory search. In *ICDE*, pages 816–825, 2007.
[19] A. W.-C. Fu, P. M. shuen Chan, Y.-L. Cheung, and Y. S. Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *VLDBJ*, 9:154–173, 2000.
[20] S. Gaffney and P. Smyth. Trajectory clustering with mixtures of regression models. In *KDD*, pages 63–72, 1999.
[21] E. J. Keogh. Exact indexing of dynamic time warping. In *VLDB*, 2002.
[22] J.-G. Lee, J. Han, and X. Li. Trajectory outlier detection: A partition-and-detect framework. In *ICDE*, pages 140–149, 2008.
[23] J.-G. Lee, J. Han, X. Li, and H. Gonzalez. Traclass: trajectory classification using hierarchical region-based and trajectory-based clustering. *PVLDB*, 1:1081–1094, 2008.
[24] J.-G. Lee, J. Han, and K.-Y. Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD*, pages 593–604, 2007.
[25] S. T. Leutenegger, J. M. Edgington, and M. A. López. Str: A simple and efficient algorithm for r-tree packing. In *ICDE*, pages 497–506, 1997.
[26] Z. Li, B. Ding, J. Han, and R. Kays. Swarm: Mining relaxed temporal moving object clusters. *PVLDB*, 3:723–734, 2010.
[27] Z. Li, M. Ji, J.-G. Lee, L. A. Tang, Y. Yu, J. Han, and R. Kays. Movemine: mining moving object databases. In *SIGMOD*, pages 1203–1206, 2010.
[28] X. Lin, S. Ma, H. Zhang, T. Wo, and J. Huai. One-pass error bounded trajectory simplification. *PVLDB*, 10:841–852, 2017.
[29] C. Long, R. C.-W. Wong, and H. V. Jagadish. Direction-preserving trajectory simplification. *PVLDB*, 6:949–960, 2013.
[30] C. Long, R. C.-W. Wong, and H. V. Jagadish. Trajectory simplification: On minimizing the direction-based error. *PVLDB*, 8:49–60, 2014.
[31] C. S. Myers and L. R. Rabiner. A comparative study of several dynamic time-warping algorithms for connected-word recognition. *Bell System Technical Journal*, 60:1389–1409, 1981.
[32] N. Pelekis, I. Kopanakis, E. E. Kotsifakos, E. Frentzos, and Y. Theodoridis. Clustering trajectories of moving objects in an uncertain world. In *ICDM*, pages 417–427, 2009.
[33] S. Ranu, D. P, A. Telang, P. Deshpande, and S. Raghavan. Indexing and matching trajectories under inconsistent sampling rates. *ICDE*, pages 999–1010, 2015.
[34] S. Ray, A. D. Brown, N. Koudas, R. Blanco, and A. K. Goel. Parallel in-memory trajectory-based spatiotemporal topological join. *Big Data*, pages 361–370, 2015.
[35] L. K. Sharma, O. P. Vyas, S. Scheider, and A. K. Akasapu. Nearest neighbour classification for trajectory data. In *ICT*, pages 180–185, 2010.
[36] 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.
[37] N. Ta, G. Li, T. Zhao, J. Feng, H. Ma, and Z. Gong. An efficient ride-sharing framework for maximizing shared route. *IEEE Trans. Knowl. Data Eng.*, 30(2):219–233, 2018.
[38] H. Tan, W. Luo, and L. M. Ni. Clost: a hadoop-based storage system for big spatio-temporal data analytics. In *CIKM*, pages 2139–2143, 2012.
[39] K. Toohey and M. Duckham. Trajectory similarity measures. *SIGSPATIAL Special*, 7:43–50, 2015.
[40] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.*, 40:175–179, 1991.
[41] M. Vlachos, D. Gunopulos, and G. Kollios. Discovering similar multidimensional trajectories. In *ICDE*, pages 673–684, 2002.
[42] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. J. Keogh. Indexing multidimensional time-series with support for multiple distance measures. In *KDD*, pages 216–225, 2003.
[43] H. Wang, H. Su, K. Zheng, S. W. Sadiq, and X. Zhou. An effectiveness study on trajectory similarity measures. In *ADC*, pages 13–22, 2013.
[44] H. Wang, K. Zheng, X. Zhou, and S. W. Sadiq. Sharkdb: An in-memory storage system for massive trajectory data. In *SIGMOD*, pages 1099–1104, 2015.
[45] X. Wang, H. Ding, G. Trajcevski, P. Scheuermann, and E. J. Keogh. Experimental comparison of representation methods and distance measures for time series data. *Data Mining and Knowledge Discovery*, 26:275–309, 2012.
[46] D. Xie, F. Li, and J. Phillips. Distributed trajectory similarity search. *PVLDB*, 10:1478–1489, 2017.
[47] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: efficient in-memory spatial analytics. In *SIGMOD*, pages 1071–1085, 2016.
[48] B.-K. Yi, H. V. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE*, pages 201–208, 1998.
[49] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, 1993.
[50] S. You, J. Zhang, and L. Gruenwald. Large-scale spatial join query processing in cloud. In *ICDE Workshops*, pages 34–41, 2015.
[51] J. Yu, J. Wu, and M. Sarwat. Geospark: A cluster computing framework for processing large-scale spatial data. In *SIGSPATIAL/GIS*, page 70, 2015.
[52] 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.
[53] Y. Zheng and X. Zhou. *Computing with spatial trajectories*. Springer Science & Business Media, 2011.

