Towards In-Memory Sub-Trajectory Similarity Search

Omid Isfahani Alamdari
University of Pisa
alamdari@di.unipi.it

Mirco Nanni
ISTI-CNR Pisa, Italy
mirco.nanni@isti.cnr.it

Roberto Trasarti
ISTI-CNR Pisa, Italy
roberto.trasarti@isti.cnr.it

Dino Pedreschi
University of Pisa
dino.pedreschi@di.unipi.it

ABSTRACT

Spatial-temporal trajectory data contains rich information about moving objects and has been widely used for a large number of real-world applications. However, the complexity of spatial-temporal trajectory data, on the one hand, and the fast collection of datasets, on the other hand, has made it challenging to efficiently store, process, and query such data. In this paper, we propose a scalable method to analyze the sub-trajectory similarity search in an in-memory cluster computing environment. Notably, we have extended Apache Spark with efficient trajectory indexing, partitioning, and querying functionalities to support the sub-trajectory similarity query. Our experiments on a real trajectory dataset have shown the efficiency and effectiveness of the proposed method.

1 INTRODUCTION

Following the ubiquity of location-aware devices, collecting the location data of moving objects is now much easier thanks to the developments in positioning systems and mobile communications. Many applications in the areas of transportation and urban planning can benefit from this data to improve their quality of service. Moreover, the location-based services (LBS) market that primarily depends on the data gathered by location-aware devices is rapidly expanding. Analyzing the historical trajectories of moving objects plays an essential role in the success of applications, such as carpooling and route recommendation systems. However, the extensive use of LBS applications imposes significant storage and processing challenges to application servers. Every day several gigabytes or even terabytes of moving objects tracks are accumulated in servers, making their post-processing an arduous task.

One of the critical operations in trajectory data analytics is similarity search where given a query trajectory Q and a distance threshold δ, it returns all trajectories with distance at most δ to Q, according to a distance function. We call this problem the whole trajectory similarity search problem, in which trajectories are matched as a whole. In other words, for every comparison between a query and a candidate trajectory, the first and last points of both trajectories are matched, and depending on the type of the distance function, either matching or coupling of the in-between points is established.

However, as illustrated in Figure 1, there are many situations where start and end points of trajectories are far from each other while they have similar parts in-between. In this figure, the start and end points of trajectory T_1 (blue solid line) are A and B, and for trajectory T_2 (red dashed line) they are C and D. If the distance between A and C is more than δ, the two trajectories are not considered similar, although they have very similar routes. In this paper, we address the problem of *sub-trajectory similarity search*, where we aim to find similar sub-trajectories in the dataset to the whole or a sub-trajectory of the query trajectory. An interesting application of this query is carpooling, where a user can share a ride with other users who have similar routes. For the example shown in Figure 1, the carpooling system can match trajectories T1 and T2 and hence, suggest the user who follows T1 to share the ride with the user who follows T2, with possibly a small deviation from his/her routine path.

The two state-of-the-art works [6] and [4] find similarities between whole trajectories and are unable to find similarities in the sub-trajectory level. Different from those works, our method targets the more computationally expensive problem of sub-trajectory similarity search and tries to find not only the similarities in whole trajectory curves but also in between every sub-trajectory of data and query trajectories. Furthermore, our work is different from [5] in the sense that we focus on in-memory analytics to enhance the execution of interactive queries and at the same time being able to use the sub-trajectory similarity search as a primitive for more advanced trajectory pattern mining algorithms that try to identify objects that move together closely. In this regard, our goal is to use less memory space and accelerate the local computations inside partitions.

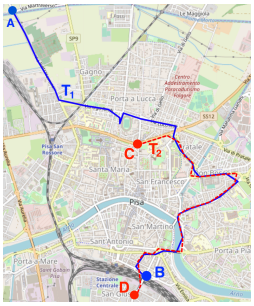

Figure 1: Two trajectories with similar routes

Parallel processing of trajectory data is inevitable while dealing with massive amounts of trajectory data. The main goal of this paper is to introduce a novel approach to the sub-trajectory similarity search problem in an in-memory cluster computing environment, i.e., Apache Spark.

2 PROBLEM FORMULATION

In this section, we formally define trajectories, sub-trajectories, and the sub-trajectory similarity search problem.

