CloudTP: A Cloud-based Flexible Trajectory Preprocessing Framework

Sijie Ruan 1, Ruiyuan Li 1, Jie Bao 2, Tianfu He 3, Yu Zheng 1,4†
1 School of Computer Science and Technology, Xidian University, Shaanxi, China
2 Urban Computing Group, Microsoft Research, Beijing, China
3 School of Computer Science and Technology, Harbin Institute of Technology, China
4 Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China
sjruan94@gmail.com liruiyuan@stu.xidian.edu.cn jiebao@hotmail.com
{Tianfu.D.He, msyzheng}@outlook.com

Abstract—Trajectory data preprocessing is to convert raw GPS logs into organized trajectories, which is a common, necessary but tedious task in many urban applications. This paper proposes CloudTP, a cloud-based flexible trajectory data preprocessing framework, to provide an efficient online service, easing the burdens of urban application builders. The proposed system is designed and implemented based on the cloud storage and parallel computing framework (i.e., Spark). Its features consist of 1) noise filtering, 2) trajectory segmentation, 3) map matching, and 4) index building. CloudTP is useful for both normal users and advanced users. By simply uploading trajectory datasets and setting corresponding parameters, normal users can get organized trajectories, statistics and visualizations on the cloud, while advanced users can also customize their own algorithms in any preprocessing module. Finally, usage scenarios are demonstrated to show the capability and flexibility of CloudTP.

Keywords—trajectory processing; cloud computing; map matching; spatio-temporal index

I. INTRODUCTION

Trajectory data, generated by vehicles, sharing bikes, animals, etc, is very useful in many urban computing applications [1], such as traffic modeling [2], urban planning [3], and path recommendation [4], [5]. Trajectory preprocessing is a common, necessary and fundamental step of these applications.

Trajectory preprocessing is important, as the data noise and outliers in raw data affect the accuracy and performance of later applications. However, it is very time consuming due to some high time complexity sub-tasks (e.g., map matching) and the increasing data volume. Furthermore, the trajectories generated by variant objects (e.g., vehicles, bikes and smart phones) are quite different in moving behaviors. There are no universal preprocessing algorithms that can be applied to all of them without any modification.

In this paper, we propose a Cloud-based flexible Trajectory Preprocessing framework, i.e., CloudTP, to ease the burdens on urban application builders. As shown in Fig. 1, CloudTP takes raw GPS logs as input, and generates organized clean trajectories for users. To improve the efficiency of dealing with large-scale raw GPS logs, it leverages parallel computing platform (i.e., Spark) to speed up task execution and the cloud storage to help index building. When the preprocessing is done, the trajectory data is organized in the cloud storage with ID-temporal and spatio-temporal indexes. CloudTP also provides an interface to retrieve the results (e.g., map-matched trajectories), as well as dataset statistics and visualizations to support aforementioned applications. It covers the following sub-tasks in the trajectory preprocessing:

- **Noise Filtering**, which filters abnormal GPS points, e.g., the locations of readings drift significantly out of the range, as these error GPS readings may incur problems in the later data mining and modeling tasks.
- **Trajectory Segmentation**, which not only detects latent information, e.g., stay points [6], but also reduces trajectory computational complexity.
- **Map Matching**, which transforms a GPS point sequence to a set of road segments. It is an essential step in road network related applications.
- **Index Building**, which builds ID-temporal index and spatio-temporal index upon the preprocessed data and organizes them in the cloud storage for query efficiency.

Fig. 1. System Overview.

The flexibility of our proposed system is two-fold: 1) Multiple traveling modes support. All the parameters related to traveling modes are configurable in CloudTP. 2) Pluggable processing modules. Advanced users can customize processing algorithms.

In order to show the capability and flexibility of CloudTP, we demonstrate three scenarios: 1) Normal users use CloudTP to submit jobs and retrieve processed results, 2) Advanced users design custom trajectory processing modules and replace the defaults in CloudTP, 3) A real urban application is demonstrated by leveraging the power of CloudTP.

†Yu Zheng is the correspondence author of this paper.
II. PRELIMINARY

A. Basic Concepts

Azure Storage. Azure Storage provides reliable, scalable storage services including Blob Storage and Table Storage. Blob Storage stores unstructured object data, which is useful to store raw GPS logs uploaded by users and statistic analysis results. Table Storage stores structured datasets. It is a NoSQL key-attribute data store and each entity in a table is uniquely identified by two-level keys, PartitionKey (PK) and RowKey (RK). Entities with sequential keys improve the performance of range queries, which is suitable to construct the temporal index over preprocessed trajectories.

