TSINGHUA SCIENCE AND TECHNOLOGY
ISSNll1007-0214 07/10 pp317–332
DOI: 10.26599/TST.2018.9010087
Volume 24, Number 3, June 2019

Trajectory Big Data Processing Based on Frequent Activity

Amina Belhassena* and Hongzhi Wang

Abstract: With the rapid development and wide use of Global Positioning System in technology tools, such as smart phones and touch pads, many people share their personal experience through their trajectories while visiting places of interest. Therefore, trajectory query processing has emerged in recent years to help users find their best trajectories. However, with the huge amount of trajectory points and text descriptions, such as the activities practiced by users at these points, organizing these data in the index becomes tedious. Therefore, the parallel method becomes indispensable. In this paper, we have investigated the problem of distributed trajectory query processing based on the distance and frequent activities. The query is specified by start and final points in the trajectory, the distance threshold, and a set of frequent activities involved in the point of interest of the trajectory. As a result, the query returns the shortest trajectory including the most frequent activities with high support and high confidence. To simplify the query processing, we have implemented the Distributed Mining Trajectory R-Tree index (DMTR-Tree). For this method, we initially managed the large trajectory dataset in distributed R-Tree indexes. Then, for each index, we applied the frequent itemset Apriori algorithm for each point to select the frequent activity set. For the faster computation of the above algorithms, we utilized the cluster computing framework of Apache Spark with MapReduce as the programing model. The experimental results show that the DMTR-Tree index and the query-processing algorithm are efficient and can achieve the scalability.

Key words: distributed R-tree; trajectory; frequent activity; query

1 Introduction

Geographic information systems use certain utilities, such as Global Positioning System (GPS), to collect the positional data. These data capture the motion history of moving objects, which are named as the trajectories. A trajectory is an ordered series of locations. Some of these locations are called the Point Of Interests (POIs), which are determined by geospatial information, including the latitude and longitude. Each POI may feature an associated description, such as name, address, activities, and other text description.

• Amina Belhassena and Hongzhi Wang are with School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China. E-mail: amina_belhasna@hotmail.fr; wangzh@hit.edu.cn.

∗ To whom correspondence should be addressed.

Manuscript received: 2017-11-22; accepted: 2018-02-22

In general, such movement objects are archived in TraJectory DataBases (TJDBs)[1] for processing and deep analysis to discover knowledge and support the decision making. For example, the application maps installed in smart phones or tablets were developed in cooperation with the public transportation businesses, including taxis, metros, and bus stores, and archive the trajectories of the previous visitors in TJDBs for such analysis to develop and improve the quality of their services.

As the ordered sequence of moving objects is archived in TJDBs, the paths on which these objects are connected are also stored in TJDBs. Therefore, the information related to the frequent trajectory is integrated in the massive TJDBs. Thus, such knowledge is necessary in numerous applications. For example, Foursquare, which is a social network based on mobile applications, provides personalized recommendations of locations to go to the closest place based on users previous browsing history. The locations recommended by this application may be unsuitable for the new visitors, especially when these locations are undesirable areas, for example, areas with unavailable security or surveillance camera. Similarly, the GPS devices use the shortest trajectories in the road network to navigate their users in attaining directions. However, in the case when users want to practice several activities, the shortest trajectory may not be the best option given the bad reputation related to the activity location.

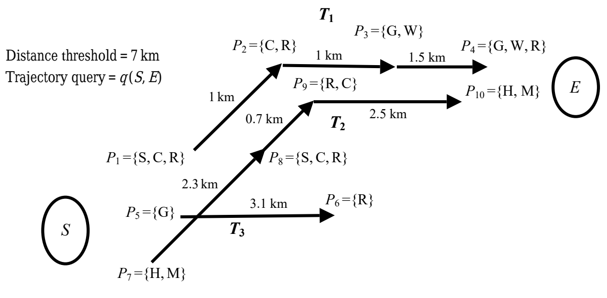
To provide the users with the ability to select the best trajectory, the most suitable POIs based on the distance and the set of desired activities must be provided. The mining frequent trajectory from the TJDBs and its query is not insignificant nor banal. Thus, several researchers have proposed the Keyword-based Trajectory Search (KTS)[2–4]. The KTS query is popular and more suitable in the case of the user needs. Such kind of query is the combination of associated geospatial with keyword information. As the users prefer to visit their POI in short distance, KTS aims to find the K trajectories that contain the most relevant activity keywords based on minimal distance.

Furthermore, the TJDB contains the trajectories created from a massive data of sequence objects, including their activities, producing a large number of sub-trajectories and long trajectories to check. Consequently, the processing of these data needs more computation and exceeds the power of previously used centralized methods. Thus, efficiently traversing the TJDB to extract knowledge and discover the frequent trajectory becomes an important task in the research field and industries.

In this paper, we define the frequent trajectory by the trajectory that can be presented as a path on which many of the activities involved in the following POIs are practiced frequently. To organize the massive trajectory data and extract their knowledge, we propose an efficient method based on two phases. The first phase aims to simultaneously manage the trajectory massive data including its activities on distributed R-Tree indexes. The second phase aims to simultaneously extract the mining from the POIs stored on the leaf nodes of the indexes. The frequent activities and rules are discovered periodically in an offline manner. Thus, to process the proposed KTS query over a large TJDB, we developed an efficient algorithm processed in parallel based on the proposed method. As depicted in Fig. 1, T_1, T_2, and T_3 are historical activity trajectories. The query q is represented by a distance threshold $\widehat{d}$, a start and a final points S and E, respectively, which are presented by the black circles in the figure. The upper table in the figure shows the distance information between the trajectory points of T_1, T_2, and T_3 and query points S and E. The lower table in the figure describes the meaning of POI of T_1, T_2, and T_3. q aims to find the trajectory with a distance less than 7 km including the activity keywords in the following manner. Table 1 presents the notation of the activities in this example.

The user plans to visit specific points in the trajectory. Afterward, she drinks coffee. Then, she will end by taking lunch. Shopping, take-coffee, and take-lunch are described as activity keywords of q which are involved in POIs corresponding to the query. The activities are in a sensitive order.

To help the user to find the best trajectory, we aimed to find the interesting patterns of this query related to the activities, which are in the form of *Shopping*, *take-coffee*, and *take-lunch*, respectively, or $\{(Shopping, take\text{-}coffee) \Rightarrow take\text{-}lunch\}$ where the order of activities appearing in the query is considered.



Euclidean distance Table

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
S	1	3	4	5.5	1.2	4,3	0.5	1.8	3.5	5
E	1	2.5	3.5	4.5	9.1	6	5.2	4.7	4	1.5

POI meaning Table

POIs	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
Significant	Shopping Center	Starbucks	Gym	KFC	McDonald’s	Muslim Restaurant	Shop Mall	Costa Coffee	Gym	Hair_cut/Makeup

Fig. 1 Trajectory example.

Table 1 Activity notation.

Activity	Notation
Take-coffee	C
Shopping	S
Game	G
Take-dinner	R
Take-lunch	R
Hair-cut	H
Watch-cinema	W
Manicure	M

Furthermore, the support of the frequent activities *Sup* and the confidence between activities-set *Conf* are important to return the best trajectory to the user, where the value should not be less than the minimum support and confidence thresholds. We assumed that minSup Threshold = 50%, and minConf Threshold = 60%. Based on the above example, Table 2 represents the Sup and the Conf of the activities implicating in each POI of T_1 and T_2. T_3 is negligible because it involves none of the activities required.

In this example, the process of the trajectory query will return the trajectory with the distance less than the distance threshold, including the frequent activity set with a high *Sup* and *Conf*. Initially, in this example, T_1 and T_2 are returned as the trajectory candidates, because their distances are minimal, and they include all the activities required with the same order as in the query. However, as is described in Table 2, T_1 will be returned as the best trajectory according to its Sup and Conf, where $Sup(\text{SC} \rightarrow \text{R}) = 80\%$, which is greater than the minSup threshold and $Conf\{(\text{SC}) \rightarrow \text{R}\} = 90\%$, which is also greater than the minConf threshold.

To extract the meaningful activity trajectory dataset, the Apriori algorithm[5–7] is initially proposed to generate the frequent pattern itemset with an easy implementation. However, with the huge information in the trajectory dataset, the computation exceeds the power of the classical Apriori algorithm implemented in centralized methods over the scan of a large dataset and computation of the occurrence frequent items in large datasets. Therefore, the parallel computing is well used to accelerate the big data mining[8].

Before mining the large dataset, the massive history trajectory data should be efficiently organized in an index such as the R-tree[9], which is more suitable for handling spatial data. Given the remarkably large data, the single-node-based method used to implement R-tree fails to answer the trajectory query which is based on location, activities, and distance. Therefore, several methods are proposed in parallel computing to implement the R-Tree in the distributed method[10–12]. However, these works involved a large number of I/O disk operations when reusing data.

Table 2 The support and the confidence information.

