Invited Paper

Parallel Discovery of Trajectory Companion Pattern and System Evaluation

Yongyi Xian[1,a) Yan Liu[1,b) Chuanfei Xu[1,c) Sameh Elnikety[2,d) Elie Neghawi[1,e)

Received: January 16, 2020, Accepted: June 29, 2020

Abstract: Trajectories consist of spatial information of moving objects. Over contious time spans, trajectory data form data streams constantly generated from diverse and geographically distributed sources. Discovery of traveling patterns on trajectory streams such as gathering and companies enables value domain applications. Such a discovery needs to process arrival records in various sources and correlate across records near real-time. Thus techniques for handling trajectory streams should scale on distributed cluster computing. The challenge is at three aspects, namely a data model to represent the continuous trajectory data, the parallelism of the discovery algorithm, and an end-to-end parallel framework. In this paper, we propose a parallel discovery method that consists of 1) a model of partitioning trajectory samples on various time intervals; 2) definition on distance measurements of trajectories; and 3) a parallel discovery algorithm. We build a stream processing workflow and investigate experiments on a public dataset to evaluate the system's performance, scalability, stability, and data intensity. Our method discovers trajectory gathering patterns with low latency and scales as the size of trajectory data grows.

Keywords: data stream processing, parallel computing, distributed computing, big data

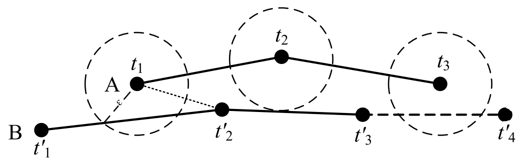
1. Introduction

A trajectory is the sequence of spatial locations or points that a moving object follows over a function of time. Trajectory data are generated by means of location-acquisition technologies such as GPS positioning, sensors probing, mobile phones monitoring and many smart devices. These spatial-temporal location data are usually recorded in the format of trajectory streams [18], [25]. Each record in a trajectory stream consists of a trajectory ID, location (including *latitude*, and *longitude*), and timestamp. In this work, we term trajectory data streams collected from heterogeneous sources as *heterogeneous streaming data*. For example, the Microsoft Geolife project [29] collected the trajectories of objects' outdoor movements that are recorded by different GPS loggers and GPS-phones. Due to heterogeneous GPS positioning sources, the objects' locations can be recorded by different time intervals. Some objects could be recorded every 3 seconds and other objects could be recorded every 5 seconds, which results in heterogeneity of trajectories.

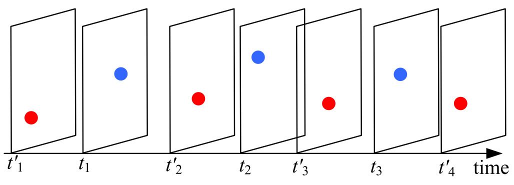
Research effort has been dedicated to discovering groups of objects that travel together over a certain duration of time [8], [11], [19], [20], [27], [28]. In these works, snapshots contains samples of trajectories at every time interval.

A simple case is illustrated in the following example of **Fig. 1**.

[1] Gina Cody School of Engineering and Computer Science, Concordia University, Canada
[2] Microsoft Research, USA
a) yongyi.xian@mail.concordia.ca
b) yan.liu@concordia.ca
c) chuanfei@encs.concordia.ca
d) samehe@microsoft.com
e) elie.neghawi@concordia.ca



(a) Real trajectories



(b) Snapshot model

Fig. 1 Illustrating example of trajectory snapshot model.

Connected Vehicles (CVs) are projected to make the roadways safer through real time exchanging messages containing location and other safety-related information with other vehicles. This requires to discover which vehicles travel together in a duration of time. Weijia Xu et al's work [24] explores the use of real world connected vehicle data set called Safety Pilot Model Delolyment (SPMD) data. The study was conducted in Ann Arbor, Michigan, involved over 2,700 vehicles. Due to the monitoring range limitation of equipments, vehicles' locations are collected from different sources. As shown in Fig. 1 (a), two vehicles' (A and B) locations are collected at $t_1, ..., t_3$ and $t'_1, ..., t'_4$ respectively ($t_i \neq t'_i$). Given a distance threshold ϵ, if the distance between any two collected locations of these two objects is not more than ϵ, we think these vehicles travel together at this time. In Fig. 1 (a), we observe the distances from any collected location of A to the trajectory of B are always less than ϵ. Therefore, they can be regarded as companions. We should discover them from trajectory data. However, based on the snapshot model (see Fig. 1 (b)), we cannot collect both a location of A and a location of B in a same snapshot. Therefore, trajectory companions is non-deterministic in this case.

Current techniques have two limitations. First, the snapshot modeling methods [8], [11], [19], [20] only consider locations of those objects of the same timestamp, while records of objects at different timestamps are missed and thus the relations are not discovered. The second limitation is that most of those techniques are demonstrated on a centralized computing environment. It is not trivial to cope these techniques with large-scale trajectory streams.

The main research we focus on is *how to define a parallel trajectory model that contains continous trajectory objects within a time period?* We further study three detailed research questions in parallelism: *RQ1*) *What is the discovery method of handling heterogeneous streaming data precisely? RQ2*) *How to parallelize the discovery algorithm on a cloud platform to scale? RQ3*) *How to reduce end-to-end delay by improving data locality?*. Our previous work in Ref. [1] targets at the first two research questions and discovers relations of trajectory objects at continous timestamps. We develop a trajectory slot model for compute trajectory companion with objects' locations at each timestamp and objects' movements in a time period. We then define distance metrics to measure densities of trajectories for their companion discovery over time. We develop a discovery algorithm and build a parallel framework to cope with data locality and load balancing.

This paper is an extension to Ref. [1] specifically to investigate the third research question. Our trajectory slot model requires intersection of trajectory objects from partitions of trajectory streams. We analyze the inverted merging method, the self-cartesian set method, the broadcast method, and the inner join hash partition method. The analysis leads to adopt the inner join hash partition method to optimze our trajectory slot model to generate trajectory companies. We further evaluate stability and data intensity in terms of the data shuffling rate with regards to the size of data per time slot. We decompose the execution time to identify the time consuming operations.

Overall, our research work has the contributions in three-fold as follows:

- Devise traveling groups termed Trajectory Companions (TCompanion) on heterogeneous streaming data. Compared to other existing traveling groups, TCompanion represents the heterogeneity of data with accuracy (Section 3, RQ1).
- Design a parallel framework with a suite of techniques and an effective load-balancing strategy (Section 4, RQ2).
- We propose optimization algorithms in our parallel framework to improve end-to-end performance. Compared to one state-of-the-art method, our algorithm can discover more accurate traveling companions with comparable throughput and latency (Section 6.2, RQ3).

The rest of the paper is organized as follows. Section 2 summarizes the related work. Section 3 formally defines the problem and the model in this paper. Section 4 and Section 4.3 present our solutions. Section 6 illustrates the experimental results. Section 8 concludes the paper.

2. Related Work

2.1 Discovery of Traveling Groups

A number of concepts have been defined to discover a *group* of objects that move together for a certain period of time, including *flock* [5], *convoy* [8], *swarm* [11], *companion* [19], [20], and *Gathering* [27], [28]. These concepts can be distinguished by characteristics such as how a *group* is defined and whether the definition requires the time period to be consecutive. These characterizes are further explained as follows:

- *flexible group pattern* denotes a method captures the pattern of any shape. For example, a method that adopts the density-based clustering belongs to this category as objects can form any shape in a density-based clustering group;

- *consecutive time* denotes a cluster of objects lasting for consecutive timestamps. For instance, *companion* groups objects in a density-based clustering and it requires a group of objects to be density-connected to each other during a consecutive time period;

- *stream processing* denotes that this method can cope with updates of trajectory data online;

- *flexible lifetime* denotes that members can join and leave a cluster arbitrarily;

- *parallelism* denotes a method can parallel process trajectory data;

- *heterogeneous pattern* means that the method can handle multiple trajectory sources with different timestamps.