A OTHER DISTANCE FUNCTIONS

(1) Fréchet distance [4] .

Definition A.1. Given two trajectories $T = (t_1, \cdots, t_m)$ and $Q = (q_1, \cdots, q_n)$, Fréchet distance is computed as below

$$F(T,Q) = \begin{cases} \max_{i=1}^{m} \texttt{dist}(t_i, q_1) & \text{if } n = 1 \\ \max_{j=1}^{n} \texttt{dist}(t_1, q_j) & \text{if } m = 1 \\ max(\texttt{dist}(t_m, q_n), \min\Big(F(T^{m-1}, Q^{n-1}), & \\ \ F(T^{m-1}, Q), F(T, Q^{n-1})\Big)) & \text{otherwise} \end{cases}$$

where T^{m-1} is the prefix trajectory of T by removing the last point.

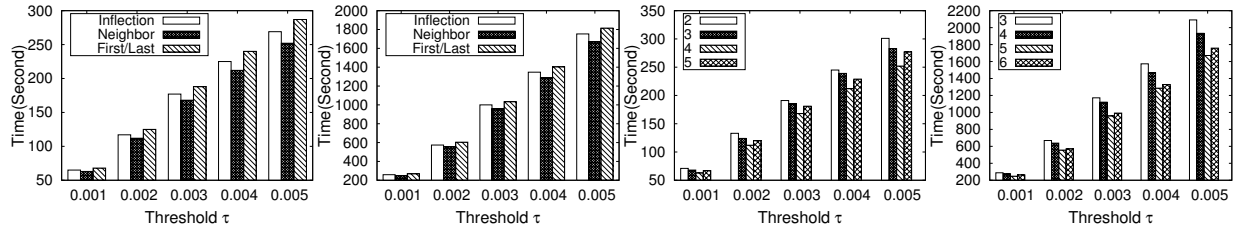
Given two trajectories T_1 and T_3 in Figure 1, $Fréchet(T_1, T_3) = 1.41$. To support Fréchet, DITA doesn't need to update τ by subtracting distance from it when querying the index. Similarly, we can still apply MBR coverage filtering, and cell-based estimation.

(2) Edit distance based function. It counts the minimum number of edits required to make two trajectories equivalent. We take edit distance on real sequence (EDR) [10] as an example.

Definition A.2 (EDR). Given two trajectories T and Q with lengths m and n, and a matching threshold $\epsilon \geq 0$, EDR_ϵ [39] is:

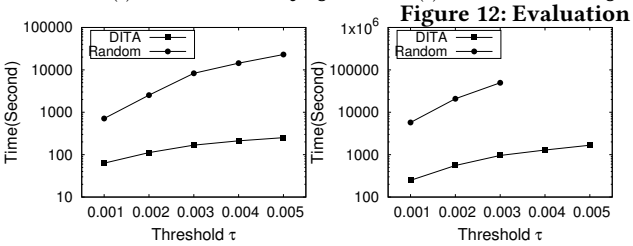
$$EDR_\epsilon(T,Q) = \begin{cases} n & \text{if } m = 0 \\ m & \text{if } n = 0 \\ \min\Big(EDR_\epsilon(T^{2,m}, Q^{2,n}) + subcost(t_1, q_1), & \\ EDR_\epsilon(T^{2,m}, Q) + 1, EDR_\epsilon(T, Q^{2,n}) + 1\Big) & \text{otherwise} \end{cases}$$

where $T^{2,m}$ stands for trajectory T with its first point removed, and $subcost(t,q) = 0$ if $\texttt{dist}(t,q) \leq \epsilon$; 1 otherwise.