POI	Act	Sup	Conf
P_1	SC → R	0.8	0.9
P_2	C → R	1.0	–
P_3	G → W	1.0	–
P_4	G,W → R	1.0	–
P_7	H → M	0.5	0.6
P_8	SC → R	0.7	0.8
P_9	R → C	1.0	–
P_{10}	H → M	0.7	0.8

In this paper, we investigated a novel problem of mining distributed large-scale trajectory data. To achieve a high query processing performance, we initially organized the history of geographic trajectory data segments, huge sets of activity keywords, and the results of frequent activity set algorithm on a Distributed Mining Trajectory R-Tree (DMTR-Tree) index.

As the process should support both the trajectory geometric points with activity keyword texts, the DMTR-Tree combines our previous index called DTR-Tree[13] with the Mining Inverted List (MIL). The MIL is a set of inverted lists that stores the outputs of frequent itemset mining algorithm.

Initially, the DTR-Tree is a set of distributed R-trees. Each R-tree stored in such partition includes a set of POIs with their activities. Next, to select the best trajectory, for the POIs stored in the leaf nodes of DTR-Tree, we applied the Apriori algorithm on the activities located at each POI. This step aims to find the frequent activities preferred by the previous users with a good support, and it helps to construct the strong association rules between the frequent activity sets. The result of this step is stored in the list MIL. Furthermore, during the query processing, we observed that the task to traverse MIL is costly and maybe shuffled. Therefore, we adopted a useful optimization strategy based on the traceability method, which achieves a lower cost by reducing the number of inverted lists in MIL. By traversing the concerned small R-Tree with its MIL, we pruned the search space efficiently to obtain the candidate trajectories.

With such index, we needed to reuse an aggregate of data through distributed parallel operations. Moreover, the massive amount of activities generate all the possible itemsets and count their occurrence present difficulty, thus prevent the scalability, as such condition rapidly becomes a combinatorial explosion problem with the increasing input trajectory activity data size. Therefore, we performed the programming tasks on Spark to process our spatial trajectory data mining using a key value method of the MapReduce to accelerate the trajectory query. Our contributions in this paper can be summarized as follows:

- To select the frequent locals with their supports, we applied the Apriori algorithm on each POI stored in the leaf node of each small R-Tree partition. In addition, we constructed the strong association rules and computed the confidence for each rule. The results are stored in the MIL.

- To reduce the massive number of inverted lists of the MIL, we proposed an optimization strategy based on the traceability method.
- To answer the query q, we developed an efficient parallel algorithm consisting of two steps. The first step aims to simultaneously prune the search space efficiently by traversing the corresponding separated indexes. The second one aims to simultaneously select the best trajectory using the optimized MIL.
- The process of trajectory query ensures a good rapidity using the key value method in the distributed Spark cluster.

This paper is organized as follows: Section 2 presents the different works related to the frequent trajectory processing, distributed frequent data mining algorithms, and distributed R-Tree index. Section 3 introduces the problem statement of this study. Section 4 presents the structure of the DMTR-Tree index. Section 5 explains how to process the trajectory query based on the frequent activities. Finally, Section 6 discusses the experimental results and the performance of this study.

2 Related Work

In this section, we review several of existing research on the frequent trajectory processing, distributed R-Tree, and distributed Apriori algorithm.

Discovering frequent pattern mining was first proposed by Agrawal and Srikant[14]. Numerous works introduced the sequential pattern mining based on frequent patterns and association rules mining in trajectories. References [15, 16] presented a method to extract the association rules from a moving object database and returning the best trajectory using a matching function, where Ref. [15] implemented a modified version of the Apriori algorithm, and Ref. [16] used a modified version of the PrefixSpan algorithm.

Moreover, Refs. [17, 18] aimed to identify the frequent subsequences in trajectories. Masciari et al.[17] introduced an approach based on the partitioning strategy to reduce the trajectory size and present the trajectories as strings; then, they utilized the windows approach mixed with a counting algorithm to mine the frequent trajectories. Monreale et al.[18] presented a new technique to predict the next location of a moving object. They built a decision tree named the T-pattern tree to hold a certain area, and it may be used to find the best path of the new trajectories. However, all the methods cited in these works implemented the frequent data mining algorithms in a centralized platform, which becomes insufficient with tending the current big data usage.

Several researchers focused on parallelizing the Apriori algorithm with the MapReduce framework[10, 19–23], improving the results of the centralized method. As the MapReduce is limited when reusing data over the iteration, and as the filtering process of the Apriori algorithm is repeated in each iteration after generating the candidate itemset, the Hadoop cluster based on the MapReduce model may hasten the process of the algorithm[24]. However, this cluster requires additional I/O disks. At present, instead of the Hadoop, Spark[25] in-memory is used to parallelize the massive trajectory data. Spark handles the problem of the Apriori algorithm with the MapReduce using its Resilient Distributed Dataset (RDD). The RDD catches the results of each iteration and provides them efficiently for the next iteration, thus reducing the number of I/O processes from the disk.

Another study[8] proposed the Yet Another Frequent Itemset Mining (YAFIM), which presents a parallel Apriori algorithm developed in Spark. This work proved that Spark is more suitable and effective to implement the Apriori algorithm in parallel compared with the MapReduce-based algorithms. The speedup caused the YAFIM to perform 18$\times$ faster than the Apriori algorithm implemented in the MapReduce framework. On the other hand, Rathee et al.[26] improved the performance of Reduced-Apriori in Spark, thus providing a highly parallel computation.

Furthermore, in order to manage the trajectory big data, trajectory indexing is well developed to organize the massive data using spatial access methods such R-Tree[9], R*Tree[27], and their variants KR*-tree[28] and bR*-tree[29]. A trajectory query based on the keyword search is proposed to process the large trajectory dataset using the hybrid indexes GAT[2] and AC-Tree[30]. For the large datasets, the R-Tree is well developed in distributed platform. The SD-Tree[31] uses interconnected servers to implement the R-Tree, where large datasets are organized into the tree. Based on the MapReduce model spatial Hadoop, the R-Tree was implemented in parallel with a distributed method[32], whereas a Hilbert R-Tree index was implemented on the H-base[33]. However, to ensure the re-usability of data in memory, Apache Spark is suitable for this purpose. The GeoSpark[34] handles spatial data and support spatial access methods, such as the R-Tree using Spark framework; it improves the performance of the previous works based on the MapReduce model and achieves a better run time performance than the spatial Hadoop. On the other hand, to group the same trajectories, trajectory clustering has been widely used in numerous applications. A partition-and-group framework was proposed for clustering trajectories[35]. The trajectory clustering algorithm TRACLUS was also developed. Initially, the algorithm partitions the whole trajectory into a set of line segments at characteristic points. Then, the similar line segments in a dense region are grouped into a cluster.

3 Problem Statement

In this section, we define and present the frequent trajectory query problem studied in this work. Table 3 summarizes the symbols used in this section.

Definition 1. Trajectory A trajectory $traj$ is defined as an ordered sequence of *POI*:

$$traj = \langle POI_0.L_0, POI_1.L_1, POI_2.L_2, \ldots, POI_n.L_n \rangle.$$

Each *POI* includes a group of activities A_i, $POI = [\sum_{i=1}^{k} A_i].L$. $L \in \mathbf{R}^2$ denotes the geographic location of *POI*. Consequently, we can define the trajectory *traj* by the following equation:

$$traj = \sum_{j=1}^{k} \left[\sum_{i=1}^{n} A_i.L \right]_j.$$

Definition 2. Sub-trajectory The sub-trajectory $subT$ is an ordered sequence of *POI*s $\in$ *traj*.

$$subT(s,e) = \langle s.G_s, POI_1.L_1, POI_2.L_2, \ldots, POI_n.L_n, e.G_e \rangle.$$

Table 3 Definition of symbols.

Symbol	Definition
traj	A set of trajectories $traj = \{traj_1, traj_2, \ldots, traj_n\}$
$subT(s,e)$	A set of sub-trajectories included in *traj*, where s and e represent the start and end points, respectively.
POI	A set of point of interest $POI = \{POI_0, \ldots, POI_n\}$ included in *subT*.
$\widehat{d}$	Distance threshold
$A_i.L$	A set of activities that can be found in *POI* where L represents the longitude and latitude.
dis	Geo-location distance measured between *traj* and trajectory query q
f	Evaluated function measured between *traj* and trajectory query q

where $L_i \in \mathbf{R}^2$ refers to the geographic locations of POIs. G_s and $G_e \in \mathbf{R}^2$ are the geographic locations of the start and the final points s and e, respectively. *subT* is organized in a small R-Tree partition index.

Definition 3. Query distance measure In this study, the query distance was used to efficiently traverse the indexes to obtain trajectory matching. The distance measured is based on the Euclidean distance, which is used to measure the shortest distance in the plane with lightweight computation.

