DITA: A Distributed In-Memory Trajectory Analytics System

Zeyuan Shang
Brown University
zeyuan_shang@brown.edu

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. In this paper, we demonstrate a distributed in-memory trajectory analytics system DITA to support large-scale trajectory data analytics. DITA exhibit three unique features. First, DITA supports threshold-based and KNN-based trajectory similarity search and join operations, as well as range queries (i.e., space and time). Second, DITA is versatile to support most existing similarity functions to cater for different analytic purposes and scenarios. Last, DITA is seamlessly integrated into Spark SQL to support easy-to-use SQL and DataFrame API interfaces. Technically, DITA proposes an effective partitioning method, global index and local index, to address the data locality problem. It also devises cost-based techniques to balance the workload, and develops a filter-verification framework for efficient and scalable search and join.

ACM Reference Format:
Zeyuan Shang, Guoliang Li, and Zhifeng Bao. 2018. DITA: A Distributed In-Memory Trajectory Analytics System. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183713.3193553

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 studies focus on optimizing the analytics techniques in a single machine [3–8, 11]. 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 of (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*[12], *LCSS*[11] and *EDR*[6], and the metric ones like Fréchet [2]. We observe that existing studies either support one or two of them, or define its own one, while it is critical to support all these similarity functions in one system for different analytics purposes and/or 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, design a global index to find relevant data partitions that contain possible answers, and a local index to compute answers in each partition. We propose effective filter-verification algorithms to compute the answers: 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 mechanisms to prevent from stragglers. All techniques can support most existing trajectory similarity functions aforementioned.

As a distributed in-memory trajectory analytics systems, DITA can be employed in a variety of real-world applications:

Trajectory Similarity Queries. Given two sets of trajectories (or a single query trajectory and a set of trajectories), DITA can be used to identify the pairs of trajectories whose similarity is either beyond some threshold (i.e., threshold-based) or is among top K (i.e., KNN based) with SQL syntax or DataFrame API for interactive analytics.
Trajectory Range Queries. Given a set of trajectories and a range (e.g., a rectangle or circle) constraint on space or time DITA can be used to find trajectories meeting the constraint efficiently.

Moreover, DITA can visualize the results on an map interactively to better interpret the results. Our source code is publicized at https://github.com/TsinghuaDatabaseGroup/DITA. For more technical details, please refer to our full research paper [10].

[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/10.1145/3183713.3193553

2 OVERVIEW

2.1 Definitions

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.

Supported Trajectory Similarity Functions. We support Dynamic Time Warping (DTW) [9], Fréchet distance [2], edit distance on real sequence (EDR) [6] and longest common subsequence distance (LCSS) [11] for measuring the distance between trajectories.

Trajectory Range Function. We define that a trajectory T is *in* a spatio-temporal object (e.g., a rectangle, a circle) if all of its points are in the corresponding object.

2.2 Supported Query Operations

Our system can support the following six core query operations on trajectories.

Definition 2.2 (Threshold-based Trajectory Similarity Search). Given a query trajectory Q, a collection of trajectories $\mathcal{T}$, a trajectory similarity function f (e.g., DTW) and a threshold τ, the threshold-based trajectory similarity search operator finds all trajectories $T \in \mathcal{T}$, such that $f(T, Q) \leq \tau$.

Definition 2.3 (KNN-based Trajectory Similarity Search). Given a query trajectory Q, a collection of trajectories $\mathcal{T}$, a trajectory similarity function f (e.g., DTW) and an integer k, the knn-based trajectory similarity search operator finds a set of k trajectories $\mathcal{K} \subset \mathcal{T}$, such that for each $K \in \mathcal{K}$ and $T \in \mathcal{T} - \mathcal{K}$, $f(K, Q) \leq f(T, Q)$.

Definition 2.4 (Threshold-based Trajectory Similarity Join). Given two collections of trajectories $\mathcal{T}$ and $\mathcal{Q}$, a similarity function f (e.g., DTW) and a threshold τ, the threshold-based trajectory similarity join operator finds all similar pairs $(T, Q) \in \mathcal{T} \times \mathcal{Q}$, such that $f(T, Q) \leq \tau$.

Definition 2.5 (KNN-based Trajectory Similarity Join). Given two collections of trajectories $\mathcal{T}$ and $\mathcal{Q}$, a trajectory similarity function f (e.g., DTW) and an integer k, the knn-based trajectory similarity join operator finds a set of k similar pairs $\mathcal{P}$, such that for each $(T, Q) \in \mathcal{P}$ and $(T', Q') \in \mathcal{T} \times \mathcal{Q} - \mathcal{P}$, $f(T, Q) \leq f(T', Q')$.

Definition 2.6 (Trajectory Range Search). Given a query spatio-temporal object Q, and a collection of trajectories $\mathcal{T}$, the trajectory range search operator finds all trajectories $T \in \mathcal{T}$, such that T is *in* Q.