Apache Spark. Apache Spark is a general in-memory distributed computing framework. All the data to be processed will be transformed into Spark core data structure, i.e., Resilient Distributed Dataset (RDD), which is a partitioned collection of data elements distributed over the cluster. Spark supports multiple external data sources, including Windows Azure Storage Blob (WASB). As a result, we can load raw GPS logs directly from Azure Blob to Spark. Trajectory preprocessing can be regarded as RDD transformation operations (e.g., map), while index building and statistic analysis correspond to RDD action operations (e.g., foreach, count).

B. System Overview

The framework of CloudTP is shown in Fig. 2, which contains two layers. The distributed processing layer receives raw GPS logs, efficiently processes, indexes and stores the data in a distributed way. Details will be introduced in Section III. The user interacting layer delivers the job requests of users, and provides an interface for preprocessed data retrieval and statistic analysis. Advanced users can also design custom algorithms for replacement when submitting jobs. Details will be demonstrated in Section IV.

III. DISTRIBUTED PROCESSING LAYER DESIGN

A. Noise Filtering

The noise filtering step filters noise points in raw GPS logs to make later algorithms work correctly.

Implementation. A GPS point is represented by a triple: timestamp \( t \), latitude \( lat \), and longitude \( lng \). Each raw GPS log is encapsulated into class GPSTraj, which is a list of time-ordered GPS points with an object ID. The noise filtering process can be regarded as a map operation to transform raw RDD[GPSTraj] into cleaned RDD[GPSTraj], which is executed in parallel. The built-in noise filtering algorithm is the heuristic-based outlier detection, which has been used in many applications [6], [7]. It removes points with abnormal speed. The speed threshold (Max Speed) is tunable.

Customization. Advanced users can override filter() method to define their own filters. The input and output of the method are both GPSTrajs.

B. Trajectory Segmentation

In the segmentation step, cleaned GPS logs will be segmented into several shorter trajectories based on some criteria.

Implementation. Since each cleaned GPS log may produce several trajectories after segmentation, for further processing convenience, we collect segmented trajectories in each log to form a single RDD[GPSTraj] using flatMap. We provide two commonly used algorithms: a stay point-based [8] method, which segments a trajectory according to stay points, and a time interval-based [6] method, which segments a trajectory if the time interval of two consecutive points is longer than a threshold. Parameters, e.g., max stay time (MST), max stay distance (MSD), and max time interval, can be tuned for different purposes.

Customization. Advanced users can override segment() method to define their own segment algorithms. This method receives a GPSTraj and returns a list of GPSTrajs.

C. Map Matching

The map matching step incorporates road networks to transform GPS trajectories into map-matched trajectories, which reflects the movements of objects on the road.

Implementation. The details of map matching are shown in the left part of Fig. 3. Different from previous steps, we call repartition(M) (M is the number of partitions after the transformation) to shuffle and create more partitions before running the map matching algorithm. Since Spark can only run one concurrent task on each partition, the number of partition is a key factor for parallelism. The reasons for repartitioning...
are based on two observations. On one hand, after trajectory segmentation, we can find some objects moving more frequently than others, which leads to imbalanced task distribution among partitions. On the other hand, as there are much more segmented trajectories than moving objects, we should increase the number of partitions to improve parallelism. After that, we can call map to execute the map matching task. Each map-matched trajectory MMTraj is represented by a list of time-ordered map-matched entities with an object ID, where each map-matched entity is a triple: road segment ID edgeID, enter time t\textsubscript{enter} and leave time t\textsubscript{leave}. An interactive voting-based map matching algorithm [9] is used as default, since it has a good performance even if the sampling rate is low. Whether the matching should follow the road direction (Ignore Road Direction) is tunable.

Customization. Advanced users can also implement other algorithms by overriding match() method. This method receives a GPSTraj and returns a MMTraj.

D. Index Building

In this step, the preprocessed results from previous steps, i.e., segmented trajectories and map-matched trajectories, will be indexed and stored to Azure Table for query. Moreover, if the stay point method is enabled during the segmentation, we record the stay points of each object as well for different application scenarios, e.g., mining interesting travel sequences [8].