Initially, in a given trajectory query $q = (S, E, \widehat{d})$, S and E are the start and the end points, respectively, in q. *dis* denotes the geo-location distance measured between q and *traj*.

$$dis(traj, q) = dis(S, traj) + dis(traj, E).$$

To calculate the distance $dis(traj, q)$, which aides in efficient prunning the search space, we considered $f(q, traj) \in [0,1]$ as the geo-location function measured between *traj* and q:

$$f(q, traj) = 1 - \frac{dis(S.q, POI.traj) + dis(POI.traj, E.q)}{\widehat{d}},$$

where *POI.traj* are the trajectory points, $dis(S.q, POI.traj)$ and $dis(POI.traj, E)$ are the Euclidean distances measured between $(S, POI.traj)$ and $(POI.traj, E)$, respectively.

The short trajectories are more likely to be candidates when the distance threshold is short. Consequently, the geo-location trajectories extracted will be increased, whereas d will decrease. Thus, $f \in [0,1]$. Furthermore, the trajectory query aims to find the frequent trajectories within $\widehat{d}$. Hence, $dis(q, traj) \leqslant \widehat{d}$, $f(q, traj) \geqslant 0$.

In this study, each trajectory point *POI.traj* in the space is organized through the R-Tree index and overlapped by the Minimum Bounding Rectangle (MBR). Thus,

$$f(q, traj) = 1 - \frac{dis(S.q, MBR(POI).traj) + dis(MBR(POI).traj, E.q)}{\widehat{d}}.$$

The distances between the query points S and E and *MBR*s are computed based on the midpoint m of the rectangle representing *MBR*.

$$dis(S.q, MBR(POI).traj) = \sqrt{(x.S.q - x.m)^2 + (y.S.q - y.m)^2},$$

$$dis(MBR(POI).traj, E.q) = \sqrt{(x.E.q - x.m)^2 + (y.E.q - y.m)^2}.$$

As the users prefer to visit the shortest trajectory close to their locations, the *traj* with the minimal distance should be retrieved. The distance d between the first *POI* in *traj* or in *subT* and S in q should be also minimal.

$$d = \min[dis(S.q, MBR(POI_1).traj)].$$

The distance d_n between the trajectory points POI_n

should also be specified.

$$d_n = dis[(MBR(POI_1).traj, MBR(POI_2).traj) + \cdots + dis(MBR(POI_i).traj, MBR(POI_n).traj)].$$

Therefore, the closest and the shortest trajectory query is matched based on f:

$$f(q, traj) = 1 - \frac{\min[d + d_n + dis(MBR(POI_n).traj, E.q)]}{\widehat{d}}.$$

Definition 4. Mining frequent trajectory Given a trajectory *traj* within $\widehat{d}$, initially, the mining frequent trajectory problem is to extract *traj* with its *POI*s belonging to one partition if its length is short. Then, in order to find the frequent activities A_i held in these *POI*s, the inverted list of *POI*s should be visited.

On the other hand, the sub-trajectories *subT*s implicating the frequent ordered activities belonging to different partitions given a lengthy trajectory query are extracted. The inverted lists where the frequent A_is of each *POI* are located, should be visited in order to extract the required *subT*s and combine them to obtain the final matching trajectory.

4 DMTR-Tree

In this study, given the remarkably large trajectory data, difficulty arises from efficiently processing the trajectory query without a sophisticated index. The R-Tree index stores the geometrical trajectory data in a simple way. This index also provides an efficient method to handle the insertion and deletion. However, with the proliferation of the data, a centralized method is insufficient. Therefore, the parallelism solution is indispensable in this case.

As discussed in Section 1, two major tasks are required. The first task is query processing, and the other is frequent itemset mining. As the trajectory data include not only the geometric points but also the activities in each point, the queries may contain constraints on both the geometric points and activities. In our previous study[13], the index was aimed to manage the large-scale trajectory data through a distributed platform. However, to support the frequent itemset, which considers the activities on each geometric points, this index prohibits indexing of the frequent activities. Further, we cannot process the proposed trajectory query through this index. Therefore, to support both tasks, we have modified our previous index[13] to efficiently handle the geometric points and frequent activities.

Furthermore, given the frequent itemset mining algorithm and large items, the Apriori algorithm proves costly in the second iteration during the generation of the candidate set from the singleton items. For fast processing, we applied the algorithm on the small-distributed trees, where each one contains a small set of points holding a set of activities instead of the whole dataset. Hence, the index comprises multiple small-distributed R-trees.

We designed the DMTR-Tree given these considerations. To support the process of both geometric points and activities described by the text, the DMTR-Tree combines the previously proposed index DTR-Tree[13] with MIL. MIL is a sorted list of results of the frequent itemset mining algorithm. The former is a distributed index specific for querying trajectory, whereas the latter is used for the textual query processing. The skeleton of the DMTR-Tree organizes the trajectories on the distributed R-Tree indexes to share the computation capacity between machines on the Spark cluster, where each machine maintains a small R-Tree for the data located on it. With Spark, the number of disks I/O is minimized during in-memory data fitting. For each small index, we mined the association rules to extract all the frequent activities and construct the strong association rules between activities in each point. The activities are organized in the inverted list. Thus, both query processing and mining tasks could be efficiently handled.

4.1 DMTR-Tree structure

A DTMR-Tree combines the DTR-Tree with MIL using the MapReduce as the programming model in the Spark cluster. The master M includes a set of slaves, $\text{Slave}_i = \text{Slave}_1, \text{Slave}_2, \ldots, \text{Slave}_n$, which means that the rectangle in M interconnects with all Slave_is.

We summarized the structure and the construction of the DTR-Tree index and the major principle of MIL in the following paragraphs.

- **DTR-Tree structure:** The R-Tree index aims to store spatial and non-spatial data consisting of objects $o_i \langle o.id, o.L, o.A \rangle$. The object o_i contains three attributes. $o.id$ represents a unique identifier, $o.L$ is the geospatial information determined in latitude and longitude, and $o.A$ contains the activities in the item. Each object belongs to a partition based on its $o.L$. The R-tree uses the MBR to store data based on $o.L$. The data are stored in the leaves referenced with their $o.id$. The DTR-Tree is a collection of R-trees processed by each worker in the cluster. The process of DTR-

Tree construction consists of two phases: parallel partition and local construction[13].

- **Mining inverted list:** We applied the Apriori algorithm to extract the frequent activities and construct the association rules of each point stored in the R-Tree leaf. We stored all the results in the inverted lists, which form the MIL. These lists are stored in the HDFS files.

Using a custom partitioner developed based on a range partition, the trajectory data are grouped into partitions ρ, as shown in Fig. 2a. r_ρ represents the root of R-Tree stored in a partition ρ, $MBR(r_\rho)$ is an MBR of the root (Fig. 2b). As depicted in Fig. 2c, for each partition ρ, a tuple $\langle \rho, MBR(r_\rho)\rangle$ has an inverted list. All the R-tree indexes and the inverted lists are handled within a cluster.

After processing all the R-Trees, the master node uses a list $L = \langle \rho_i, MBR(r_{\rho_i})\rangle$, where $MBR(r_{\rho_i})$ is the MBR of the R-tree stored in the partition ρ_i. Based on the spatial distance used between $MBR(R\text{-}Tree)$ and the MBR of the query points $MBR(q)$, the master node knows whether the search is to be pruned in order to accelerate the query processing.

The trajectory query q is characterized by a start point S and a final point E generated in the master node. Initially, the process of q starts with traversing the indexes to obtain the trajectory candidates. Afterwards, based on the result previously obtained, and the outputs of the distributed Apriori algorithm processing, the best trajectory is sorted and returned as results.

4.2 Optimization

The DMTR-Tree is a set of R-tree indexes combined with the inverted lists in MIL. Each local machine features an R-Tree index storing the trajectory points in the leaf nodes. Each point of this R-tree contains one inverted list containing the results of the frequent activities and association rule algorithms.

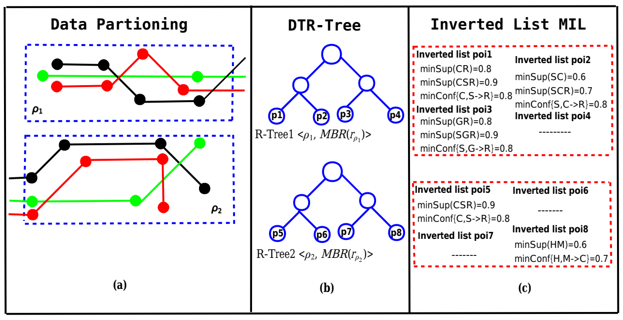


Fig. 2 DMTR-tree structure.

For the query processing method in Section 5, we observed that the task of traversing the huge set of inverted lists by each local machine is costly and may be shuffled, as it requires visiting each inverted list of each trajectory point, storing the primary results, comparing the single frequent activity results, and then selecting the best trajectory according to its support and confidence. To solve this problem, we optimized the traversing of the inverted list tasks on the query processing by collecting the inverted lists based on the traceability method. This process provides us an optimization chance. The following paragraph explains the method proposed to solve such problem.