(a) Pivot Selection: Beijing (b) Pivot Selection: Chengdu (c) Pivot Size: Beijing (d) Pivot Size: Chengdu

Figure 12: Evaluation on Pivot Selection Methods(DTW)



(a) Partitioning Scheme: Beijing (b) Partitioning Scheme: Chengdu

Figure 13: Evaluation on Partitioning Schemes(DTW)

Given two trajectories T_1 and T_3 in Figure 1, let $\epsilon = 1$, we have $EDR_\epsilon(T_1, T_3) = 2$. To support EDR, for each MBR in the trie, we compute the distance. If it is beyond ϵ, we subtract threshold by 1. If current threshold is below zero, we prune this trie node. Secondly, we employ other filtering techniques in the leaf node. For example, we build a global grid map, map each point to a grid on the map, and use an inverted list whose key is the grid and value are trajectories in the leaf node that contain points in this grid. For each point $q \in Q$, we find all grids that are within distance of ϵ from q, and use the inverted list to find trajectories that contain points in those grids as candidates. Thirdly, we utilize length filtering: given two trajectories T and Q of length m and n, we have $EDR_\epsilon(T, Q) \geq |m - n|$. If $|m - n| > \tau$, T and Q cannot be similar.

(3) Longest common subsequence distance (LCSS). It measures the maximum number of equivalent points with traversing the two trajectories monotonically from start to end [39].

Definition A.3 (LCSS). Given two trajectories T and Q with lengths m and n, an integer $\delta \geq 0$ and a matching threshold $\epsilon \geq 0$, the $LCSS_{\delta,\epsilon}$ is defined as below [41]:

$$LCSS_{\delta,\epsilon}(T,Q) = \begin{cases} n & \text{if } m = 0 \\ m & \text{if } n = 0 \\ LCSS_{\delta,\epsilon}(T^{m-1}, Q^{n-1}) & \text{if } |m-n| \leq \delta \ \& \\ 1 + min(LCSS_{\delta,\epsilon}(T^{m-1}, Q), & \texttt{dist}(t_m, q_n) \leq \epsilon \\ LCSS_{\delta,\epsilon}(T, Q^{n-1})) & \text{otherwise} \end{cases}$$

where T^{m-1} is the prefix trajectory of T with the last point removed.

Given two trajectories T_1 and T_3 in Figure 1, let $\delta = 1$ and $\epsilon = 1$, we have $LCSS_{\delta,\epsilon}(T_1, T_3) = 2$. To support LCSS, we adopt the above-mentioned three modifications for EDR since they still work for LCSS. One thing to note here that for the first modification, when we partition the points into MBRs, we partition by both its coordinates and its index in the trajectory. For each MBR, we compute the minimum distance between it and the part of the query trajectory which fulfills the index constraint (as $|m - n|$ in Definition A.3), if it is beyond ϵ, we would subtract current threshold by 1.

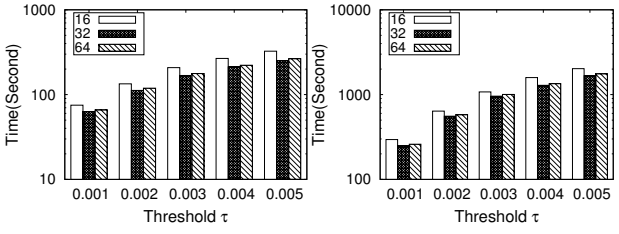
Table 4: Varying # of Partitions with DTW Distance

N_G	Time (Beijing)		Time(Chengdu)	
	Search (ms)	Join (s)	Search (ms)	Join (s)
32	4	604	10	5943
64	2	252	6	2834
128	12	334	13	1671
256	35	697	25	3035