Definition 2.1. A trajectory T is a sequence of time-stamped sample points $\langle p_1, \ldots, p_{|T|} \rangle$ where each p_i is a triplet (x_i, y_i, t_i)

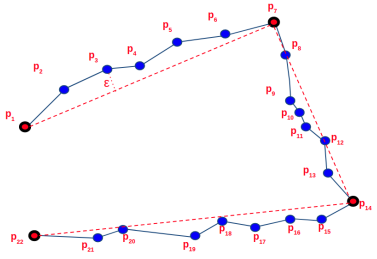

Figure 2: A trajectory and its simplifications

that indicates the spatial location (x_i, y_i) of the moving object at timestamp t_i.

Even though the movement is a continuous phenomenon, a trajectory is recorded by a finite set of discrete sample points. Let $T[i]$ be the i^{th} sample point of T. We consider that the move in-between two consecutive points $T[i]$ and $T[i+1]$ is on a straight line with constant speed and hence, the expected location of the trajectory T between $T[i]$ and $T[i+1]$ is obtained by a linear interpolation. We use $T(t)$ to denote the position (x, y) of the object at timestamp $t \in [t_1, t_{|T|}]$. Furthermore, we represent the number of points in a trajectory T with $|T|$ and its spatial length as $SL(T) = \sum_{i=1}^{|T|-1} d(p_i, p_{i+1})$ where d is the Euclidean distance.

Definition 2.2. A **sub-trajectory** $T[i:j]$ is a contiguous subsequence of T starting at point $T[i]$ and ending at point $T[j]$, where $1 \leq i \leq j \leq |T|$.

Definition 2.3. An **approximation line segment** for trajectory (or sub-trajectory) T, denoted by ALS^T, is the line segment $\overline{p_1 p_{|T|}}$ connecting the first and last point of T.

Definition 2.4. Given a query trajectory Q, a set of trajectories $\mathcal{D} = \{T_1, \ldots, T_N\}$, a distance function $\mathcal{F}$, a distance threshold $\delta > 0$ and a minimum length threshold $\lambda > 0$, the sub-trajectory similarity search returns the set of trajectories $\mathcal{R} = \{R_1, \ldots, R_M\}$, s.t. for every $R_j \in R$, the following holds:

(1) There exists a $T_i \in \mathcal{D}$ s.t. R_j is a sub-trajectory of T_i,
(2) There exists a sub-trajectory $Q[a:b]$ of Q such that $\mathcal{F}(R_j, Q[a:b]) \leq \delta$,
(3) $SL(R_j) \geq \lambda$

3 SYSTEM OVERVIEW

In this section, we describe our proposed method for distributed processing of the sub-trajectory similarity search. After reading the trajectory dataset, three steps of sampling, partitioning, and indexing are executed. In the sampling phase, a uniform random sample of the trajectory points is drawn from the input points. The idea is to acquire knowledge about the distribution of data in the geographical extent of the input dataset. This sample of points is used to build the *partitioner* object, which provides information about the computed boundaries of partitions and methods for locating objects in the partitions. Then, the physical partitioning of trajectory objects is performed using the partitioner, and trajectory objects are created inside partitions. In the final step, we build a 2-level distributed index over the trajectory data. This index will be used by the query processor to discard the irrelevant data to the query, both at the global and local levels. Finally, the query processor verifies the similarity between the candidate sub-trajectories that were not discarded and the query. The structure of the distributed index is provided in Figure 3.

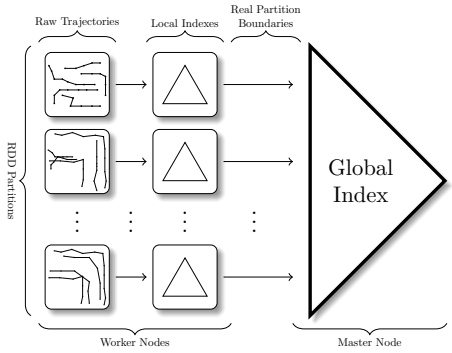

Figure 3: The distributed indexing structure.

4 DISTRIBUTED TRAJECTORY INDEXING

In this section, we elaborate on a bottom-up approach for building the distributed trajectory index. Each data partition builds its local index, and the global index *summarizes* the information about local indexes. We dedicate this section for describing how we partition trajectory data and build local and global indexes.

4.1 Trajectory Partitioning

We use a two-step partitioning approach, in which the data is first partitioned based on trajectory identifiers resulting in all points of each trajectory being ended up in the same machine. Then, we utilize the Sort-Tile-Recursive (STR) [3] method that considers the distribution of points to compute the partition boundaries based on the sample of points drawn from the dataset.