After processing the distributed frequent activities and the association rule mining algorithms, each point includes an inverted list. Therefore, for each point of the trajectory candidates, its inverted list should be consulted to compute the total *Sup* and *Conf*. Thus, such process is costly and may be shuffled.

Therefore, we planned to collect the inverted lists of the points appearing in the same trajectory or sub-trajectory stored in each local machine using the traceability method, which is defined as the capability to identify an activity point by combining the successive locations of the POIs in the trajectory.

The following algorithm shows the process of collecting the inverted lists. Assuming that ρ_i is the partition stored R-Tree$_i$, this tree includes several trajectory point $poi = \langle traj_{id}, poi_i\rangle$, where $traj_{id}$ is the historical trajectory identifier containing the point of interest poi_i. To ensure the traceability of the activities belonging to poi, we tested the identifier id of the trajectory with the identifier j of poi as in Line 3 of Algorithm 1. Then, we added the results of the frequent mining algorithm to the inverted list of $traj$ if it exists. Otherwise, we constructed a new one (Lines 4 and 7).

In addition, we analyzed the time complexity of the query based on the inverted lists visited. Assuming that

Algorithm 1 Optimization

```
1:  Input: $MIL\langle T_{id}, poi_j\rangle$
2:  for each $poi_j$ of $MIL$ do
3:      if $id = j$ then
4:          if $MIL$ = true then
5:              add $Conf$, $Sup$ of $poi_j$ to $MIL\langle\rangle$
6:          else
7:              $newMIL\langle\rangle()$;
8:          end if
9:      end if
10: end for
```

q is the query returning T as the trajectory candidates, each trajectory T contains P, which presents the number of trajectory POIs. Additionally, each P includes an inverted list. For each P, the running time to traverse its lists is τP. For each T in q, the running time to visit the inverted lists is $\tau = \tau p_1 + \tau p_2 + \cdots + \tau p_n$. Thus, $\tau = \sum_{i=1}^{n}(\tau p)$. The complexity is $\tau = O(n)$. Considering all T of q, the time complexity is $\tau = T \cdot O(n)$.

After collecting the inverted lists based on the optimization strategy previously discussed, the complexity time τ for each trajectory T is $\tau = O(1)$. Thus, the τ for all T in q is $\tau = T \cdot O(1)$.

To visit the inverted lists for the selection of the best trajectory, this analysis demonstrates the low cost of the optimized method. Figure 3 illustrates an example of the comparison of the time consumption for the query q in such R-Tree index.

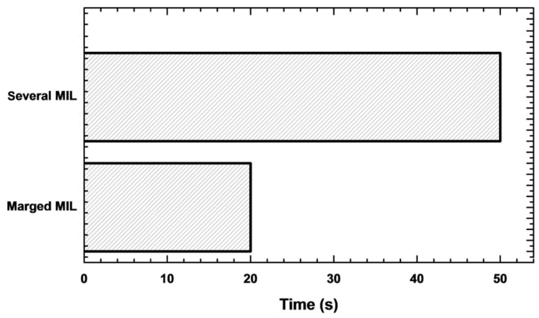


Fig. 3 Comparison methods.

5 Query Processing

The query q to be processed aims to find trajectory t, which includes the frequent activities located on the POIs with a distance of no more than a distance threshold $\hat{d}$. Based on the support and the confidence of the frequent activity set, q returns the best trajectory to users. The master node is considered as the user interface on which q is located. The query q is defined by a start and a final points, S and E, respectively.

$$dis(P, q) = dis(S, P) + dis(P, E) \tag{1}$$

Using function f, which is explained in Section 3, the distance between q and trajectory points P of *traj* can be calculated easily with a lower cost based on the Euclidean distance. To find the trajectory matching q, we passed by the following two phases.

5.1 Phase one

After the construction of each R-Tree by each worker in the cluster, each worker sends the MBR of the index with its partition id to the master. Then, the master node stores all the results through a list. In this phase, the process starts by traversing the index set stored in the HDFS files to obtain the trajectory candidates, from where the distance is less or equal to the distance threshold implicating all the activities required by q.

$L = \langle P_{id}, MBR(rP_{id})\rangle$ contains the partition P_{id} with its $MBR(rP_{id})$, which represents the MBR of the root r of the R-Tree index. Using L, based on Formula (1), the nearest MBRs covering S and E of q within a distance threshold $\widehat{d}$ could be found. Furthermore, to efficiently and effectively prune the search space, in such R-Tree partition if the distance between its $MBR(rP_{id})$ and q exceeds the distance threshold, then visiting all the nodes to obtain the points required by q is unnecessary.

In this study, we considered that there are two types of trajectories: the short ones, which could belong to one partition, and the long ones, which are divided into several sub-trajectories stored in different partitions. Therefore, for each partition j, if *MBR-Root*$_j$ covers S or E, we returned *R-Tree*$_j$ to find the trajectory *traj* or the corresponding sub-trajectories *subT*s.

Algorithm 2 presents the pseudo code for processing the query q. The algorithm finds the trees covering S and E based on L from line 7 to line 12. If the distance between these trees is less than $\widehat{d}$, we searched from line 13 to line 14. We considered two cases to search the trajectory required.

First case: If the *id*s of the partitions are equal (line 15), then we obtained the same partition with the same tree. Thus, the trajectory required is short and it just belongs to one partition.

Algorithm 3 explains the process of this case. Initially, each worker uses the *RDDfind* to return the node that should be visited. This RDD is based on *Filter* procedure (line 5). If the node is a leaf, and if the distance from this node and q does not exceed $\widehat{d}$, then the worker updates the linked list $K\langle\rangle$. Otherwise (for non-leaf nodes), *RDDfind* is invoked by the worker to restart the search recursively (lines 22–35).

The trajectory is extracted according to the linked list $K\langle traj_{id}, p_{wz}\rangle$. K features two attributes: $traj_{id}$, which represents the trajectory key, and p_{wz}, which is the trajectory points of $traj_{id}$ that is determined as $p_{wz} = \langle p_w, \{Act_z\}\rangle$, where p_w is the trajectory point identifier, and Act_z is a set of an ordered sequence of activities involved in $traj_{id}$.

$$Act_z = \begin{cases} \text{TRUE}, & \text{If the ordered activities exist in } p_w; \\ \text{FALSE}, & \text{otherwise.} \end{cases}$$

Algorithm 2 Query processing

```
Input:
    • Query $q$
    • list $L\langle id_i, R\text{-}tree_i\rangle$
    • $\widehat{d}$: distance threshold
Output:
    • Trajectory $traj$ required by $q$
1: initialize $i = 1$
2: $traj = \langle\rangle$ // activity fifo list
3: $Result[\,] = \varnothing$// trajectory table
4: $D[\,] = \varnothing$
5: $j = i$
6: $L' = \langle id_j, R\text{-}tree_j\rangle$
7: for each $R\text{-}tree_i$ in $L$ do
8:     find($S$, $R\text{-}tree_i$)
9: end for
10: for each $R\text{-}tree_j$ in $L'$ do
11:     find($R\text{-}tree_j$, $E$)
12: end for
13: if dis[($S$, $R\text{-}tree_i$)+ dis($R\text{-}tree_j$, $E$)]<= $\widehat{d}$ then
14:     return $R\text{-}tree_i$, $R\text{-}tree_j$
15:     if $i = j$ then // the same partition
16:         Algorithm 3
17:     else
18:         for $k = i$ to $j$ do
19:             find // RDDfind of Algorithm 3
20:             for each sub-trajectory $tn$ do
21:                 if all $Act_z$ of $p_{wz} = true$ then
22:                     Remove $Act_z$ from $traj$
23:                     for each element $z$ of $p_{wz}$ do // traverse the inverted list of $p_{wz}$
24:                         $U$ = Sum(Sup($Act_z$))
25:                         $R$ = extract the rules of $Act_z$
26:                         $C$ = Sum(Conf($R$))
27:                     end for
28:                     return $U, C$
29:                     Store $\{traj_n\}$ with its $U, C$ on $Result[\,]$
30:                 end if
31:             end for
32:         end for
33:         for each element $r$ on $Result$ do
34:             $D$ = $Find\text{-}duplicate(element[r])$
35:             Return $D$ // List $D$ contains the trajectory with the Sup of each activity
36:         end for
37:         TRAJECTORY-CHOICE($D$) // Algorithm 4
38:     end if
39: end if
```