As aforementioned, PK and RK are ideal choices for temporal dimension indexes. For example, for segmented trajectories, we treat each GPS point as a table entity, whose PK and RK represent the time that is accurate to hour and the exact time of the point respectively. More specifically, if a point is generated at 16:23:08 on Feb 3\textsuperscript{rd}, 2017, its PK and RK are set as 2017020316 and 20170203162308 respectively. Key design details can be found in our previous works [10], [11].

ID-temporal Indexing. The segmented trajectories, stay points, and map-matched trajectories are stored in corresponding tables named by object ID. For example, in the right part of Fig. 3, we first assign a key to each map-matched trajectory, which is the object ID it belongs to. We then call groupByKey(N) to group trajectories according to the keys, which can reduce the number of partitions to N. Finally, each group of trajectories is inserted into corresponding table (identified by the object ID) in batch using foreach, which avoids the high I/O overhead of frequent small writings. To answer ID-temporal query, we first find the object ID-named table, and then filter the temporal range based on keys.

Spatio-temporal Indexing. GPS points in the segmented trajectories and stay points, are stored in the corresponding tables named by space. We partition the whole space into uniform grids (other static partition techniques will work as well), and assign a key to each spatial data, which is the grid ID it belongs to. Then, the records grouped by grids are stored in corresponding grid tables like ID-temporal indexing. In each table entity, we add its object ID to the original RK as suffix to guarantee the uniqueness, e.g., 20170203162308_001. Due to the cheap storage price and the requirement of query efficiency, the data is actually copied into two copies, one of which is organized by moving objects and the other is organized by spatial grids. As for spatio-temporal query, we first find the grid IDs intersecting with the query range, and then filter the temporal range in each grid ID-named table.

E. Statistic Analysis

Some statistics, such as the number of objects, the number of segmented trajectories, the number of points, spatial minimum bounding box and dataset average sampling rate can be easily collected from the cached RDD. These statistics will be stored in Azure Blob for later visualization. The spatial and temporal distributions of segmented trajectories and stay points are also stored in Azure Blob for later visualization.

IV. Demonstration Scenarios

A. Normal Users

As shown in Fig. 4, normal users can upload raw trajectory dataset to Azure Blob, specify the road networks of a city, tune parameters and submit their jobs through the CloudTP client. Each job will be assigned a unique token. There is a group of radio buttons for traveling mode selection. If one of them is selected, the recommended algorithms and default parameters for each module will be set. When Custom is selected, these settings can be tuned based on users’ domain knowledge and requirement. Once submitted, the request will be delivered to the Spark cluster through our back-end server.
C. Example Application

CloudTP was used in our previous work [3], which provides suggestion on bike lane construction based on sharing-bikes’ trajectories, as shown in Fig. 7(a). A fundamental step in this work is to preprocess the bike trajectories and obtain the map-matched results. In the noise filtering module, we set the maximum speed to 6 m/s by leveraging the flexibility of CloudTP. Since the trip information should be reserved to reflect users’ real demands in this application, we simply remove the segmentation module. In the map matching step, we check the Ignore Road Direction option to loose the matching constraint. At first, it took us 2 days to match one-week trajectories (about 57 thousands trajectories) on the map on a single server with 8 cores and 56 GB RAM. The processing time was significantly reduced to 30 minutes using CloudTP with 5 data nodes (8 cores, 28GB RAM) in the Spark cluster. Finally, the one-month trajectories can be successfully preprocessed within 2 hours, with basic statistics and distributions automatically collected, as shown in Fig. 7(b).

Fig. 5. CloudTP result viewing interface.

Fig. 6. Segmenter module customization.

B. Advanced Users

Advanced users can design their own custom processing modules in each step. For example, if an advanced user wants to segment GPS logs into trajectories with equal number of points, she can design her own module MySegmenter by implementing the given interface. The pseudo code is shown in Fig. 6. The compiled codes in jar archive can be uploaded from the client, and the corresponding class name needs to be specified for replacement, as shown in the Step 2 of Fig. 4.

Fig. 7. Bike lane planning.

ACKNOWLEDGMENT

The work was supported by the National Natural Science Foundation of China (Grant No. 61672399, No. U1609217) and the China National Basic Research Program (973 Program, No. 2015CB352400).

REFERENCES