After this step, the challenging task is to distribute the trajectory data to different partitions in possibly different computing nodes. Considering the complexities of trajectory data such as skewness and inherent sequentiality, and avoiding the reconstruction costs, we distribute the whole trajectory objects. That is, all points of the same trajectory end up in the same partition. We assign a trajectory to the partition that its bounding rectangle has the largest intersection with the MBR of the trajectory. In case of ties, one is selected randomly. This technique avoids the reconstruction of trajectories for similarity comparison that may require a huge amount of data shuffle among nodes.

4.2 Local Indexing

Once partitioning is done, a local index should be constructed and cached for each partition to help in executing queries. In this step, we divide a trajectory at some significant points such that the directional trend for each sub-trajectory is preserved. For this reason, we use the popular Douglas-Peucker (*DP*) algorithm [1] to identify those points.

The original idea behind this algorithm is to approximate the trajectory with some points, known as splitting points and discard other points. Different from the original algorithm, we segment the trajectory at those splitting points. We index each sub-trajectory T with ALS^T using a query-only R-Tree. Thus, the *ALS* of sub-trajectories could be used for comparison and pruning. In Figure 2 the whole trajectory is simplified as 3 approximate line segments, between points p_1, p_7, p_{14} and p_{22}. The line segments approximate the original curve by a deviation of at most ϵ [1]:

Lemma 4.1. *Every point in the sub-trajectory T, segmented using a tolerance threshold ϵ, has a distance at most ϵ to the ALS^T.*

Since the exact distance computation between trajectories is expensive, in the search phase we perform most of the comparisons with the *ALS* of trajectories to reduce the size of candidate set (on which the exact distance is evaluated) as much as possible.

4.3 Global Indexing

To locate trajectories at the search phase, we need this global index to keep track of *real* boundaries of data inside partitions rather than computed boundaries after the sampling phase. The real boundaries of all local indexes (i.e. the root node of each local R-Tree) is fetched to the master node to build the global R-Tree index. This index is the first access method utilized in the search phase to prune partitions – and the trajectories inside – that are far away from the query trajectory.

5 DISTRIBUTED SUB-TRAJECTORY SIMILARITY SEARCH

The search procedure is composed of three steps. In the first step, the global index is used to identify partitions that may contain results and discard those that are sufficiently far away from the query. In the second step, for each of the partitions identified, a task is initiated, and the partition's local index is used to find the trajectories whose Minimum Bounding Rectangle (MBR) overlaps with the query sub-trajectory. Before computing the exact distance function, in the third step, we safely prune some sub-trajectories by their *ALS*, and then for the remaining we identify the relevant parts of sub-trajectories to the query sub-trajectory. For those relevant parts, we compute the exact distance function and collect the results.

5.1 Partition Pruning

Given a query trajectory Q and a distance threshold δ, we compute the MBR of the Q and expand the MBR by δ to cover all trajectories which may be adjacent to the query trajectory by its boundaries. Then we query the global index to obtain the partitions that have data boundary overlap with the expanded MBR. It is required to initiate tasks only for these partitions and send the query trajectory to them.

5.2 Search within Partitions

Given a query trajectory Q, a distance threshold δ and an error tolerance ϵ for simplification, simplify Q with the same algorithm described in Section 4.2 and the same ϵ used for segmenting data trajectories. We denote a (simplified) segment of the segmented query as *query segment*.

5.2.1 Pruning by MBR of ALS. For each query segment, we first retrieve all data sub-trajectories from the local R-Tree whose MBR overlaps with its expanded MBR. Here, we expand the MBR of the query segment by a buffer of $2\epsilon + \delta$. The intuition is that, based on Lemma 4.1, the points of data sub-trajectories could be in at most ϵ distance from their *ALS* and the same holds for the query sub-trajectory. Thus, we add 2ϵ to account for the error bound of query and data sub-trajectory. The addition of δ to the expansion is justified in the same way for the partition pruning. This step prunes a large number of segments that could be safely discarded.

5.2.2 Pruning by ALS. For the remaining data segments after pruning by MBR, we do a simple pruning based on the distance computation between two line segments. Indeed, in some cases the MBR of data trajectories overlap with query segments, yet the ALS of data and query trajectories have minimum distance of more than $2\epsilon + \delta$, and therefore the trajectories can be ignored.