Algorithm 3 Short trajectory case

```
Input:
    • $RTree$
Output:
    • Trajectory $traj$ required by $q$
1: initialize $X[*, *] = 0$
2: initialize $D[*] = \varnothing$
3: Step 1: Traversing index
4: $RDD\ tree = sc.parallelize(RTree[\,]).map(RTree[\,])$
5: $RDD\ find = Tree.Filter(tree).collect(\,)$
6: Step 2: find the trajectory matching $q$
7: Return the updated linked list $K$ by each slave
8: $traj_n$: the trajectory
9: for each trajectory $traj_n$ do
10:     if all $Act_z$ of $p_{wz}$= true then
11:         for each element $z$ of $p_{wz}$ do // traverse the inverted list of $p_{wz}$
12:             $U$ = Sum(Sup($Act_z$))
13:             $R$ = extract the rules of $Act_z$
14:             $C$ = Sum(Conf($R$))// the sum of the Conf of rules containing the activities required
15:         end for
16:         return $U, C$
17:         Store $\{traj_n\}$ with its $U$, $C$ on $D$
18:         return $D$
19:         TRAJECTORY-CHOICE($D$) // Algorithm 4
20:     end if
21: end for
22: procedure FILTER(Tree tree)
23:     $s$: entry
24:     if $dis(s.Entry, q) < \widehat{d}$ then
25:         if $s$ is a non-leaf then
26:             for each child $s'$ node $N$ do
27:                 $s'.FILTER$
28:             end for
29:         else
30:             Update $K\langle id_i, p_i\rangle$
31:         end if
32:     else
33:         break
34:     end if
35: end procedure
```

For example, after extracting the trajectory $traj_1$ in Fig. 1, the linked list K is updated to

$$\langle P_1, \{true, true, true\}\rangle;$$
$$\langle P_2, \{false, true, true\}\rangle;$$
$$\langle P_3, \{false, false\}\rangle;$$
$$\langle P_4, \{false, false, true\}\rangle.$$

The outputs of *RDDfind* are collected to return $K\langle\rangle$ by the master node. Next, for each trajectory candidate, the sum of *Sup* and *Conf* of its required frequent activities could be computed when traversing its inverted list (lines 9–21) in Algorithm 3. Afterward, for the trajectory matching, the *Sup* and *Conf* are stored in the list D (line 17). Finally, using Algorithm 4, the best trajectory is returned as a final result, with the details provided in Section 5.2.

Second case: This case is modeled when the trajectory required is long. Their sub-trajectories are

Algorithm 4 Trajectory choice

```
1: procedure TRAJECTORY-CHOICE(List $D$)
2:     for each $D_n$ of $D$ do
3:         if $(Sup(D_n) > Sup(D_{n+1}))\&\&(Conf(D_n) > Conf(D_{n+1}))$ then
4:             if $dis(D_n, q) < dis(D_{n+1}, q)$ then
5:                 return $D_n$
6:             end if
7:         else
8:             if $(Sup(D_n) > Sup(D_{n+1}))$ then
9:                 if $dis(D_n, q) < dis(D_{n+1}, q)$ then
10:                    return $D_n$
11:                end if
12:            else
13:                return $D_{n+1}$
14:            end if
15:        end if
16:    end for
17: end procedure
```

belonged to multiple partitions. Let i and j be the partitions, where $i \neq j$ (line 17 of Algorithm 2).

Denoting $R\text{-}Tree_{ij} = (R\text{-}Tree_i,\ R\text{-}Tree_j)$ the trees overlapping S and E, respectively. To select the nodes that should be visited in $R\text{-}Tree_{ij}$, the distance between q and $R\text{-}Tree_{ij}$ should be equal or no more than $\widehat{d}$. Therefore, the distance formula could be written as $dis(q, R\text{-}Tree_{ij}) = dis(S, R\text{-}Tree_i) + dis(R\text{-}Tree_j, E) \leqslant \hat{d}$. If the distance is well verified, the sub-trajectories are returned from $R\text{-}Tree_{ij}$ by invoking *RDDfind* as described in line 19 of Algorithm 2.

The process of this RDD is similar as explained in the first case. Then, for each sub-trajectory returned, the master node processes the linked lists. This process saves all the activities required in the first time, and then removes the first one found in the sub-trajectory candidate as in line 22. Afterward, the sum of *Sup* and *Conf* of the frequent activities required in the sub-trajectories is computed by traversing the inverted lists of their points (lines 23–28). Then, based on the table *Result*, all the sub-trajectories with their *Sup* and *Conf* are returned (line 29), also the duplicated trajectories are selected as candidates. Finally, Algorithm 4 is invoked to sort the best trajectory. Its process will be discussed in detail in Section 5.2.

5.2 Phase two

As we have mentioned in the previous sections, we have implemented the Apriori algorithm on each R-Tree partition. The results, including the frequent activities and the strong rules between the activity set with their supports and confidences, are stored in the inverted list in the HDFS files. This phase aims to sort the shortest frequent trajectory candidates with the best support and confidence of the frequent activity set. Assuming that $traj_i = \{traj_1, traj_2, \ldots, traj_n\}$ are the trajectory candidates returned after collecting the linked list $k\langle\rangle$ of the previous phase. Each point p of $traj_i$ contains *Sup* and *Conf* of its activities. Therefore, the support and the confidence of each trajectory are computed as follows:

$$Sup(traj) = \sum_{i=1}^{n} Sup(p_n),$$
$$Conf(traj) = \sum_{i=1}^{n} Conf(p_n).$$

Algorithm 4 uses the *TRAJECTORY-CHOICE* procedure. The previously returned list D is used as the input of the procedure. The algorithm begins by traversing D to compare the trajectories. If *Conf* exists, then we sort the shortest trajectory as the best trajectory corresponding to q, where its *Sup* and *Conf* are higher (lines 3–5). Otherwise, each single activity appears in different points of the trajectory, based on its *Sup*, we sorted the best trajectories where the *Sup* of the frequent activity set is higher (lines 7–10).

For the query activity $q.A = \{A_1, A_2, \ldots, A_n\}$, where A_i denotes the activities included in the trajectory candidates list D, and the comparison time between trajectories of D is $O(n)$.

6 Performance Evaluation

In this section, we evaluate and test the performance of the DMTR-Tree index and trajectory query-processing algorithm. All the algorithms proposed are implemented using Spark in memory computing framework.

6.1 Experimental settings

The algorithms are implemented on Spark-1.6, using the HDFS (version 2.6.0) with Hadoop (version 2.6.0) to store the input datasets, the distributed R-Tree indexes, and the different inverted lists (MIL). All experiments were conducted on a cluster of three machines, with each featuring a 32 GB (4×8 GB) RAM, 64-bit quad-core i7 processor, and four 7200 rpm SATA Disks (4×1 TB). The computing cores are all running on UBUNTU (version 14.02) and Java 1.8 with Maven (3.0.4).

6.2 Datasets

In our experiments, to obtain a large dataset, we used two historical datasets with different features obtained from Microsoft web (http://research.microsoft.com/). The first one (https://www.microsoft.com/en-us/download/details.aspx?id=52367) contains 17 621 trajectories with an average total distance of about 1.2×10^6 km. The second one (https://www.microsoft.com/en-us/research/publication/t-drive-trajectory-data-sample/) contains 10 357 trajectories with 1×10^6 points, and the total distance of the trajectories reaches 9×10^6 km.

As these data contain no frequent activity, we modified them using database managemen tools, such as MySQL and PHPMyAdmin, to add the activity row. Thus, we obtained the dataset rows in the form of (id, latitude, longitude, and activity set). To fill the activity-set rows, we utilized another dataset (frequent itemset mining dataset) crowd from the net. The following is an example of such trajectory point: 7105, 116.42456, 39.86968, [r z h k p z y x].

6.3 Speed performance analysis

To process the frequent trajectory query algorithm, we started by managing the huge data into different distributed indexes. Then, we implemented the frequent mining algorithm for each point in the index. Based on the results obtained during the experiments, we tested the performance and the speedup of the algorithms proposed.

To test the scalability of the DMTR-Tree index and its speedup to manage the massive trajectory data, we started by measuring the changing time with the dataset size and the core numbers in each machine to construct the R-Tree indexes. In addition, the performance of the Apriori algorithm was implemented for all the points of each index before constructing the inverted file MIL.

To test the scalability of the DMTR-Tree, we fixed the node number to three and varied the dataset sizes. Here, the dataset was replicated twice and thrice to allow testing of the scalability of the experiments on the DMTR-Tree. Figure 4 shows the scalability of the distributed indexes. Notably, the time to construct the distributed R-tree slowly increases in the Hadoop compared with the implementation of the indexes in Spark because the MapReduce involves reusing data across multiple operations. Therefore, the Hadoop stores data in the file system at the end of each iteration and then reads them in the following iteration, resulting in high cost. However, the Spark handles the limitations of MapReduce by reusing data across the iteration in memory and without writing nor reading the information from the file system. Moreover, the

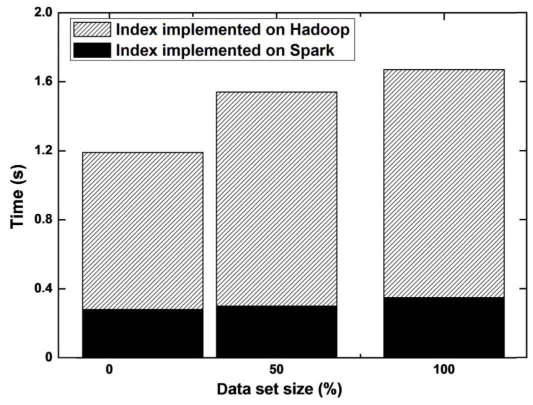


Fig. 4 Scalability of distributed indexes.