These characteristics affects the accuracy and effectiveness of a discovery traveling groups or companions from heterogeneous trajectories. For example, an algorithms without *flexible group pattern* may miss objects belongs to traveling groups whose location distributions are not regular. As illustrated Example 1, at timestamp t'_1, o_2 leaves o_1 (i.e., the distance is more than the threshold), and then o_2 is back after the following time period. If the method without *flexible lifetime*, we will miss the companions o_1 and o_2.

Existing methods have limitations by satisfying all these characteristics. We summarize the characteristics of these methods in **Table 1**.

In this paper, we propose to discover generic traveling groups and ensure the parallel processing of heterogeneous trajectory streams. To our best knowledge, *Gathering* approach is the current state-of-the-art method. In our experiment described in Section 6.2, we will adjust *Gathering* to parallel processing, and compare our proposed method with it.

2.2 Trajectory Segmentation

In many scenarios, such as trajectories clustering, we need to divide a trajectory into segments often referred to minimum bounding rectangles (MBRs) for a further process. We study different segmentation techniques as follow.

- *Equal-Split*. This technique produces MBRs of fixed time interval of length n. That is, the number of sequence objects. It is a simple approach with the linear cost with regard to the length of a trajectory. The length of MBRs is affected by the number of

Table 1 Group characteristics and discovery methods.

	flock	*convoy*	*swarm*	*companion*	*gathering*	the proposed
flexible group pattern	×	√	√	√	√	√
flexible consecutive time	×	×	√	×	√	√
stream processing	×	×	×	√	√	√
flexible lifetime	×	×	×	×	√	√
parallelism	×	×	×	×	√	√
heterogeneous pattern	×	×	×	×	×	√

splits, k. The increase of k can lead to larger storage space.

- *k-Optimal*. We can discover m MBRs of a trajectory that take up the least volume with a dynamic programming algorithm that requires $O(n^2m)$ computing time [7].

- *Greedy-Split*. An MBR is assigned to each of the n sequence objects. At each object, the consecutive MBRs are merged that will consume the least volume. This has a computing time of $O(nlogn)$. Totally N objects are assigned to k splits. This approach is suitable when one is dealing with object sequences of different lengths with the total complexity of $O(k + NlogN)$.

Although k-Optimal and Greedy-Split introduce least volume consumption, the complexity is higher than Equal-Split. Moreover, space utilization of MBRs is not our major concern. Our main purpose of utilizing MBRs is to prune segment pair by calculating the Euclidean distance between MBRs, which will be described in Section 3.2. For simplicity, we apply Equal-Split in our case.

2.3 Parallel Platforms for Trajectory Data

A number of high-performance parallel platforms exist that can handle data streams [13], [14], [15], [26]. In particular, we study different parallel platforms that are suitable for stream trajectory data.

MapReduce Online [17] pipelines the intermediate data between Map and Reduce operators. MapReduce Online differs from the traditional MapReduce where mappers transmit data to reducers in a push fashion. Therefore, the reducers do not have to wait until the last map task has finished. However, MapReduce Online lacks of the ability to cache intermediate level data, which is crucial for trajectory analysis that consists of massive iterative operations.

Apache Storm [10] is a low latency, in-memory data stream processing system. It implements the data flow model in which data flows continuously through a network of transformation entities. In many enterprise applications, Storm is used as their real-time architecture for integrating and processing streaming data.

Apache Spark [9] is a fast, in-memory data processing framework originally developed by UC berkeley AMPLab. The aim of Spark is to make data analytic program run faster by offering a general execution model that optimizes arbitrary operator graphs, and supports in-memory computing. It uses a main memory abstraction called resilient distributed dataset (RDD) with which spark performs in-memory computations on large clusters in a fault-tolerant manner [26]. Spark Streaming is extended from Spark by adding the ability to perform online processing through similar functional interfaces to Spark, such as *map*, *filter*, *reduce*, and so on. Spark Streaming runs streaming computations as a series of short batch jobs on RDDs, and it can automatically parallelize the jobs across the nodes in a cluster. Also, it supports fault recovery for a wide array of operators.

We finally select Spark as the computing platform for our work. Particularly, we use Spark Streaming to realize the trajectory data stream processing workflow.

2.4 Batch-based Spatial Data Processing

We analyze papers on spatial data processing since trajectories consist of spatial points or locations. Generally, the parallel architecture of batch-based trajectory data processing includes 3 key steps: First, a partition method is chosen to divide all trajectory data into batches. Second, a spatial index for these batches of data is built to improve data search efficiency. Finally, partitioned batch data are run in parallel on a number of nodes. The whole set of trajectory data are stored in HDFS or distributed file systems.

For spatial indices, there exist a large body of research work such as R-tree [6], multi-version B-tree [2], quad-tree [16] and so on. R-tree is one of most popular spatial indexes for multi-dimensional data. The R-tree index height-balances index structure. Objects are represented by MBRs. Each leaf node of the R-tree points to the MBRs of objects and each internal node points to other internal nodes or leaf nodes [6]. For trajectory data, Saltenis et al. [21] propose an extension version of R-tree termed Time-Parameterized R-tree (TPR-tree) that augments the R-tree with velocities to index moving objects. Another often used index structure is the quadtree. Samet [16] has done a thorough survey of the quadtree and the related hierarchical data structures.

In parallel processing, Ahmed Eldawy et al. propose a MapReduce framework for spatial data called SpatialHadoop [3] that adds a simple and expressive high level language for spatial data types and operations. In the storage layer of SpatialHadoop, it adapts traditional spatial index structures to support spatial data processing and analysis.

Above techniques are suitable to batch-based processing that requires all historical trajectory data be archived to construct spatial indexes. In trajectory streaming, only limited trajectory data can be buffered given a time window. Even though the memory resources allow to store massive amount of historical data, updating spatial index at each time window to include newly arrival data streams incurs high cost. Improving the segmentation and indexing techniques in batch processing is not the focus of our solution to trajectory data streaming. Instead, we propose a method with data partition and traveling group discovery in the streaming mode.

3. The Trajectory Slot Model

We define *Trajectory Slots* to denote subsets of trajectories in equal time intervals. Each trajectory slot consists of moving ob-

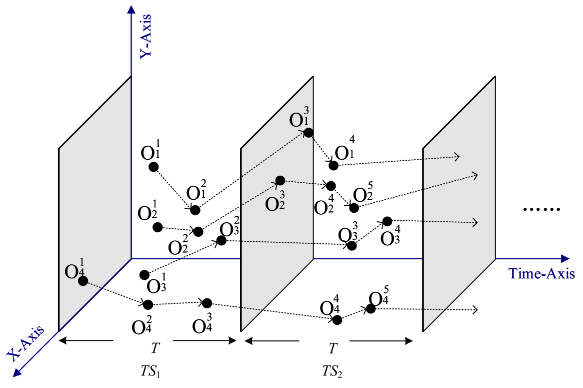


Fig. 2 Trajectory slot model.

Table 2 Commonly used symbols.

Symbol	Description
TS	trajectory Slot
O	set of moving objects
o_i	*i-th* object in O
o_j^x	*x-th* point or location of object o_j
$C_s(o_i)$	coverage set of o_i
n	number of partitions
t	timestamp (in seconds)
$\mathcal{T}$	set of objects' trajectories
$\overline{uv}$	lines in the trajectory
T	duration of trajectory slot (in seconds)
ϵ	distance threshold
μ	density threshold
ℓ	companion size threshold
k	companion time duration threshold

jects from all trajectories within the time period of T. These objects are of different timestamps. For the visualization simplicity, **Fig. 2** plots four moving objects namely o_1, o_2, o_3, o_4 crossing 2 time slots of T. Along the trajectory of each object, a *point* o_j^x, denoted as *x-th* point of object o_j, contains the location information as well as timestamp when object o_j is sampled.

For data streams, new trajectory slot (TS) is generated by arriving trajectory data within every T time. For analysis, a certain number of trajectory slots can be buffered for analysis.