B ADDITIONAL EXPERIMENTS

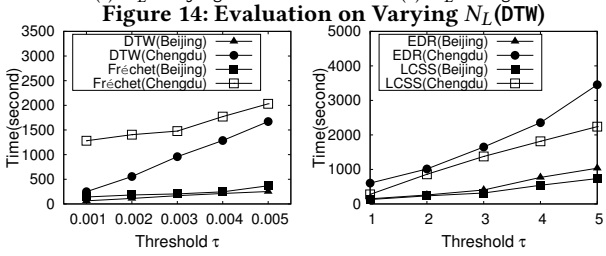
Varying # of Partitions. As discussed in Section 4, the total number of partitions was $N_G \times N_G$. Table 4 showed the performance of our method by varying N_G, and we can observe that both the performance of search and join first increased and then decreased with increasing N_G. This was because that although increasing the number of partitions improved parallelism, it required transmitting more data (i.e., candidate trajectories) between workers and introduced more overhead (e.g., querying the local index). We could obtain a best parameter value $N_G = 64$ for search. Further, by comparing between search and join on the same dataset, we found that the optimal number of partitions for join was a little larger than that of search, since the join computation typically consisted of larger workloads, and more number of partitions would allow for more parallelism and help balancing the loads between workers. Therefore, we believed that the expected workload of a particular partition was supposed to be medium, not too large for incurring un-balancing loads and not too small for introducing much overhead. Moreover, for join, the optimal number of partitions on Chengdu was larger than the optimal number of partitions on Beijing, because Chengdu had much more candidates and answers than Beijing (even more than one order of magnitude). As mentioned above, increasing the number of partitions on Chengdu would help keep the expected workload at a medium level to improve the overall performance. Based on these observations, we could utilize a binary search to find the best number of partitions when building indexes on a new dataset.

Pivot Selection Strategy. We evaluated our three pivot selection strategies. Figures 12(a) and 12(b) showed the performance of DITA with different pivot selection strategies by varying threshold. Neighbor achieved the best performance among all three strategies, and First/Last was the worst. For example, when $\tau = 0.005$ on Beijing, Neighbor took 252 seconds, Inflection took 269 seconds, and First/Last took 287 seconds. This was because: (1) First/Last found points with maximum distance from first/last points, which could have bad performance when there are many points closed to each other but far from first/last points; (2) Inflection took direction into account but it neglected points in the same direction with long distances; (3) Neighbor tried to find pivot points



(a) N_L: Beijing (b) N_L: Chengdu

Figure 14: Evaluation on Varying N_L (DTW)



(a) Varying τ: DTW and Fréchet (b) Varying τ: EDR and LCSS

Figure 15: Other Distance Functions

as far as possible between each other, thus forming a representative set of points for a particular trajectory.

Varying Pivot Size. Figures 12(c) and 12(d) showed the performance of DITA with different pivot size K by varying the threshold. We had the following observations. Firstly, $K = 4$ achieved the best performance on Beijing. This was because that: (1) $K = 3$ could not filter dissimilar pairs efficiently compared with others since it utilized the least number of points; (2) $K = 5$ took more time querying the index but did not improve the pruning power greatly; (3) Since it took much more time for verification than indexing, $K = 5$ was better than $K = 3$. Secondly, $K = 4$ achieved the best performance on Beijing, while $K = 5$ achieved the best performance on Chengdu, because trajectories in Chengdu had longer length than those in Beijing on average, thus more pivot points would increase performance along with increasing average length. Thus, the selection of pivot size was obviously a trade-off between querying index time and pruning power which should be fine-tuned.

Partitioning Scheme. Figures 13(a) and 13(b) compared the performance for similarity join of two different partitioning schemes: Random (random partitioning for all trajectories) and DITA (our method). We could find that DITA outperformed Random even by several orders of magnitude, which lied in two folds: (1) DITA partitioned trajectories according to their first and last points, which made similar trajectories tend to reside together, thus for each trajectory, DITA only sent it to a few relevant partitions while Random sent it to all partitions. Therefore, the global transmission cost was greatly reduced by adopting DITA. (2) Locally, since similar trajectories tended to reside together, and their points would also tend to be well clustered, the clustered MBRs will be more compact (or smaller) than those in random-partitioned partitions, making it more efficient to find candidates for DITA than Random. Therefore, the local computation cost was greatly reduced by employing DITA.

Varying N_L. Figures 14(a) and 14(b) showed the performance of DITA with different N_L by varying threshold. We found that $N_L = 32$ achieved the best performance, while $N_L = 16$ was the worst. For example, when $\tau = 0.005$ on Chengdu, $N_L = 32$ took 1671 seconds, $N_L = 64$ took 1760 seconds, and $N_L = 16$ took 2022 seconds. This was because that: (1) $N_L = 16$ did not separate points well and clustered MBRs were a little big, making it not efficient enough for filtering dissimilar pairs; (2) $N_L = 64$ created too many child trie nodes, and it took a lot of time to access the index but did not improve the pruning power greatly; (3) Since it took much more time for verification than indexing, $N_L = 64$ was better than $N_L = 16$. Thus, the selection of N_L was obviously a trade-off between querying index time and pruning power which should be fine-tuned.