Spark Shared Variable (similar to broadcast variables) includes sharing the data between the workers in memory without obtaining them from the file system nor requesting them from the master node in each iteration, thus reducing the communication between nodes and the running time spent in I/O.

This condition is also the same for the implementation of the distributed Apriori algorithm in Spark. As noted in Fig. 5, the Apriori implementation in Spark is faser than MRApriori, which is a standard Apriori implemented on the Hadoop.

However, as our work aimed to find the frequent trajectories including the massive activities, the Apriori process will slow down with the increasing activities. Figure 6 illustrates the performance comparison between Apriori and FP-Growth in Spark. According to Fig. 6, FP-Growth is more efficient than Apriori when data size increases. This result is attributed to the large activity candidate set problem that is solved with FP-Growth[36] by avoiding the candidate generation step.

6.4 Query processing

This work is designed to process the massive trajectory

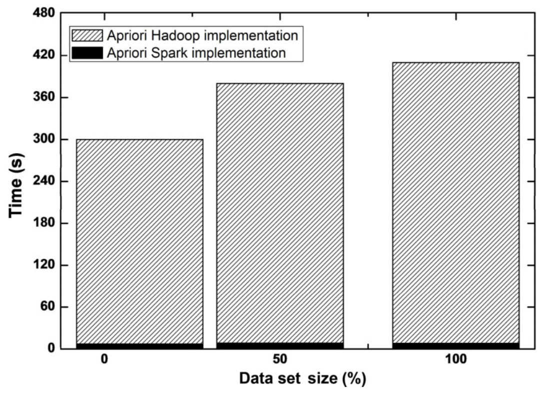


Fig. 5 Apriori performance in SPARK.

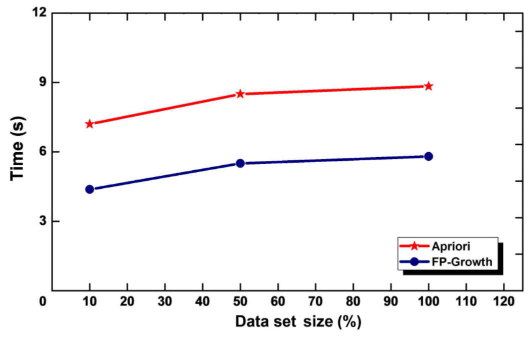


Fig. 6 Apriori vs FP-Growth.

based on frequent activity set. The query aims to find the frequent trajectories required by the users including the most frequent activities within a predefined distance. To evaluate the performance of the query, we analyzed the speedup of the query results based on several impacts, such as trajectory length, distance threshold, activities, and the support, and the confidence of the frequent activity set.

6.4.1 Impact of the distance threshold

To test the effectiveness of the distance threshold, we fixed the activity number to 4, and varied the distance threshold from 5 km to 30 km. As noted in Fig. 7, the number of visited trajectories increases with increasing distance, as there are a large number of trajectories (including both types of trajectories) which are more likely to be candidates with considerable distance.

Moreover, as the query process requires visiting a large number of nodes in the trees, it traverses more R-tree partitions. Consequently, the process requires more time to answer the query when the distance is increased (as shown in Fig. 8).

6.4.2 Impact of activity set

To test the effectiveness of the activity set, we fixed the distance threshold to 10 km and varied the number of activities from 4 to 10. As noted in Fig. 9, the number of

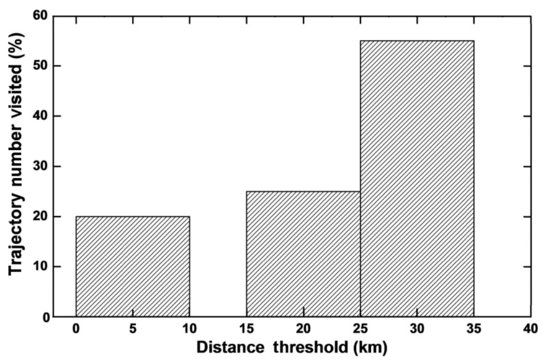


Fig. 7 Number of trajectory query based on distance threshold.

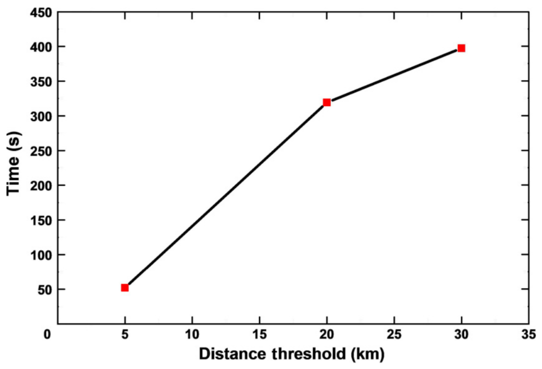


Fig. 8 Distance query performance.

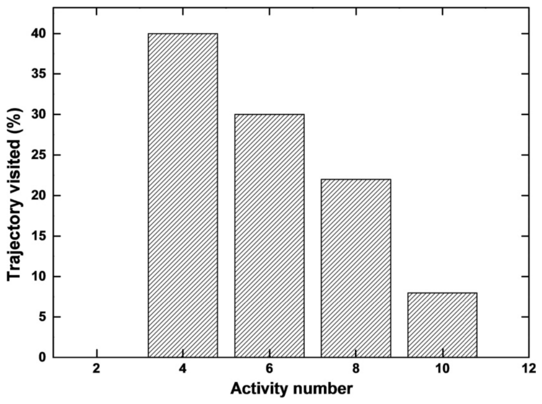


Fig. 9 Number of trajectory query based on activity number.

visited trajectory decreases with the increasing activity number because a few trajectories cover all the intended activities.

Furthermore, as noted in Fig. 10, the query processing consumes substantial running time when the activity number increases, as the query process initially requires visiting numerous indexes to obtain the sub-trajectories, which are then collected and returned to the trajectories as results.

6.4.3 Impact of the confidence of the frequent activity set

To test the impact of the confidence, we fixed the distance to 10 km and varied the confidence

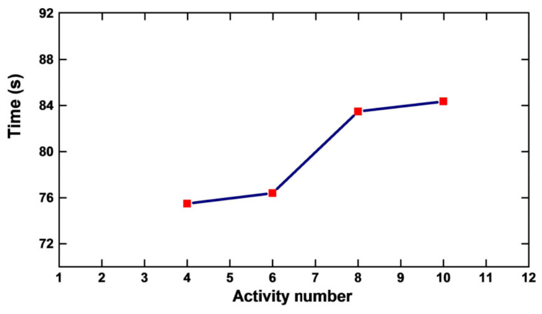


Fig. 10 Activity query performance.

threshold from 0.3 to 0.8. As depicted in Fig. 11, the number of the trajectories increases when the confidence decreases, as the frequent itemset algorithm returns a large number of activities as results when their confidences are greater than the confidence threshold. Consequently, the search becomes less pruned. Therefore, the trajectories are more likely to be candidates in this case.

6.4.4 Results correctness

To test the correctness of the trajectory data results, which are returned by Algorithm 2 in Section 5, we have tested the direction of the trajectories that are stored in the list D through the query q. As the trajectory *traj* is presented by a set of POIs $(POI_1, POI_2, \ldots, POI_n)$. *traj* can be presented as a collection of vectors, $\overrightarrow{POI_1 POI_2}$, $\overrightarrow{POI_2 POI_3}, \ldots, \overrightarrow{POI_k POI_n}$. Table 4 illustrates the analysis of the trajectory results.

As depicted in the table, the system returns "zero trajectory" when the distance measured between the query q and *traj* equals to zero. In this situation, the user has inputted the wrong query points. For example, he/she has inputted the final point as his/her start point. As the query in this example lies in the same location, the distance will be zero.

Further, the system can eliminate the trajectories that are similar to q, but their directions are different

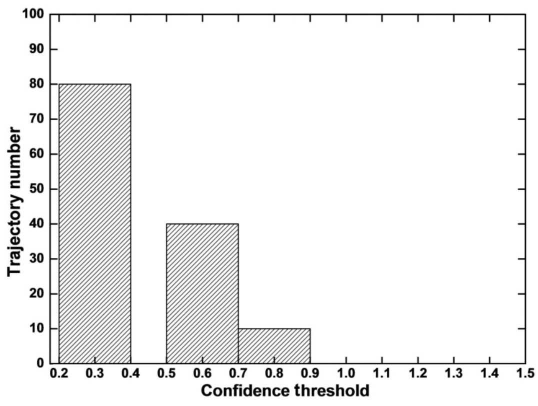


Fig. 11 Activity query performance based on the confidence threshold.