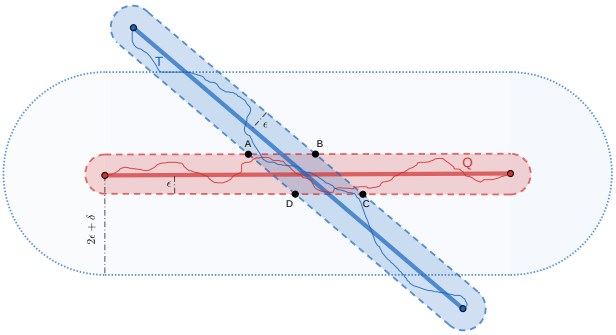

Figure 4: Example of different arrangement of ALS of sub-trajectories.

5.2.3 Arrangement of Query and Data Sub-trajectories. After the filtering of the previous step, the remaining sub-trajectories have ALS within the buffer around at least one of the query segments. Figure 4 illustrates an arrangement between query and data ALSs. The intersection area between the ϵ-buffers around query and data segments is determined by a polygon with vertices A, B, C, D. We search for the similar parts of trajectories inside these polygons.

5.2.4 Relevant Parts of Sub-trajectories. In this step, we find the relevant parts of sub-trajectories by obtaining the parts of the original trajectories inside the intersection polygon. Those parts could be similar and the exact distance should be computed between them. We find the pairs of trajectory points p_i and p_{i+1} that overlap with the sides of the intersection polygon. We interpolate points where the line segment $\overline{p_i p_{i+1}}$ exactly overlaps the side and add them to the final candidate (sub)trajectory. We perform the same procedure for every overlapping point pairs.

5.2.5 Exact Distance Computation. The sub-trajectory pairs obtained from the previous step are the final candidates for the exact distance computation. We use discrete Fréchet distance [2] for this purpose, which is a good measure for curve comparison and capable of maintaining the order of coupling points. If the distance between a data and query sub-trajectory pair is not larger then distance threshold δ, and its spatial length is higher than λ, we add the data sub-trajectory to the results.

6 EXPERIMENTS

In this section, we describe the evaluation of the proposed method on a small cluster of machines. The objective of this experimental evaluation is to assess the performance of the proposed method in terms of query latency and index building time.

6.1 Cluster Setup

All experiments were conducted on a cluster of 5 machines divided as 1 master node and 4 worker nodes. Each machine has a 4-core Intel Core i5-7400 @ 3.00GHz processor and 16 GB main memory. From each machine, 4 GB of main memory and 1 CPU core is reserved for Operating System and Hadoop daemons. Thus, the 4 worker nodes can provide a total of 48 GB of RAM and 3 cores each. Each node runs Ubuntu 18.04.2 LTS with Hadoop 2.7.2 and Spark 2.4.0. All implementations are in Scala programming language v2.11.6.

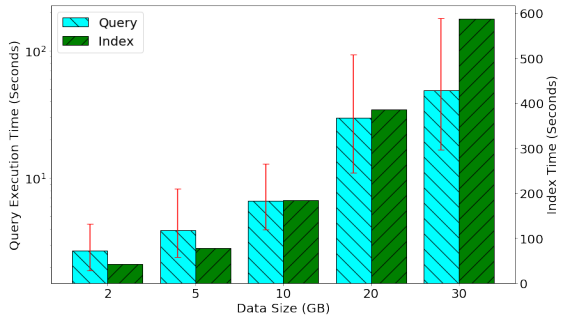

Figure 5: Query and index time with respect to data size.

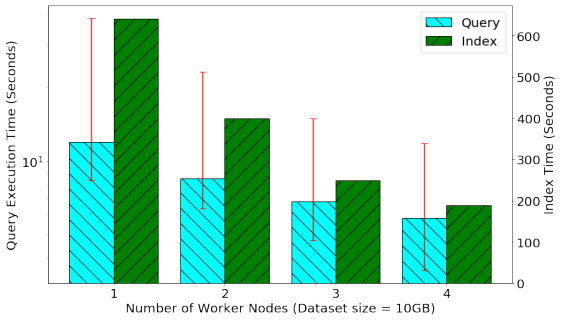

Figure 6: Query and index time with respect to cluster size.