2.3 System Architecture of DITA

Next, we briefly describe the architecture of DITA (Figure 1).

Extended SQL. We extend Spark SQL to support the core query operations described in Section 2.2.

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

```
SELECT * FROM T WHERE f(T, Q) ≤ τ
SELECT * FROM T WHERE f(T, Q) KNN k
```

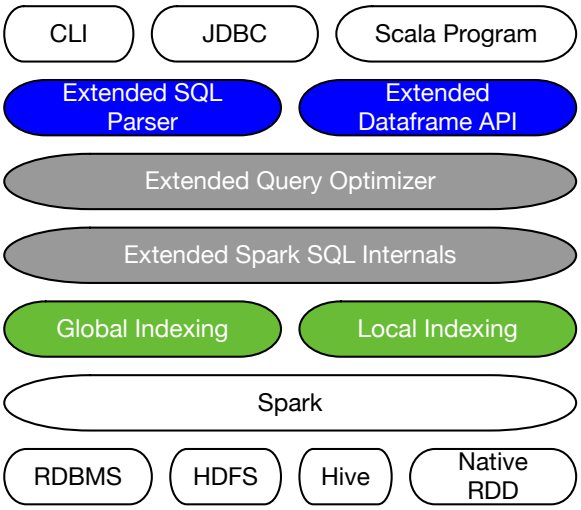

Figure 1: DITA Architecture

(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 similarity function f and a threshold τ.

```
SELECT * FROM T JOIN Q ON f(T, Q) ≤ τ
SELECT * FROM T JOIN Q ON f(T, Q) KNN k
```

(3) Trajectory Range Queries. Users can utilize the following query to find trajectories in table $\mathcal{T}$ that are within the range w.r.t. a rectangle with the bottom-left point (x_1, y_1) and top-right point (x_2, y_2) or a circle with the center (x, y) and radius r. Note that DITA can support multi-dimensional points.

```
SELECT * FROM T WHERE T IN MBRRANGE (POINT(x1, y1), POINT(x2, y2))
SELECT * FROM T WHERE T IN CIRCLERANGE (POINT(x, y), r)
```

DataFrame. In addition to the extended SQL syntax, 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 queries and range queries.

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 elaborated in Section 3.1.

```
CREATE INDEX TrieIndex ON T USE TRIE.
```

Query Processing. Given a SQL query or DataFrame API request, DITA transforms it into a logical plan, and then optimizes it with rule-based optimizations (e.g., predicate pushdown, constant folding). Afterwards, DITA 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 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 3.3.

Interactivity. We integrate our system with Spark SQL and Apache Zeppelin[1] to provide interactive analytics. We further implement a user-friendly interface which visualizes the results on the map.

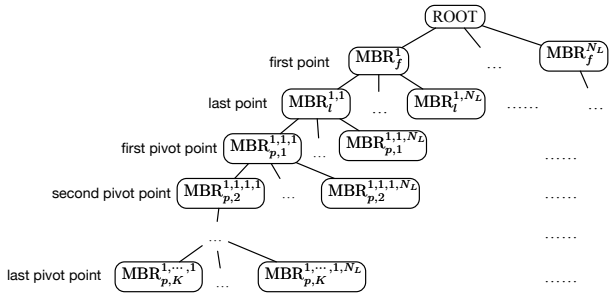

Figure 2: Trie Index

3 IMPLEMENTATION OF THE DITA SYSTEM

3.1 Indexing

3.1.1 Basic Idea. To facilitate illustrating our idea, we will use the DTW similarity function as an instance by default. We first select several pivot points from a trajectory T to approximately represent it, and then the 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 these pivot points, which supports accumulating the distance level by level to improve the pruning power. Details can be found in our full paper [10].

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. We have several strategies in calculating the weight, e.g., we use the distance between neighbor points or the distance between first/last points as the weights.

Trie Index. We use the first, last and pivot points of trajectories to build the trie index, as shown in Figure 2. We first group all trajectories by their first points into N_G disjoint buckets, then we further group the trajectories in each bucket by their last points into N_G sub-buckets. Next, we repeat this process using the first pivot point, then the second one, until the last pivot point. For each bucket, we calculate the minimum bounding rectangle (MBR). When querying the index, we calculate the distance between MBR and the trajectory and use it as the lower bound of the distance between trajectories. If it is beyond the threshold τ, we could safely prune this trie node; otherwise we update the threshold by subtracting the distance from it, which could improve our pruning power. In other words, we query the index by accumulating the distance level by level in order to enable early termination.