Table 4 Data correctness results.

Distance $d(q, T_D)$	Result	Remark
0	O trajectory	The same points
20	25 trajectories	–
30	55 trajectories, 5 trajectories filter out	The 5 trajectories are similar to q, but they are in the opposite direction

(line 3 in Table 4). Figure 12 presents an example of such scenario, where two trajectories are close to each other. The similarity here is based on the frequent activities and the minimal distance between trajectories. However, this assumption may be incorrect when two trajectories overlap, and their distance will be zero, indicating their different directions (Fig. 12b).

Based on these analyses, we conclude that the proposed method is useful and yields good results.

Comparison In addition, to analyze and compare the trajectory data to find the frequent objects that have moved in a similar way, an efficient clustering algorithm for trajectory dataset can be useful. Therefore, the proposed idea of this paper can be adopted using the trajectory clustering method based on the TRACLUS framework[35]. This framework aims to discover the common sub-trajectories across the clustering process; these sub-trajectories are very useful to analyze regions for special interests, such as POI, in the trajectory.

TRACLUS is based on two phases: partitioning and grouping. The first phase aims to partition the trajectory into a set of line segments using a Minimum Description Length (MDL). The second phase aims to group the similar line segments using the density-based line-segment clustering algorithm. Figure 13 depicts an example of the framework.

To process TRACLUS, two parameters should be feed in, ϵ and *MinLns*. ϵ is the maximum distance

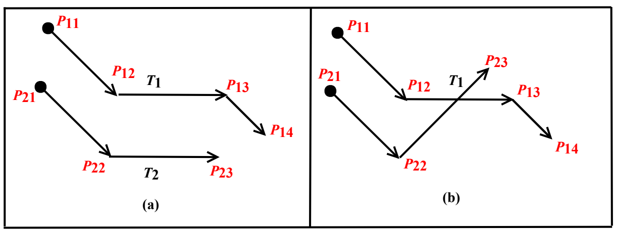


Fig. 12 Trajectory direction example.

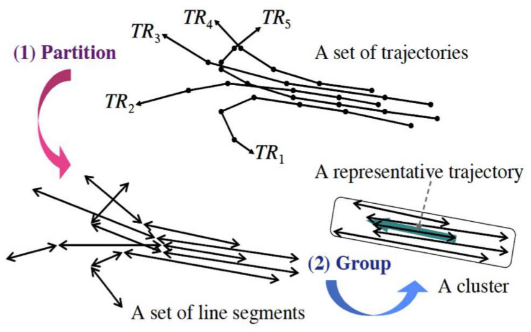


Fig. 13 An example of the partition-and-group framework[35].

determining the neighbor of one line segment, and *MinLns* is the minimum number of line segments required to construct a cluster.

To process our query, we have modified the algorithm to cluster not only the trajectory points but also the activities. However, we need to know which cluster the trajectory query could be belonged to. Therefore, we cannot directly process the query after the clustering.

To adopt our query through this framework, we have implemented an easy method to process the algorithm, as our query is determined by the start and the final points, S and E, respectively, a set of desired activities a_i, and a distance threshold $\widehat{d}$. Our method aims to cluster the query based on the location of S and E. Therefore, q can be presented as a single segment formed by S and E.

However, the partition phase in TRACLUS should achieve two desirable properties, which are preciseness and conciseness. The preciseness is defined by the difference between a trajectory and its partitions, which should be as small as possible. While the conciseness is defined by the number of trajectory partitions, which should also be as small as possible. The preciseness and conciseness are contradictory to each other. Therefore, if only S and E of q are selected as characteristic points (the characteristic points are the points on which the behavior of a trajectory changes rapidly), conciseness is maximized, but preciseness might be minimized. Therefore, we cannot cluster q as a single long segment.

Given a trajectory query q with S and E, $d(S, E)$ is the distance measured between S and E. We suppose θ as the length of trajectory segment. We divide q based on σ where $\sigma = d(S, E)/2$, which is the midpoint measured between S and E, $d(\sigma, S) > \theta$ and $d(E, \sigma) > \theta$.

First, we obtained two segments which are $segment_1 = (S, \sigma)$ and $segment_2 = (\sigma, E)$. Then, we restarted the process in order to find σ of the $segment_1$ and $segment_2$. We continue the process until the distance measured between the points in the segment does not exceed θ. Figure 14 presents an example of the query partition.

However, in our proposed method, if all points of q, including all σ, are selected as characteristic points, then preciseness is maximized, but conciseness is minimized. Therefore, we have adopted the MDL principle as in Ref. [35] to maximize the conciseness based on its component $L(H)$. The other component $L(D|H) = 0$ is not used because the preciseness is maximized.

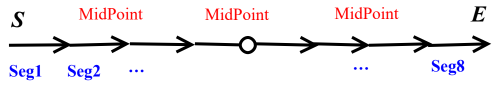


Fig. 14 An example of trajectory query partition.

Based on Euclidean distance, denoting $len(\sigma_j, \sigma_{j+1})$ as the length of a line segment of $\sigma_j \sigma_{j+1}$, $L(H)$ aims to define the sum of the length of all trajectory partitions, as presented in the following equation:

$$L(H) = \sum_{j=1}^{pr-1} \log_2(len(\sigma_j, \sigma_{j+1})).$$

Based on these modifications, we have processed the approximate trajectory partitioning as in Ref. [35]. Furthermore, in order to group the trajectory query into clusters, the density-based clustering algorithm based on DBSCAN was used as in the original papers[35, 37]. We have continued the process of TRACLUS as in Ref. [35]. Finally, based on the trajectory query segments including the points S and E, we have obtained the common sub-trajectories close to q.

However, we have observed that the framework exhibits low efficiency, especially when the query arrives. Thus, the framework should be segmented into clusters based on two parameters ϵ and *minLns*. These two parameters are sensitive. Further, user should have enough information about these parameters, implying that we should feed in the right values of the parameters to help the users. However, if the wrong pair of parameters is feed in, the final clustering results may be valueless, and thus, nothing can be generated. However, as q is varied from user to user, so we cannot know which query will be used by the user. Thus, efficiently generating these parameters presents difficulty. As in our proposed index and the trajectory query algorithm, the user needs not to feed in any parameters nor possess any knowledge about the process to obtain real results. Hence, our study features efficiency and usefulness.

Moreover, in order to compare the analyses of the performance between TRACLUS and our study, we have evaluated the computation time based on the dataset size. Figure 15 shows the comparison results between two studies. As noted, these studies yield a similar running time with that of small data. However as noted in the figure, our study required a short running time, especially in the case of large datasets. Therefore, we have ensured that our work achieves a good rapidity and efficiency.

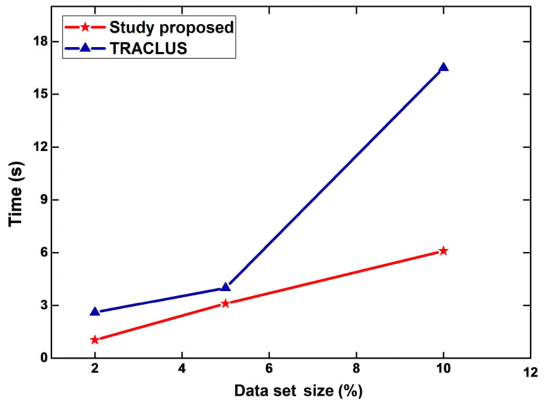


Fig. 15 Performance comparison.

7 Conclusion

An efficient method was used in this paper to handle the problem of the frequent trajectory query. This method is based on the distance and frequent activities holding each POI. To handle this problem efficiently, we proposed the DMTR-Tree index, which organizes the massive amount of trajectory points with their frequent activities. To achieve a good speedup and scalability of the index, we implemented the index in a distributed platform using the Apache Spark and MapReduce model. Furthermore, based on the Apriori algorithm, we extracted the frequent activities and constructed their association rules. To accelerate the Apriori processing, we implemented it on each trajectory point stored in the leaves of each distributed small index instead on the whole data. Experimental results show the good scalability of the index, and the query-processing algorithm is efficient and achieves a rapid response from the users. In the future, we plan to handle terabyte-sized datasets by adding more machines to the cluster and implementing another version of Apriori algorithm to outperform the classical one.

Acknowledgment

This paper was partially supported by the National Natural Science Foundation of China (Nos. U1509216 and 61472099), the National Sci-Tech Support Plan (No. 2015BAH10F01), the Scientific Research Foundation for the Returned Overseas Chinese Scholars of Heilongjiang Provience (No. LC2016026), and MOECMicrosoft Key Laboratory of Natural Language Processing and Speech, Harbin Institute of Technology.

References

[1] H. H. Aung, L. Guo, and K. L. Tan, Mining sub-trajectory cliques to find frequent routes, in *Proc. 13th Int. Symp. Advances in Spatial and Temporal Databases*, Munich, Germany, 2013, pp. 92–109.