In the time order, points of an object are connected spatially that forms virtual *polylines* of an object. The assumption is if the distance of two polylines within a time slot is measured, and distance is within a threshold value for a sequence of time slots, the two objects are likely forming a traveling company within these time slots.

This model partitions the streaming data arriving in time order into slots. The responsiveness requirement of a discovery method becomes handling objects of all trajectories within T time. **Table 2** summarizes the symbols to be used in this model.

3.1 Concept Definition

Based on trajectory slot model, we formulate our problem of trajectory company discovery with definitions on concepts and a processing framework.

Definition 1. (Slot Trajectory Coverage): Let O_{TS} be the object set in a trajectory slot TS, ϵ be the distance threshold, and $o_i, o_j \in O_{TS}$. o_j is a slot trajectory coverage for o_i denoted by $o_j \triangleright o_i$ if $Dist(o_i, o_j) < \epsilon$, where $Dist$ denotes Euclidean distance between o_i and o_j. Distance measurement will be discussed in Section 3.2.

We further aim to find all the slot trajectory coverage for each object, and combine them into a coverage density set for each object. The definition is given below.

Definition 2. (Coverage Density Reachable): $o_i \in O_{TS}$, the coverage set of o_i contains each object that is a slot trajectory coverage for o_i, denoted as $C_s(o_i) = \{o_j \in O_{TS} | o_j \triangleright o_i\}$. Let ϵ be the distance threshold and μ be the density threshold, object o_i is coverage density reachable from o_j, if $|C_s(o_i)| \geq \mu$.

In Fig. 2, assume ϵ-neighbourhood of o_2 covers o_3 within TS_1. According to Definition 1, we compute $o_2 \triangleright o_3$. Therefore, $o_3 \in C_s(o_2)$. Likewise, assume the ϵ-neighborhood of o_2 covers o_1. So $o_1 \in C_s(o_2)$. In conclusion, $C_s(o_2) = \{o_1, o_3\}$ and $|C_s(o_2)| = 2$. If μ is set to be 2, $|C_s(o_2)| >= \mu$ and thus o_2 is coverage density reachable from o_1, o_3.

Definition 3. (Coverage Density Connection): For $o_i \in O_{TS}$, the coverage density connection of o_i is defined as a set $c_d(o_i)$=$o_i \cup C_s(o_i)$, where $|C_s(o_i)| > \mu$. Following the above example, the coverage density connection at TS_1 is $\{ o_1, o_2, o_3\}$.

Definition 4. (Trajectory Companion): Let k be the duration threshold, and ℓ be the size threshold, trajectory companion is defined as a set of objects, if i) the objects are of coverage density connection for a continuous k trajectory slots and ii) $|TC| > \ell$.

Assume that $k = 2$, $\ell = 3$, $\{o_1, o_2, o_3\}$ is coverage density connection in TS_1 and $\{o_1, o_2, o_3\}$ is a coverage density connection in TS_2. Therefore, the set $\{o_1, o_2, o_3\}$ satisfies the requirements of $k = 2$, $\ell = 3$. Therefore, the set $\{o_1, o_2, o_3\}$ is derived as a trajectory companion given the time periods of TS_1 and TS_2.

The above definition of trajectory companion means objects of trajectories being spatially close enough (within a distance threshold) over a fixed time period. It is not necessary for one object to be close enough to its companion objects at every timestamp. Thus this trajectory slot model has the characteristics of *flexible group pattern*, *flexible consecutive time*, *flexible lifetime* and *heterogeneous pattern* in Table 1.

3.2 Distance Metrics

The concept of coverage density reachable depends on the distance between two moving objects. We propose the Euclidean distance measured by two approaches , namely Point-to-Polyline (P2PL) and Polyline-to-Polyline (PL2PL).

3.2.1 Point-to-Polyline Approach

The Point-to-Polyline (P2PL) measures the minimum perpendicular distance of each point to a line segment pair. Let L_i and L_j be the polylines of the objects o_i and o_j. Assume object o_i moves along the polyline L_i and passes the points $<(x_1, y_1), (x_2, y_2), (x_{m-1}, y_{m-1}), (x_m, y_m)..., (x_n, y_n)>$ in order, where (x_i, y_i) $(1 < m \leq n)$ denotes the spatial coordinate. Given a point of o_j as $p_j = (x_p, y_p)$, and $s_i^{(m)}$ represents the m-th line segment of the polyline L_i of o_i. A vector v perpendicular to the line segment $s_i^{(m)}$ is given by

$$v = \begin{bmatrix} y_m - y_{m-1} \\ -(x_m - x_{m-1}) \end{bmatrix}, \qquad (1)$$

Let r be a vector from the point p_j to the point (x_{m-1}, y_{m-1}) in

$s_i^{(m)}$,

$$r = \begin{bmatrix} x_{m-1} - x_p \\ y_{m-1} - y_p \end{bmatrix}, \tag{2}$$

then the distance from p_j to the $s_i^{(m)}$ is given by projecting r onto v, giving,

$$\begin{aligned} d_m(s_i, p_j) &= |\hat{v} \cdot r| \\ &= \frac{\left|(x_m - x_{m-1})(y_{m-1} - y_p) - (x_{m-1} - x_p)(y_m - y_{m-1})\right|}{\sqrt{(x_m - x_{m-1})^2 + (y_m - y_{m-1})^2}} \end{aligned} \tag{3}$$

Therefore, the distance between any two objects o_i and o_j is defined as the minimum distance among all (s_i, p_j) pairs such that,

$$D(o_i, o_j) = \min_{m \in \mathcal{I}} \{d_m(s_i, p_j)\}. \tag{4}$$

3.2.2 Polyline-to-Polyline Approach

Similar to the point to polyline approach, the polyline to polyline (PL2PL) approach also partitions the trajectories into time slots. We analyze polyline distance relations in each slot. In the d-dimensional space, the polyline of object o_i passes the points $<(x_1^{(1)}, ..., x_1^{(p)}, ..., x_1^{(d)}), t_1>$, $<(x_2^{(1)}, ..., x_2^{(p)}, ..., x_2^{(d)}), t_2>$,..., $<(x_n^{(1)}, ..., x_n^{(p)}, ..., x_n^{(d)}), t_n>$ by order, where $x_i^{(p)}$ $(1 \leq i \leq n)$ denotes p-th dimension coordinate, and t_i denotes timestamp to pass $x_i^{(p)}$. Assume that o_i moves along with this polyline, and keeps uniform speed between two points. The p-th dimension coordinate of o_i can be estimated as

$$x^{(p)} = \begin{cases} x_1^{(p)} + \frac{x_2^{(p)} - x_1^{(p)}}{t_2 - t_1} \cdot (t - t_1) & (t_1 \leq t < t_2) \\ x_2^{(p)} + \frac{x_3^{(p)} - x_2^{(p)}}{t_3 - t_2} \cdot (t - t_2) & (t_2 \leq t < t_3) \\ \dots & \\ x_{n-1}^{(p)} + \frac{x_n^{(p)} - x_{n-1}^{(p)}}{t_n - t_{n-1}} \cdot (t - t_{n-1}) & (t_{n-1} \leq t < t_n) \end{cases} \quad (1 \leq p \leq d) \tag{5}$$

Assume that a location of o_i collected at t_1 timestamp is denoted by $o_i^{t_1}$ and a location of o_j collected at t_2 timestamp is denoted by $o_j^{t_2}$. Following spatio-temporal data processing work [12], this distance should include two parts: spatial distance and temporal distance. In our study, we employ the distance function that considers both space and time factors below.

$$\begin{aligned} &F_\alpha(o_i^{t_a}, o_j^{t_b}) \\ &= \sqrt{SpatialDist^2(o_i^{t_a}, o_j^{t_b}) + \alpha \cdot TemporalDist^2(t_a, t_b)}, \end{aligned} \tag{6}$$

where $SpatialDist(.,.)$ denotes Euclidean distance,

$$SpatialDist(o_i^{t_a}, o_j^{t_b}) = \sqrt{\sum_{p=1}^{d} (x^{(p)} - x'^{(p)})^2} \tag{7}$$

where $x^{(p)}$ and $x'^{(p)}$ are the p-th dimension coordinates of o_i and o_j respectively, which can be obtained by Eq. (5). $TemporalDist(.,.)$ is a normalized time-distance function, applied only within a time slot T, where temporal distance indicates how close between t_a and t_b. The value of this distance is evaluated as

$$TemporalDist(t_a, t_b) = \begin{cases} \frac{|t_a - t_b|}{T} & |t_a - t_b| \leq T \\ N/A & |t_a - t_b| > T \end{cases} \tag{8}$$

The distance between any two objects o_i and o_j is define as

$$D(o_i, o_j) = \min_{t_a, t_b \in \mathcal{I}} \{F_\alpha(o_i^{t_a}, o_j^{t_b})\} \tag{9}$$

4. Parallel Processing Framework

We define a two-phase trajectory processing framework to address the remaining characteristics of stream processing and parallelism in Table 1.