6.2 Dataset

The dataset used in the experiments is a total of 30 GB of GPS traces of one year of car trajectories in the area of Tuscany, Italy. Since the whole track of each user could be very long, if the distance or time interval between two subsequent points is higher than predefined thresholds, we divide the track into two trajectories, corresponding to meaningful trips. We used the cut-off spatial and temporal thresholds of 300 meters and 30 minutes, respectively.

6.3 Index Building Time

The index building is comprised of reading a sample of the dataset, computing the partition boundaries, physical partitioning, building local indexes, and finally building the global index. The building of indexes takes a significant amount of time, but it should be noted that the generated indexes can be stored in HDFS and used later without the need for the recomputing. We study the scalability of our method against the dataset size and cluster size, in building the index structure and querying. The right axes in Figure 5 and Figure 6 show the time for building the index for different sizes of data and different number of worker nodes in the cluster.

6.4 Query Latency

For the query latency, we executed 50 queries randomly chosen from the same dataset and the run-times are averaged. Since the latency of different queries may vary significantly, we also report the latency for the 5% and for the 95% of the run-times. Depending on the location of the query, whether it is in a high density area or sub-urban area, different queries can have different run-times. The left axes in Figure 5 and Figure 6 report the query latencies for different sizes of data and computing nodes. As an example, the query run-time for the 10GB dataset is 6.6 seconds in average with a 5% − 95% interval of [3.9 − 12.9] seconds.

6.4.1 Visualization of Results. Figure 7 shows three examples of a query trajectory and one of the results from data. The dashed (red) and solid (blue) trajectories are query and data, respectively. The markers show the starting point of the trajectories. The highlighted part (in gray) of the data trajectory is the similar part. This figure shows the effectiveness of our method in identifying the similar sub-trajectories in big trajectory dataset.

7 CONCLUSION

In this paper, we proposed a novel approach for sub-trajectory similarity search in a big trajectory dataset. In our proposed method, we rely on approximate line segments of trajectories, as

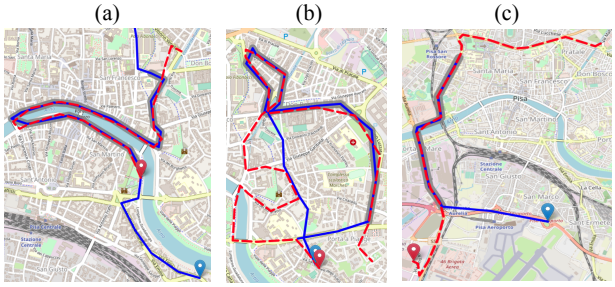

Figure 7: Visualization of results.

working with them is simple and fast. Our aim is to postpone the expensive exact distance computation until the end of a suite of pruning techniques on ALSs. The pruning starts from simple line segment comparisons and continues with ALS matching and extracting relevant parts of sub-trajectories. At the end, we compare the relevant parts and check whether they are similar according to the distance threshold. We performed experiments to analyze the performance of our method, which shows good results in answering queries, and visual representation of some of the results suggests that it can produce significant output.

Our future works on this topic include experimenting the tool on larger datasets and larger parallel platforms, as well as exploring its applicability in applications such as traffic jam detection, carpooling and other clustering-based ones.

ACKNOWLEDGMENTS

This work is partially supported by the European Community H2020 programme under the funding scheme *Track &Know* (Big Data for Mobility Tracking Knowledge Extraction in Urban Areas), G.A. 780754, trackandknowproject.eu.

REFERENCES

[1] David Douglas and Thomas Peucker. 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer* 10, 2 (1973), 112–122.
[2] Thomas Eiter and Heikki Mannila. 1994. *Computing discrete Fréchet distance.* Technical Report. Citeseer.
[3] Scott T Leutenegger, Mario A Lopez, and Jeffrey Edgington. 1997. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings 13th International Conference on Data Engineering.* IEEE, 497–506.
[4] Zeyuan Shang, Guoliang Li, and Zhifeng Bao. 2018. DITA: Distributed in-memory trajectory analytics. In *Proceedings of the 2018 International Conference on Management of Data.* ACM, 725–740.
[5] Panagiotis Tampakis, Christos Doulkeridis, Nikos Pelekis, and Yannis Theodoridis. 2019. Distributed Subtrajectory Join on Massive Datasets. *arXiv preprint arXiv:1903.07748* (2019).
[6] Dong Xie, Feifei Li, and Jeff M. Phillips. 2017. Distributed Trajectory Similarity Search. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1478–1489. https://doi.org/10.14778/3137628.3137655