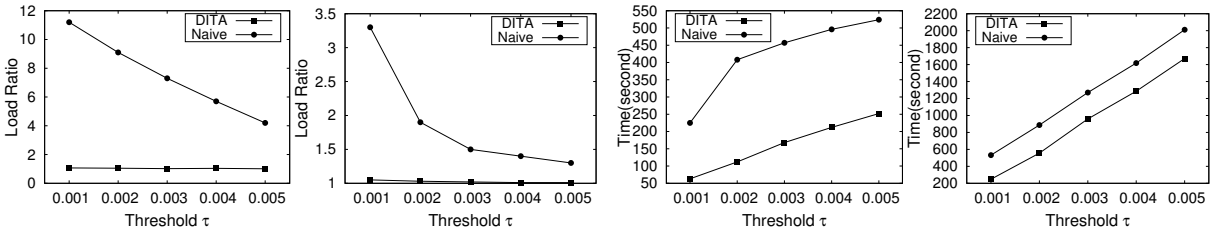
Table 5: Indexing Time (Seconds) and Size (MB)

Methods	Sample Rate	Time	Global Size	Local Size
DITA (Beijing)	0.25	43	14	307
	0.5	94	14	659
	0.75	150	14	1014
	1.0	197	14	1446
DITA (Chengdu)	0.25	97	65	820
	0.5	189	65	1395
	0.75	270	65	1852
	1.0	345	65	2224
DFT (Beijing)	1.0	108	48	12843
DFT (Chengdu)	1.0	265	78	34251

Indexing Size and Time. Table 5 varied dataset size and showed the time for building indexes and indexes' space usage, where dataset size 0.5 meant that we sampled 50% data to conduct the experiments. We made the following observations. Firstly, it took not much time to build indexes (within 6 minutes). Secondly, the indexing time increased nearly linearly with dataset size, because when there were too few trajectories (by default 16) in a trie node, DITA would stop creating child trie nodes for this node. Thus, with more data given, there would be more trie nodes proportional to the number of trajectories, and the time for building trie indexes would also increase in the same way. Thirdly, the size of the global index was less than 100MB for both Beijing and Chengdu (14MB and 65MB respectively), making it accessible for replicating to every worker. The size of global index did not change with varying dataset sizes because it was decided by the number of partitions. Moreover, the size of the local indexes increased linearly with dataset size, because we need to index more trajectories in the local index. Further, Chengdu had larger local indexes size than Beijing as Chengdu had more partitions and more trajectories. Finally, although the indexing time for DITA was a little longer than that of DFT, DITA had a smaller global index size and much smaller (even by one order of magnitude) local index size than DFT.

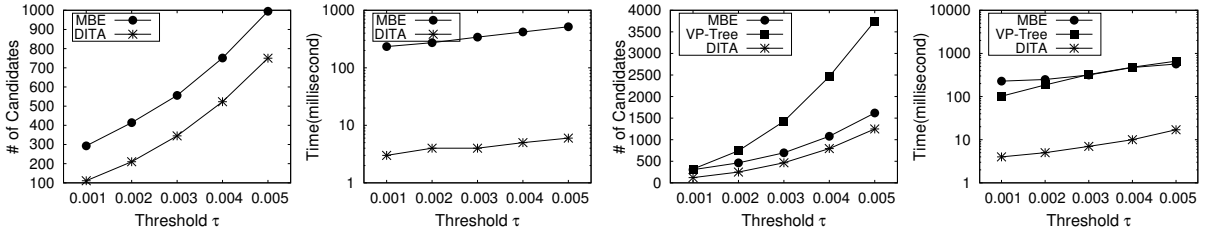
Supporting other distance functions. We evaluated DITA's performance with different distance functions, including DTW, Fréchet, EDR and LCSS ($\epsilon = 0.0001, \delta = 3$), in Figure 15. We could observe that: (1) Fréchet was slower than DTW with the same threshold, because DTW sums the values up through the whole path from $(1, 1)$ to (m, n) while Fréchet chooses the maximum value; (2) LCSS was faster than EDR with the same threshold, because LCSS has an index constraint while EDR does not.