- **The coverage density connection discovery phase.** In this phase, we first utilize the trajectory slot model to set up the unit time T of a slot. We then partition the number of trajectories within each slot into n sub-sets (sub-partitions), where n is a key parameter to adjust the level of parallelism. Next we find all the coverage density connections in each sub-partition.
- **The trajectory companion generation phase.** In this phase, we merge coverage density connections in sub-partitions, and the generate trajectory companions based on results for k continuous trajectory slots.

The processing elements and data flows are illustrated in **Fig. 3**. In our framework, the procedure includes 4 steps:

- *Step I: Trajectory partition*–data within each trajectory slot are partitioned into n sub-partitions;
- *Step II: Find coverage density reachable*–find coverage density reachable for each object in every sub-partition;
- *Step III: Find coverage density connection*–find all coverage density connections in each sub-partition;
- *Step IV: Merge*–find coverage density connections in different sub-partition that have same objects and merge them.

In the following sections, we present the parallel algorithms and techniques on developing the two-phase framework.

4.1 Load-balanced Trajectory Partition

When partition the trajectories, we aim to balance the load on each partition in term of the number of objects in each partition. We apply the K-D tree [4] indexing technique. In the K-D tree, there exist nearly equal amount of data in the tree nodes. The steps is illustrated as Algorithm 1.

First, the algorithm computes the variance of all points in x-dimension and y-dimension respectively. Data in a dimension with a larger variance would have more dispersive distribution, hence the dimension of a larger variance is selected to further split the space. The method computes the variance in each dimension and splits points into smaller regions until the number of regions equals to the fixed number n. The value n is related to the level of parallelism. The way to determine how many partitions to set will be concluded in Section 6.4.

4.2 Parallel Discovery of Coverage Density Connection

Algorithm 2 first discovers the coverage density reachable objects and then combined them into coverage density connection by processing the objects in each partition returned from Algorithm 1.

Algorithm 3 illustrates the polyline-to-polyline (*PL2PL*) distance calculation used in computing the coverage density reachable of each object. Due to the space limitation, we omit the

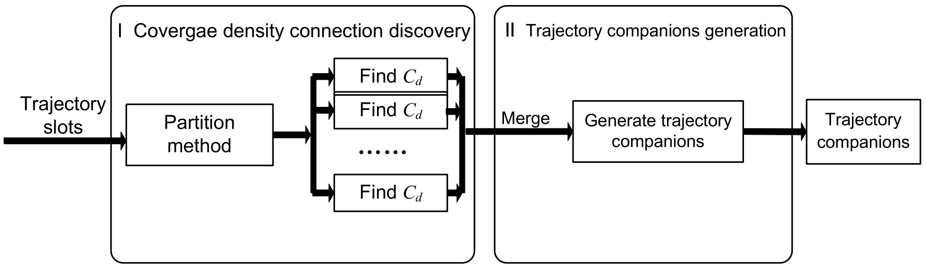

I Covergae density connection discovery
II Trajectory companions generation
Trajectory slots
Partition method
Find C_d
Find C_d
……
Find C_d
Merge
Generate trajectory companions
Trajectory companions

Fig. 3 The structure of two-phase framework.

Algorithm 1: K-D Tree Based Partition

```
Input  : trajectory data $\mathcal{T}$ in a slot, object set $O$, number of partitions $n$
Output: $\{P_1, ..., P_n\}$ and $\{PL_1, ..., PL_n\}$
1  $m \longleftarrow 1$
2  while $m < n$ do
3      for each region do
4          Compute the the variance in each dimension
5          Pick up the middle value in the dimension with lager variance
6          Split the region into two smaller regions with the middle value
7      $m \longleftarrow m + 1$
8  Extend the borderlines of each regions
9  Give each region an ID
10 for each point $v$ of any object in $O$ do
11     if $v$ is located in i-th region then
12         Put $v$ into $P_i$
13 for each line $\overline{vu}$ in $\mathcal{T}$ do
14     if $\overline{vu}$ and i-th region have common points then
15         Put $\overline{vu}$ into $PL_i$
16 Return $P_i$ and $PL_i$
```

Algorithm 2: Coverage Density Connection Discovery

```
Input  : trajectory data $\mathcal{T}$ in a slot, number of partitions $n$
Output: coverage density connections in each sub-partition
1  Call Algorithm I
2  for each object $o_i$ in a sub-partition do
3      Find coverage density reachable from $o_i$
4      Return coverage density connections
```

point-to-polyline distance calculation. Let L_i and L_j be the polylines of the objects o_i and o_j, and s be the segment of a polyline. Each segment pair is in a key-value pair: <<i, j>, <s_i^f, s_j^g>>, where i, j denotes unique identications of polylines (L_i and L_j), and <s_i^f, s_j^g> denotes the segment pair in L_i and L_j respectively (s_i^f is the f-th segment in L_i and s_j^g is the g-th segment in L_j).

We assure the segment pairs within same polyline pair can only be assigned to the same sub-partition. This is done by means of *hash partition* that partitions segment pairs based on the hash codes of the keys (Line 10 of Algorithm 3). Therefore, in the reduce phase, we can use *reduceByKey*() function to put segment pairs with the same key together, and then find the minimum distance of segment pairs in the same polyline pair as the polyline distance (Line 20).

To reduce the data intensity of Algorithm 3, we introduce two pruning rules.

Algorithm 3: Online Discovery Algorithm

```
Input  : $\{\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_i, ...\}$ and $k$
Output: trajectory companions
1  for each $\mathcal{T}_i$ (not empty) do
2      if $i$ mod $k \neq 0$ then
3          Call Algorithm III to discover $\forall c_d$ within $\mathcal{T}_i$
4          Put $\forall c_d$ into a set $\mathcal{DC}$
5      else
6          Find $p_c$ in $\mathcal{DC}$
7          Keep $p_c$ into memory
8          Call Algorithm III to discover $\forall c_d$ in $\mathcal{T}_i$
9          Put $\forall c_d$ into a set $\mathcal{DC}$
10         Delete $\forall c_d$ within $\mathcal{T}_{i-k}$ from $\mathcal{DC}$
11         Merge $p_c$ and $\forall c_d$ in $\mathcal{T}_i$ into trajectory companions
12         Return trajectory companions
```

Pruning Rule I: If the shortest distance between the polyline of o_i and the polyline of o_i is larger than ϵ, then o_j is not slot trajectory coverage for o_i, so it can be pruned safely.

Since the shortest distance between the polyline of o_i and the polyline of o_j is not more than $\min_{t_a,t_b\in\mathcal{I}}\{F_\alpha(o_i^{t_a}, o_j^{t_b})\}$, it is larger than ϵ so that $D(o_i, o_j) \geq \epsilon$.

Pruning Rule II:
If $\min_{t_a,t_b\in\mathcal{I}}\{\sqrt{\alpha \cdot TemporalDist^2(t_a, t_b)}\} > \epsilon$, then o_j is not slot trajectory coverage for o_i, so it can be pruned safely.