Distributed Indexing. When building an index on a huge collection of trajectories, we first divide them into multiple partitions on different machines for distributed computation. Then DITA employs two levels of indexes: (1) the global index that finds relevant partitions that may contain trajectories similar to Q; (2) the local index that finds candidate trajectories in each relevant partition locally. We use the trie index scheme in all steps (including partitioning, global indexing and local indexing).

3.2 Search Query Implementations

Threshold-based Similarity Search. The search queries are processed in three steps: (1) the master (called the driver in Spark) uses the global index to locate 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.

KNN-based Similarity Search. First, we need to estimate a similarity threshold τ which guarantees to have at least K trajectories whose distance to Q is less than or equal to τ. Besides, we would like to make τ as small as possible to reduce computation. To achieve this, we replicate the query trajectory Q to all partitions. In each partition, we query the local index to get the number of candidates, and we employ a binary search to find the minimum threshold such that there are more than K candidates. We then compute the similarity between these candidates and the query trajectory, and aggregate them to get the least K distance as the estimated threshold τ_{min}. Next, we run the threshold-based similarity search with the threshold τ_{min} and get the K trajectories with least distances.

Range Search. Similar to threshold-based similarity search, we use the global and local index to find the results in the spatio-temporal object (e.g, a circle or a rectangle).

Table 1: Dataset

Datasets	Cardinality	AvgLen	MinLen	MaxLen
Beijing	293,536	30.07	10	72

3.3 Join Query Implementations

Threshold-based Similarity Join. Although join can be implemented based on our search method, we further devise cost-based optimizations to better coordinate the distributed join. The idea is to measure the computation cost and transmission cost by joining on a sampled dataset, and use these costs to balance the workloads.

KNN-based Similarity Join. Similar to KNN-based similarity search, we need to estimate the minimum threshold first. In each partition, we use binary search again to find the minimum threshold such that there are more than K candidate pairs, then we aggregate them to get the least K distances as the estimated threshold τ_{min}. Next, we run the threshold-based similarity join with the threshold τ_{min} to get the K trajectory pairs with the least distances.

4 DEMONSTRATIONS

We demonstrate three use cases of DITA, 1) trajectory similarity search queries, 2) trajectory similarity join queries, and 3) trajectory range queries. We integrate our system with Spark SQL and Zeppelin, and use MapBox[2] to visualize the results. We use the Beijing dataset, which was collected from the GPS device on taxis in Beijing[3]. Table 1 shows some statistics of this dataset.

Similarity-based Trajectory Recommendation. Trajectory recommendation is useful in analyzing large volumes of trajectories and find potentially interesting ones. Given a trajectory T, one possible way is to employ the similarity-based trajectory search to find trajectories similar to T as candidates for recommendation. Regarding similarity functions, DITA supports the mostly widely adopted ones such as DTW, Fréchet, EDR and LCSS, to fit different analytical scenarios and purposes. For example, if we want to find similar trajectories of a bus route around Tsinghua University, we can simply issue a threshold-based similarity search SQL query