Evaluation on Load Balancing. To justify that our balancing mechanism in fact works, we collected total execution time for each worker, and calculated the ratio of the longest to the shortest as the un-balanced ratio. We compared two methods, Naive without any balancing mechanism, and DITA with our balancing mechanism on the un-balanced ratio and total execution time for join. The results were shown in Figure 16. We made the following observations: (1) the ratios for two methods decreased w.r.t. an increasing



(a) Load Ratio: Beijing (b) Load Ratio: Chengdu (c) Total Time: Beijing (d) Total Time: Chengdu

Figure 16: Evaluation on Load Balancing (DTW)



(a) # of Candidates (DTW) (b) Time (DTW) (c) # of Candidates (Fréchet) (d) Time (Fréchet)

Figure 17: Comparison with Centralized Baselines (Search)

Table 6: Datasets (Centralized)

Datasets	Cardinality	AvgLen	MinLen	MaxLen	Size
Chengdu(tiny)	1000,000	38.5	6	205	1.88GB

Table 7: Indexing Time and Size (Centralized)

Methods	Index Time (seconds)	Index Size (MB)
DITA	57	219
MBE	834	1257
VP-Tree	3507	3021

threshold, because that although a larger threshold increased the largest workload among all partitions, it also made the number of partitions with "large" workloads much more and these partitions would probably be executed on different workers, thus the un-balancing issue was kind of mitigated; (2) the un-balancing issue was more severe on Beijing than Chengdu, it was mainly because that Chengdu had more partitions and more answers thus making the number of partitions with "large" workloads much more; (3) our balancing mechanism worked well with little overhead.

C COMPARISON WITH CENTRALIZED BASELINES

Centralized Baseline Methods. We made a centralized implementation of DITA and compared with two existing centralized methods: VP-Tree (vantage point tree) [19], and the Minimal Bounding Envelop MBE [42]. VP-Tree utilized the triangle inequality property and thus it could only support Fréchet which was a metric distance function. MBE can support both DTW and Fréchet. We compared the number of candidates survived after pruning as well as the total runtime.

Dataset. We compared all methods on a single machine in our cluster using a small dataset sampled from Chengdu (Chengdu(tiny), see Table 6), because VP-Tree and MBE failed to build index in 12 hours on the whole Chengdu dataset.

Indexing. As shown in Table 7, DITA built the index with much shorter time and smaller space usage. As shown below, our method achieved a higher pruning power and higher query performance.

Experiment Results for Trajectory Search. We randomly sampled 1000 queries from Chengdu(tiny) and compared the number of candidates per query and the average query latency of each method when varying the threshold on the Chengdu(tiny) dataset. Figure 17 showed the results. We made the following observations. (1) With the increase of threshold, the three methods took more time and generated more candidates, because a larger threshold led to more answers. (2) In term of the number of candidates, DITA was better than VP-Tree and MBE, since (i) DITA designed a more efficient indexing scheme with accumulating distance computation than VP-Tree and MBE, and (ii) our proposed cell-based pruning had a more accurate bound than MBE using a multi-level trie index which accumulated distance level by level, and to our best knowledge, there has not been such an indexing scheme that can support such an accumulation of distance, and (iii) we employed the ordering constraint in distance functions (e.g., DTW, Fréchet) to further improve the performance. Moreover, we partitioned pivot points into several MBRs and indexed them, while most existing methods indexed MBRs of trajectories. In other words, we employed a finer-grained indexing scheme than previous studies. (3) In term of running time, DITA significantly outperformed the two baselines, even by one order of magnitude, because DITA incurred smaller overhead in index probing with higher pruning power, and employed an efficient verification process to further boost the performance. (4) The performance gap between DITA and the two baselines was much larger on DTW than Fréchet, because DITA's indexing scheme could accumulate the distance level by level for DTW.

Experiment Results for Trajectory Join. We also compared our method with centralized approaches for trajectory joins. However, VP-Tree and MBE were extremely slow in processing the trajectory join operation even on this small Chengdu(tiny) dataset, because trajectory join was more expensive than trajectory search. DITA outperformed them by around 40 times in efficiency, due to our effective index and pruning techniques. For example, under the choice of the Fréchet distance, when $\tau = 0.005$, DITA took 4.7 hours to join on Chengdu(tiny), while MBE took 157 hours and VP-Tree took 183 hours respectively.