Due to $\min_{t_a,t_b\in\mathcal{I}}\{\sqrt{\alpha \cdot TemporalDist^2(t_a, t_b)}\} \leq D(o_i, o_j)$, $\min_{t_a,t_b\in\mathcal{I}}\{\sqrt{\alpha \cdot TemporalDist^2(t_a, t_b)}\} > \epsilon \Longrightarrow D(o_i, o_j) \geq \epsilon$.

4.3 Trajectory Companions Generation on Streaming Data

We propose an online incremental algorithm to compute the Phase II procedure that is the trajectory companions generation on streaming data. In a streaming data application, trajectory data are often received incrementally. As such, the latest batch of trajectory data should be appended to the streams periodically. Our algorithm checks the discoveries from the most recent trajectory slots and decides if they can be extended into companions with the new arriving trajectory data.

First, we introduce a new concept of *promising companion candidate*.

Definition 5. (Promising Companion Candidate): Let k be the duration threshold, and l be the size threshold. A group of objects are promising companion candidates (denoted by p_c), if the group members coverage density connected by themselves for at least continuous $k - 1$ slots and is not less than ℓ.

According to this definition, we first divide the trajectory streaming data into trajectory slots $\{\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_i, ...\}$, where $\mathcal{T}_i$

denotes trajectory data within *i-th* slot. Then we can only check the arriving data in the next trajectory slot to decide whether there exists new arriving coverage density connections (c_d) and save into the density connection set $\mathcal{DC}$. We find p_c that lasts at least $k-1$ slots within $\mathcal{DC}$. If c_d and p_c generate trajectory companions in the k-th slots, these trajectory companions can be found immediately. We conclude the Online Discovery Algorithm in Algorithm 3.

5. Merging Methods and Analysis

To discover promising companion candidates and trajectory companions, we need to find the same objects from all the connections and intersect them. Assume an average M coverage density connections in each slot and we need $k-1$ iterations to generate p_c, then intersecting and merging coverage density connections in each iteration has $O(M^2)$ complexity.

In this section, we propose intersecting and merging methods and formally analyze their effectiveness to improve the runtime performance on the Spark Streaming platform. In particular, we aim to achieve effective data locality and reduce data shuffling. Data shuffling incurs significant cost since it requires frequent data serialization/deserialization, disk I/Os, and even data transmission across physical worker nodes. Poor data locality causes extra data shuffling to occur. We introduce an example below to best analyze the method as follows in Sections 5.1 to 5.4.

Example: Let $\mathcal{DC}$ be the coverage density connection set, $|\mathcal{DC}|$ be the number of coverage density connections. Also, each coverage density connection c_d is in the format of Spark RDDs such as $<<TS_{id}, P_{id}>, \{object_{id}...\}>$, where TS_{id} denotes the identification (ID) of a trajectory slot, P_{id} denotes the ID of a sub-partition in each slot, and $object_{id}$ denotes the ID of an object contained by the same coverage density connection.

Given $\mathcal{DC} = \{c_{d1} = <<4,1>, \{1,2,3\}>, c_{d2} = <<4,2>, \{2,3,4\}>, c_{d3} = <<5,1>, \{5,6,7\}>, c_{d4} = <<5,2>, \{1,3,4\}>\}$, find the same objects from all connections in each time slot and intersect them.

5.1 Inverted Merging Method

The inverted merging method first inverts the keys and values of the RDD of coverage density connections. The inverted pair becomes $<<\{object_{id}...\}>, TS_{id}, P_{id}>$. Then each pair is further mapped to a new set of key-value pairs as $<object_{id}, <TS_{id}, P_{id}>>$. The next step is to reduce values with the same key as $<object_{id}, <TS_{id}, P_{id}>...>$. Eventually based on the values that are the original slot ID and sub partition ID, the original values of objects are merged. This method avoids intersecting every pair of coverage density connects by means of Spark key operations only. Assume that $|C|$ is the average number of coverage density connections that contain the same object in one slot. The number of pairs equals to $n \cdot |C| \cdot |O|$, where $|O|$ is the number of objects in one slot and n is the number of trajectories. This method is still intensive on both computing time and in-memory storage space.

5.2 Self-cartesian Set Method

Cartesian operation returns the cross product of two RDDs or RDD to its identical self. Given the above example, we compute the self-cartesian such that $\mathcal{DC} \times \mathcal{DC} = \{(c_{di}, c_{dj}) | i \neq j$ and $c_{di}, c_{dj} \in \mathcal{DC}\}$ in the following matrix,

$$\mathcal{DC} \times \mathcal{DC} = \begin{bmatrix} - & (c_{d1}, c_{d2}) & (c_{d1}, c_{d3}) & (c_{d1}, c_{d4}) \\ (c_{d2}, c_{d1}) & - & (c_{d2}, c_{d3}) & (c_{d2}, c_{d4}) \\ (c_{d3}, c_{d1}) & (c_{d3}, c_{d2}) & - & (c_{d3}, c_{d4}) \\ (c_{d4}, c_{d1}) & (c_{d4}, c_{d2}) & (c_{d4}, c_{d3}) & - \end{bmatrix}$$

In the worst case scenario, assuming c_{d1}, c_{d2}, c_{d3}, and c_{d4} are all located in different worker nodes, c_{d2}, c_{d3}, and c_{d4} need to shuffle to where c_{d1} is located to form the first row of the matrix. This costs $|\mathcal{DC}| - 1$ times data shuffling since we ignore the self-contained pair. Similarly, the same procedure repeats for the rest of rows. Therefore, the total cost of data shuffling, Γ_1, is calculated as

$$\Gamma_1 = (|\mathcal{DC}| - 1) \cdot |\mathcal{DC}| \simeq |\mathcal{DC}|^2. \tag{10}$$

Next, we intersects each pair (c_{di}, c_{dj}). We observe that 66% of pairs (e.g. (c_{d1}, c_{d3})) are within different time slots meaning they should not be merged. This indicates the cartesian method produces unnecessary pairs that would not be merged eventually.

5.3 Broadcast Method

Broadcast variable allows a read-only dataset to be shared throughout the cluster. Assuming c_{d1}, c_{d2}, c_{d3}, and c_{d4} are in separate worker nodes each. A broadcast method first collects $\mathcal{DC}$ on the driver node and stores it as a broadcast variable. Next, the broadcast variable is redistributed back to each worker node such that they all own a copy of $\mathcal{DC}$. On each worker node, we then compute $c_{di} \cdot \mathcal{DC} = \{(c_{di}, c_{dj}) | i \neq j$ and $c_{dj} \in \mathcal{DC}\}$. In our example, we obtain the result as the following,

$$c_{di} \times \mathcal{DC} = \begin{cases} c_{d1} \times \mathcal{DC} \rightarrow (c_{d1}, c_{d2}), (c_{d1}, c_{d3}), (c_{d1}, c_{d4}) \\ c_{d2} \times \mathcal{DC} \rightarrow (c_{d2}, c_{d1}), (c_{d2}, c_{d3}), (c_{d2}, c_{d4}) \\ c_{d3} \times \mathcal{DC} \rightarrow (c_{d3}, c_{d1}), (c_{d3}, c_{d2}), (c_{d3}, c_{d4}) \\ c_{d4} \times \mathcal{DC} \rightarrow (c_{d4}, c_{d1}), (c_{d4}, c_{d2}), (c_{d4}, c_{d3}) \end{cases}$$

where the equation in each roll represents the computation on each worker node. Finally on each worker node, the method iterates through pairs of coverage density connections locally to merge them. During this process, the total communication cost, Γ_2, is calculated as two rounds transmission of $\mathcal{DC}$ among w worker nodes such that,

$$\Gamma_2 = 2 \cdot w \cdot |\mathcal{DC}|. \tag{11}$$

When $\Gamma_2 < \Gamma_1$, the performance of the broadcast method is better than the self-cartesian set method. The assumption of the broadcast method is that the collected data size should fit in memory of the driver node, otherwise out-of-memory exception could lead to runtime failure to the driver node. Therefore, the method is limited to the size of $\mathcal{DC}$.