[2]https://www.mapbox.com/
[3]http://more.datatang.com/en

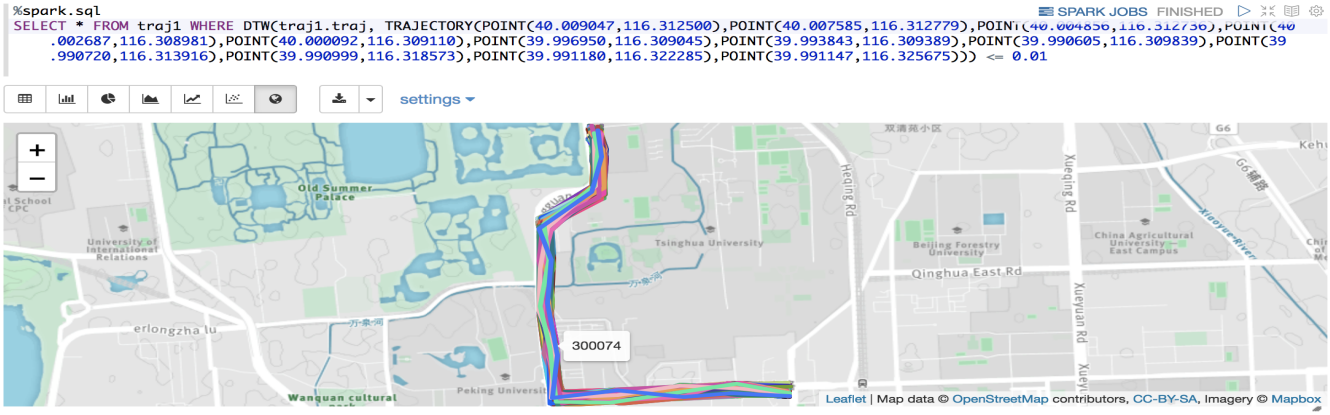

Figure 3: Similarity-based Trajectory Recommendation

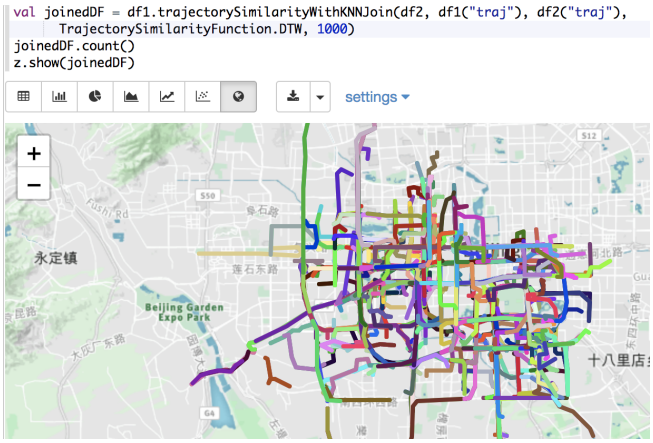

Figure 4: Trajectory Clustering

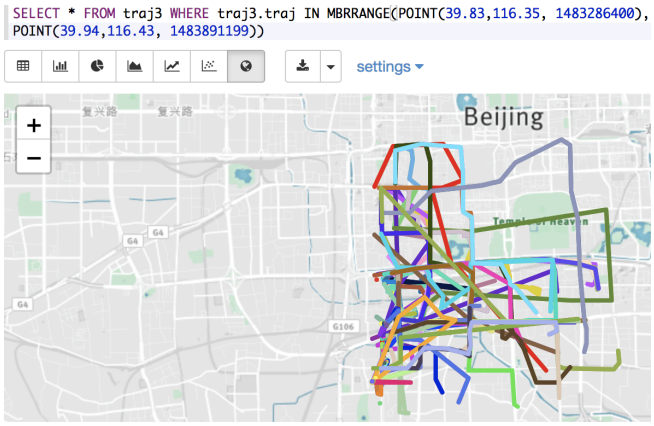

Figure 5: Range-based Trajectory Query

using DTW as the similarity function (at the top of Figure 3) and DITA would compute the results and visualize them on the map. As shown in Figure 3, these similar trajectories follow roughly the same path with only minor differences, and detailed statistics will be displayed when the user hovers the pointer over the trajectories.

Trajectory Clustering. Trajectory clustering aims to group a set of objects into classes of *similar* ones, which can be achieved by employing trajectory similarity joins. DITA implements both threshold-based and KNN-based similarity joins. For example, given two sets of trajectories in Beijing (i.e., *df1* and *df2* in Figure 4, which are two equal-sized halves of trajectories in Beijing), we may use DataFrame API to find the top 1,000 similar trajectory pairs. Similar trajectory pairs are drawn using similar colors. From Figure 4, we can observe that most similar trajectories locate in the south-east part or the center part of Beijing. This can be used in subway station planning and road planing.

Range-based Trajectory Query. The range-based trajectory query is able to find the trajectories that pass a given spatial range within a specified time range. It is especially useful in traffic flow analysis, congestion reasoning, route planning, etc. As shown in Figure 5, we use a SQL query to find trajectories in a specific spatio-temporal range, i.e., within the Southern Second Ring of Beijing in one week of Year 2017 (1483286400 and 1483891199 are the Unix timestamps). As a result, the envelop of the returned trajectories forms a rectangle, which is our query rectangle at the top of Figure 5.

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. Guoliang Li is the corresponding author.

REFERENCES

[1] Apache zeppelin. http://zeppelin.apache.org/.
[2] H. Alt and M. Godau. Computing the fréchet distance between two polygonal curves. *Int. J. Comput. Geometry Appl.*, 5:75–91, 1995.
[3] 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.
[4] P. Bakalov, M. Hadjieleftheriou, and V. J. Tsotras. Time relaxed spatiotemporal trajectory joins. In *GIS*, pages 182–191, 2005.
[5] L. Chen and R. T. Ng. On the marriage of lp-norms and edit distance. In *VLDB*, pages 792–803, 2004.
[6] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD*, pages 491–502, 2005.
[7] H. Ding, G. Trajcevski, and P. Scheuermann. Efficient similarity join of large sets of moving object trajectories. In *TIME*, pages 79–87, 2008.
[8] E. Frentzos, K. Gratsias, and Y. Theodoridis. Index-based most similar trajectory search. In *ICDE*, pages 816–825, 2007.
[9] 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.
[10] Z. Shang, G. Li, and Z. Bao. Dita: Distributed in-memory trajectory analytics. In *SIGMOD*, 2018.
[11] M. Vlachos, D. Gunopulos, and G. Kollios. Discovering similar multidimensional trajectories. In *ICDE*, pages 673–684, 2002.
[12] B.-K. Yi, H. V. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE*, pages 201–208, 1998.