[2] K. Zheng, S. Shang, N. J. Yuan, and Y. Yang, Towards efficient search for activity trajectories, in *Proc. 29th Int. Conf. Data Engineering*, Brisbane, Australia, 2013, pp. 230–241.

[3] C. Zhang, J. W. Han, L. D. Shou, J. J. Lu, and T. La Porta, Splitter: Mining fine-grained sequential patterns in semantic trajectories, *Proc. VLDB Endow.*, vol. 7, no. 9, pp. 769–780, 2014.

[4] W. Chen, L. Zhao, J. J. Xu, G. F. Liu, K. Zheng, and X. F. Zhou, Trip oriented search on activity trajectory, *J. Comput. Sci. Technol.*, vol. 30, no. 4, pp. 745–761, 2015.

[5] R. Agrawal and R. Srikant, Fast algorithms for mining association rules in large databases, in *Proc. 20th Int. Conf. Very Large Data Bases*, Santiago de Chile, Chile, 1994, pp. 487–499.

[6] R. J. Jr. Bayardo, Efficiently mining long patterns from databases, *ACM SIGMOD Rec.*, vol. 27, no. 2, pp. 85–93, 1998.

[7] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, Parallel algorithms for discovery of association rules, *Data Min. Knowl. Disc.*, vol. 1, no. 4, pp. 343–373, 1997.

[8] H. J. Qiu, R. Gu, C. F. Yuan, and Y. H. Huang, YAFIM: A parallel frequent itemset mining algorithm with spark, in *Proc. 28th Int. Parallel & Distributed Processing Symp. Workshops*, Phoenix, AZ, USA, 2014.

[9] A. Guttman, R-trees: A dynamic index structure for spatial searching, in *Proc. 1984 ACM SIGMOD Int. Conf. Management of Data*, Boston, MA, USA, 1984, pp. 47–57.

[10] J. Dean and S. Ghemawat, MapReduce: Simplified data processing on large clusters, in *Proc. 6th Conf. Symp. Operating Systems Design & Implementation*, San Francisco, CA, USA, 2004.

[11] A. Eldawy, L. Alarabi, and M. F. Mokbel, Spatial partitioning techniques in Spatial Hadoop, in *Proc. 41st Int. Conf. Very Large Data Bases*, Kohala Coast, HI, USA, 2015, pp. 1602–1605.

[12] H. C. Yang, A. Dasdan, R. L. Hsiao, and D. S. Parker, Map-reduce-merge: Simplified relational data processing on large clusters, in *Proc. ACM SIGMOD Int. Conf. Management of Data*, Beijing, China, 2007, pp. 1029–1040.

[13] H. Z. Wang and A. Belhassena, Parallel trajectory search based on distributed index, *Inf. Sci.*, vols. 388&389, pp. 62–83, 2017.

[14] R. Agrawal and R. Srikant, Mining sequential patterns, in *Proc. 11th Int. Conf. Data Engineering*, Taipei, China, 1995.

[15] M. Morzy, Prediction of moving object location based on frequent trajectories, in *Proc. 21st Int. Conf. Computer and Information Sciences*, Istanbul, Turkey, 2006, pp. 583–592.

[16] M. Morzy, Mining frequent trajectories of moving objects for location prediction, in *Proc. 5th Int. Conf. Machine Learning and Data Mining in Pattern Recognition*, Leipzig, Germany, 2007, pp. 18–20.

[17] E. Masciari, G. Shi, and C. Zaniolo, Sequential pattern mining from trajectory data, in *Proc. 17th Int. Database Engineering Applications Symp.*, Barcelona, Spain, 2013, pp. 162–167.

[18] A. Monreale, F. Pinelli, R. Trasarti, and F. Giannotti, WhereNext: A location predictor on trajectory pattern mining, in *Proc. 15th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, Paris, France, 2009.

[19] N. Li, L. Zeng, Q. He, and Z. Z. Shi, Parallel implementation of apriori algorithm based on MapReduce, in *Proc. 13th ACIS Int. Conf. Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, Kyoto, Japan, 2012, pp. 236–241.

[20] M. Y. Lin, P. Y. Lee, and S. C. Hsueh, Apriori-based frequent itemset mining algorithms on MapReduce, in *Proc. 16th Int. Conf. Ubiquitous Information Management and Communication*, Kuala Lumpur, Malaysia, 2012.

[21] J. Guo and Y. G. Ren, Research on improved a priori algorithm based on coding and MapReduce, in *Proc. 10th Web Information System and Application Conf.*, Yangzhou, China, 2013, pp. 294–299.

[22] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, New algorithms for fast discovery of association rules, in *Proc. 3rd Int. Conf. Knowledge Discovery and Data Mining*, Newport Beach, CA, USA, 1997.

[23] N. Li, L. Zang, Q. He, and Z. Z. Shi, Parallel implementation of apriori algorithm based on MapReduce, *Int. J. Netw. Distrib. Comput.*, vol. 1, no. 2, pp. 89–96, 2013.

[24] W. Tong, C. Rudin, D. Wagner, and R. Sevieri, Learning to detect patterns of crime, in *Proc. European Conf. Machine Learning and Knowledge Discovery in Databases*, Prague, Czech Republic, 2013, pp. 515–530.

[25] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, Spark: Cluster computing with working sets, in *Proc. 2nd USENIX Conf. Hot Topics in Cloud Computing,* Boston, MA, USA, 2010.

[26] S. Rathee, M. Kaul, and A. Kashyap, R-Apriori: An efficient apriori based algorithm on spark, in *Proc. 8th Workshop on Ph.D. Workshop in Information and Knowledge Management*, Melbourne, Australia, 2015, pp. 27–34.

[27] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, The R*-tree: An efficient and robust access method for points and rectangles, in *Proc. 1990 ACM SIGMOD Int. Conf. Management of Data*, Atlantic City, NJ, USA, 1990, pp. 322–331.

[28] R. Hariharan, B. Hore, C. Li, and S. Mehrotra, Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems, in *Proc. 19th Int. Conf. Scientific and Statistical Database Management*, Banff, Canada, 2007.

[29] D. X. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa, Keyword search in spatial databases: Towards searching by document, in *Proc. 25th Int. Conf. Data Engineering*, Shanghai, China, 2009, pp. 688–699.

[30] W. Chen, L. Zhao, J. J. Xu, K. Zheng, and X. F. Zhou, Ranking based activity trajectory search, in *Proc. 15th Int. Conf. Web Information Systems Engineering*, Thessaloniki, Greece, 2014, pp. 170–185.

[31] C. Du Mouza, W. Litwin, and P. Rigaux, SD-Rtree: A scalable distributed R-tree, in *Proc. 23rd Int. Conf. Data Engineering*, Istanbul, Turkey, 2007, pp. 296–305.

[32] A. Eldawy and M. F. Mokbel, A demonstration of Spatial Hadoop: An efficient MapReduce framework for spatial data, *Proc. VLDB Endow*, vol. 6, no. 12, pp. 1230–1233, 2013.

[33] L. Wang, B. Chen, and Y. H. Liu, Distributed storage and index of vector spatial data based on HBase, in *Proc. 21st Int. Conf. Geoinformatics*, Kaifeng, China, 2013, pp. 1–5.

[34] J. Yu, J. X. Wu, and M. Sarwat, GeoSpark: A cluster computing framework for processing large-scale spatial data, in *Proc. 23rd SIGSPATIAL Int. Conf. Advances in Geographic Information Systems*, Seattle, WA, USA, 2015.

[35] J. G. Lee, J. W. Han, and K. Y. Whang, Trajectory clustering: A partition-and-group framework, in *Proc. 2007 ACM SIGMOD Int. Conf. Management of Data*, Beijing, China, 2007, pp. 593–604.

[36] J. W. Han, J. Pei, and Y. W. Yin, Mining frequent patterns without candidate generation, in *Proc. 2000 ACM SIGMOD Int. Conf. Management of Data*, Dallas, TX, USA, 2000, pp. 1–12.

[37] M. Ester, H. P. Kriegel, J. Sander, and X. W. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, in *Proc. 2nd Int. Conf. Knowledge Discovery and Data Mining*, Portland, OR, USA, 1996, pp. 226–231.

Hongzhi Wang is a professor and doctoral supervisor at Harbin Institute of Technology. He received the PhD degree in computer science from Harbin Institute of Technology in 2018. He is a recipient of the outstanding dissertation award of CCF, Microsoft Fellow, and IBM PhD Fellowship. His research area is data management, including data quality, XML data management, and graph management. He has published more than 100 papers in refereed journals and conferences.

Amina Belhassena received the PhD degree in computer science from Harbin Institute of Technology, China in 2018. She received the master degree of Technology in computer science from Abou bakr belkaid Tlemcen University, Algeria in 2012. Her research interest includes massive data computing, data mining, large-scale data management, and data indexing.