5.4 Inner Join Hash Partition Method

In Spark, partitions are each stored in a worker node's memory. One worker node may contain one or more partitions but a partition never spread on different worker nodes. By this means, an aggregation can be processed locally without shuffling if data of the same key or hashing result of the key are in the same partition. We propose a new method called Inner Join Hash Partition (IJHP) that simply hash the key to a partition as *key.hashCode()%numPartitions*. In IJHP, the slot id $slot_{id}$ is considered as the key that means density connections of the same time slot are guaranteed to be in the same partition. The data shuffling occurs when a density connection is not within the node it is hashed to. Therefore the data shuffling cost is at most the size of the density connections. As the estimation below, a factor β ($0 < \beta \leq 1$) denotes the portion of the coverage density connections that need to be shuffled,

$$\Gamma_3 = \beta \cdot |\mathcal{DC}|. \qquad (12)$$

Since $\Gamma_3 < \Gamma_2$, IJHP has the minimal data shuffling cost. Hence we decide to use the IJHP method to generate trajectory companions.

6. Evaluation

We conduct a thorough evaluate of our proposed methods on a real dataset with three algorithms: (1) snapshot based model called gathering method [27], [22]; (2) our trajectory slot model with the point-to-polyline distance measures (referred as TCompanion-P2PL) [23] [*1]; and (3) the trajectory slot model with the polyline-to-polyline distance measures (referred as TCompanion-PL2PL). We evaluate the following the quality attributes, namely Precision and Recall, Performance and Scalability, Data Shuffling and Intensity and Stability.

All experiments are conducted on Amazon Web Services (AWS). We use one driver node and up to sixteen worker nodes in AWS. The deployment settings in our experiments are as follows: each node is of AWS EC2 t2.large with 2 cores each and 8 GB memory. Amazon S3 was used for storing the original dataset. We run Apache Spark 1.5.1. We utilize Kafka to inject the data streams to the two-phase framework running on Spark Streaming [*2].

6.1 Data Samples

We use a real GPS trajectories dataset collected by the Geolife project in Microsoft Research Asia [*3]. The dataset contains 78 real users in a period over four years (from April 2007 to October 2011) and records a broad range of users' outdoor movements, including daily routines of commute as well as entertainment and sport activities, such as shopping, sightseeing, dining, hiking, and cycling. There are 17,621 trajectories and over 20 million location records with a total distance of about 1.2 million kilometers. These users' trajectories were recorded by different GPS loggers and GPS-phones with a variety of sampling rates. To generate intensive workload for the experimental purpose, we ignore the date attribute of each GPS trajectory, so that they are regarded as the trajectories within one day. Each location record contains the information of latitude, longitude and timestamp of an object. **Table 3** shows our parameter settings. α default value is 1 means that both time and space factors in the distance function have the same weight. In experiments, over 10 thousands new locations of trajectory objects are injected per second.

Table 3 Parameter settings.

Factor	Range	Default
ϵ	$\{0.00005, 0.0001, 0.0005, 0.001, 0.005\}$	0.0001
α	$[0, 1]$	1
n	$[2, 16]$	2
T	$[40, 80]$	60

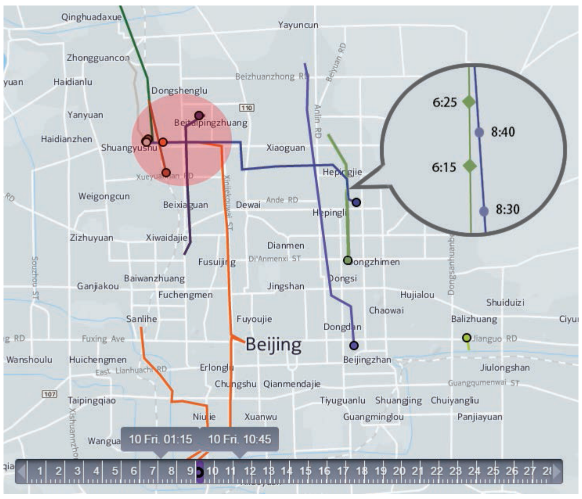


Fig. 4 Trajectory visualization.

To evaluate the precision and recall of each algorithm, we need *ground truth* by sampling the trajectories of objects within a small region (e.g., 1 km×1 km). We choose a fixed number of objects in this region. We then extract locations of these objects with a time period. Finally, we select objects that have at least one location within the distance threshold to trajectories of the original chosen objects.

We plot the trajectory samples in **Fig. 4**. The static view shows the visual hints which trajectories are spatially close to each other. Some objects' trajectories are always close but do not imply they are in companion without considering the time constraint. Consider the two trajectories within the zoomed area, each trajectory has two timestamps (shown in diamond and circle accordingly), indicating the object traveled to the north from 6:15am to 6:25am. The second object traversed the same path about 2 hours later (8:30am - 8:40am). Technically, these trajectories are not companions.

To further refine a *true trajectory companions*, we measure each pair of objects $(o_i^{t_1}, o_j^{t_2})$ by applying equation 8. If any two trajectories have k or more pairs meeting the requirement, we conclude that they are the *true trajectory companions*. Following this procedure, we discover 4 object's trajectories are companions (northwest circle in red in Fig. 4). Thus, we consider these results as *ground truth*.

6.2 Precision and Recall Evaluation

We used 12 selected sample trajectories and the predefined ground truth results to evaluate the precision and recall for each

[*1] we developed P2PL parallel implementation on Spark in batch processing corresponds to Section 2.4
[*2] http://spark.apache.org/docs/latest/streaming-kafka-integration.html
[*3] http://research.microsoft.com/en-us/downloads/b16d359d-d164-469e-9fd4-daa38f2b2e13/default.aspx

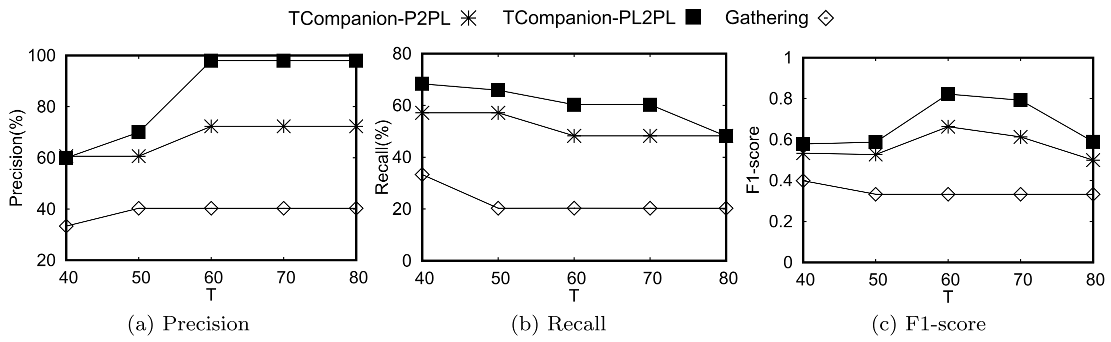


Fig. 5 Precision, Recall and F1-score.

algorithm. Suppose we use the algorithm x to find a set of trajectory companion pairs (denoted by $\tilde{TC}(x)$), and the set of ground truth trajectory companion pairs is denoted by $TC(x)$. The precision is computed as

$$Precision(x) = \frac{|\tilde{TC}(x) \cap TC(x)|}{|\tilde{TC}(x)|} \times 100\% \tag{13}$$

Also, the recall is computed as

$$Recall(x) = \frac{|\tilde{TC}(x) \cap TC(x)|}{|TC(x)|} \times 100\% \tag{14}$$

Lastly, the F1-score is computed as

$$F_1(x) = \frac{Precision(x) \times Recall(x)}{Precision(x) + Recall(x)} \times 100\% \tag{15}$$

Figure 5 shows the precision, recall and F1-score of our proposed algorithm (TCompanion) and the competitor algorithm (Gathering) [22], [27]. We vary the size of trajectory slot T from 40 to 80 seconds. For the Gathering algorithm, T represents the bound of any two snapshot groups within a gathering. We can see our algorithm has higher precision recall and F1-score. For our algorithm TCompanion, the precision increases and the recall decreases with increasing value of T. As T increases, discovered trajectory companions need to maintain the status of being close enough for kT period of time. Lager T leads to lower precision and higher F1-score. When $T = 60$, the F1-score is highest. We set 60 seconds as default value in the following experiments.

6.3 Performance Comparisons

We compare the throughput and latency of algorithms of Gathering and TCompanion. The throughput in term of location points processed per second is

$$\text{throughput} = \frac{\text{totalnumberoflocations}}{\text{procesingtime}}$$

and the average latency of processing each location record is

$$\text{latency} = \frac{\text{procesingtime} + \text{waitingtime}}{\text{totalnumberoflocations}}$$

The Gathering algorithm processes data in one snapshot each time. Since the data size within each snapshot is not big, the efficiency is not improved by scaling out the computing nodes. Hence, we set $n = 2$ for the performance evaluation experiments.

Figure 6 illustrates results by varying the distance threshold ϵ. Both throughput and average latency of the two algorithms are

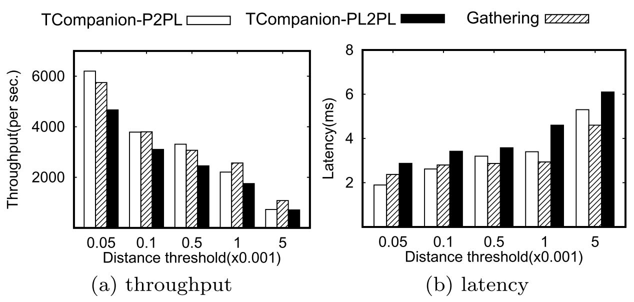


Fig. 6 Throughput and latency comparison.

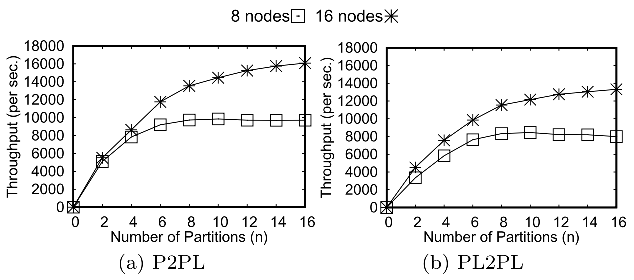


Fig. 7 Scalability comparison.

comparable. TCompanion has more distance computation since it considers all the timestamps within one time slot.

Combined the evaluation results on precision and recall, and performance, TCompanion discovers traveling companions with better accuracy and comparable runtime performance to that of Gathering. Given this observation, we focus on experiments in the following sections to further identify key factors contributing to system level quality attributes.

6.4 Scalability Evaluation

We scale out the number of worker nodes to observe the horizontal scalability between the algorithm of TCompanion with two distance metrics P2PL and PL2PL. In AWS, we deploy 8 to 16 nodes. **Figure 7** plots the throughput under different size of the cluster.

Each executor is running on one node in the cluster. By increasing the number of partitions and adding more cluster nodes, the system produces optimal throughput as the numbers of partitions and executors reach sixteen. When the number of partitions are larger than the number of executors, excessive partitions need to wait for available executors till any parallel partition completes, representing the saturation regimen. When the number of

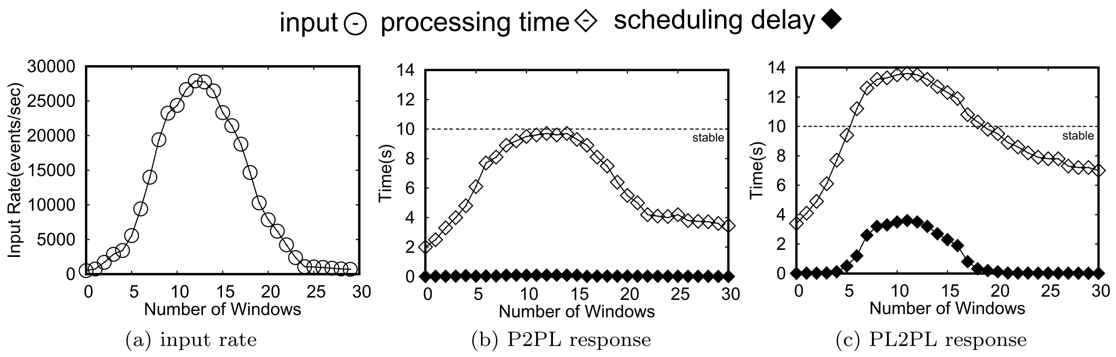


Fig. 8 Stability comparison - input rate, processing time and scheduling delay.

partitions is low, only partial executor nodes run on partitioned data with high workload while other executors are idle, which also degrades performance. The experiments shows the algorithms with both distance metrics scale as the number of executor nodes and the number of partitions grow together.

6.5 Stability Evaluation

We set up the stability evaluation in alignment with the Spark Streaming *window* operation. In Spark Streaming, data streams are received as a batch of RDDs. The number of records in a batch is determined by the batch interval. The *window* operation keeps multiplies of batch intervals to make the number of batches fit with the duration of a window. The experiment is deployed on 16 nodes of executors.

We inject the entire dataset to 30 windows into the workflow shown in **Fig. 8** (a). We consider the whole workflow system stable if the processing time of each batch of data streams is less than the batch interval. In this experiment, we set the batch interval as 10 seconds. In Fig. 8 (b) and (c), we set a horizontal line at 10 second indicating the stability threshold.

Over the span of 300 seconds (5 minutes), we observe the processing time and scheduling delay from the Spark UI in response to the input rates. If the processing time is over the threshold, it implies the system has tasks waiting in the queue and thus results in scheduling delay. From Fig. 8 (a), we can see that the input rate peaks at windows 10 to 14.

This leads to the fact that the processing time also increases in the corresponding windows (see Fig. 8 (b) and (c)). The algorithm with the P2PL distance metrics remains the processing time under 10 seconds per window across the entire experiment. With the PL2PL distance metrics, the algorithm goes over stable line at 7th window causing extra up to approximately 4-second scheduling delay. The scheduling delay declines after 13th window since the input rate begins to decrease. This indicates to further improve the stability of PL2PL to handle the peak load, the underlay cluster needs to provision extra executor nodes. The auto-scaling mechanism of Amazon Web Services can be applied to provision and deprovision worker nodes on demands that remains our future work.

6.6 Data Intensity Evaluation

In this experiment, we observe the data shuffling rate with regards to the size of data per time slot. The data shuffling rate is represented by two metrics: *shuffle read*, and *shuffle write*, measured as the ratio (%) of the input data. *Shuffle read* (or *Shuffle write*) refers to the sum of serialized *read* (*write*) data of all executors. Both of these metrics are obtained from the Spark UI utility. We tune four set of parameters, shown in **Table 4** to obtain different size of data to process per time slot. For example, when we increase the distance threshold (ϵ) and time slot (T), more point-to-polyline or polyline-to-polyline pairs meet the density reachable requirements. Thus, the algorithm generates larger number of density connections. We also decrease density threshold (μ) and size threshold (l). This increases the data density in the companion discovery phase of the algorithm.

Table 4 Parameter settings of data intensity evaluation.

Parameter Set	ϵ	k	l	n	T	μ
$S1$	0.001	3	3	8	60	3
$S2$	0.005	3	3	8	60	3
$S3$	0.005	3	3	8	100	3
$S4$	0.005	3	2	8	100	2

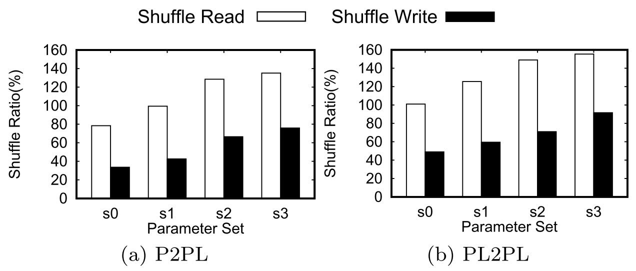
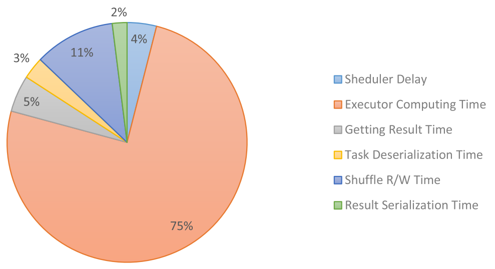


Fig. 9 Data shuffling comparison.

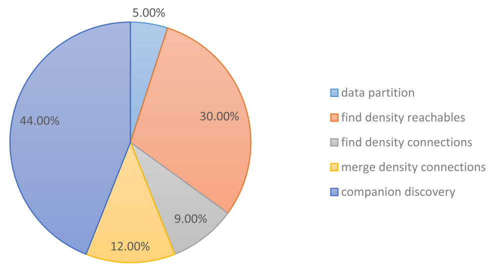
As **Fig. 9** illustrates, both read and write shuffling ratios of PL2PL is higher than P2PL. This indicates the PL2PL distance metrics has more frequent data read from and write to remote executors. The cost of data shuffling is the major contributor approximately 20% to 30% performance difference between these two metrics (see Fig. 6 in Section 6.3).

6.7 Execution Time Decomposition

We further decompose the execution time of the TCompanion algorithm with PL2PL distance metrics to understand which steps in the workflow contribute most to the time cost. **Figure 10** (a) illustrates the ratio of scheduler delay, executor computing time, getting result time, task deserialization time, shuffle read/write time, and result serialization time. First, majority of task execution time comprises of raw computation time that dominates



(a) Time distribution across Spark's metrics



(b) Time distribution across all steps in the workflow

Fig. 10 Execution time decomposition.

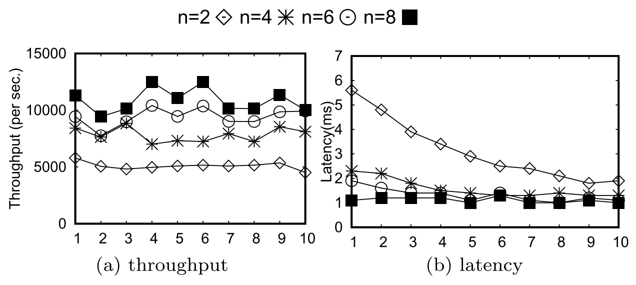


Fig. 11 Vary n from 2 to 8.

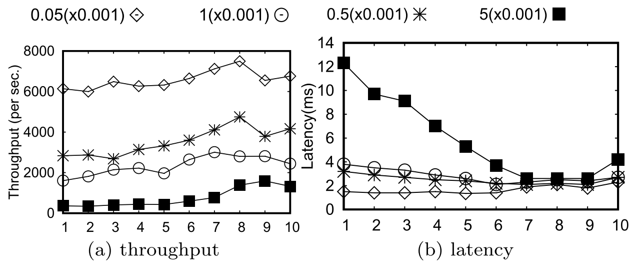


Fig. 12 Vary ϵ from 0.00005 to 0.005.

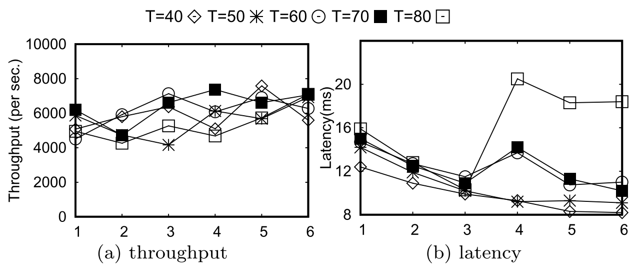


Fig. 13 Vary T from 40 to 80.

about 75% of total time. Second, data shuffle read/write time takes 11%. This indicates that although data shuffling has effects on the total time cost but it is not the main performance bottleneck tuned by our optimization techniques.

Figure 10 (b) illustrates the time distribution over all steps in the workflow. The companion discovery phase takes 44% of overall task execution time. This phase contains one transformation to generate all subsets from density connections in order to find trajectory companion (Algorithm 3 line 3). Its complexity yields $O(2^n)$.

6.8 Effect of Parameters on TCompanion

We analyze the performance of TCompanion (PL2PL) under parameter settings since the distance metrics of polyline-to-polyline produces higher precision. We run the algorithm in 10 windows with each duration of 60 s that is in total 10 minutes. We tune the distance threshold ϵ and observe its effects on throughput and latency. We vary the distance threshold ϵ from 0.00005 to 0.005, with the geospatial meaning of 10 to 100 meters. Other parameters use default settings. **Figure 12** shows the throughput is higher and the latency is lower by decreasing ϵ. One reason is lager ϵ covers more objects, thus more coverage density connections is generated. Intuitively, discovering companions from these coverage density connections takes longer time. The figure also shows during the time window 7, 8, 9, the workflow produces higher throughput and lower latency than other time windows. This indicates fewer objects from the data streams form coverage density connections and thus have less computation and data shuffling cost.

To evaluate with Spark Streaming precisely, we test the performance of our algorithms in multiple Spark Streaming windows [*4]. We first evaluate the effect of number of work nodes, n. In AWS, we deployed from 2 to 8 compute units, namely n is varied from 2 to 8, and other parameters are used default settings.

[*4] http://spark.apache.org/docs/latest/streaming-programming-guide.html

Figure 11 plots the throughput and latencies of TCompanion under different n in 10 windows (i.e., each window's size is 60 s) respectively. Clearly, the throughput is higher and the latency is smaller when we use more work nodes. It can be seen that our method's performance can be improved dramatically if we add the number of work nodes. Namely, the scalability of our method is good.

Lastly, we study the effect of time slot size T on the performance and we vary T from 40 seconds to 80 seconds. We also set window size to be same as T, and run the method in 6 windows. As illustrated in **Fig. 13** (a), the throughput are not related with varying T, because the average processing time for each location record is not related with T. However, the latency becomes high with increase of T (see Fig. 13 (b)). The reason is larger T will have more waiting time in Spark Streaming processing.

7. Discussion and Reflection

We discuss our reflection on three factors in the design and implementation of the parallelism.

(1) Data parallelism - We focus on data parallelism that partition the workload over multiple worker nodes at the data input stage and the data processing stage. During the data input stage, the data partition is performed on continuous streams that inject into the analysis workflow by a time-window. Therefore, the partition method is not performed one time only but rather iteratively as the data streams ar-

rive at a time interval. Our technique to balance continuous streaming load is at two levels. First, at the algorithm level, we apply the KD-tree partition algorithm that partitions of trajectories are balanced across worker nodes. Second, at the API level, we replace the default implementation of Spark to assure data locality that data records from the same partition are not spread on different worker nodes. At the data processing stage, we examine various data aggregation methods quantitatively and select the most efficient one to further reduce the size and frequency of data shuffling. As a result, our method assures data shuffling is not the bottleneck of throughput. This is validated by the execution time decomposition.

(2) Algorithm composition - The discovery method is composed by steps of analysis. Each step consists of functions such as using inverted index and range search to detect a *crow* of moving object in the *gathering* algorithm. Certainly the detection can have alternative ways for searching and detecting a crow. Therefore each way performs differently in terms of computing intensity and data shuffling. Our design of the workflow is modular that decouples the functions in each step of the analysis method. Hence functions and even metrics can be changed or replaced in the workflow to select the best suitable one in terms of accuracy and performance. For example, we use point to polyline and polyline to polyline.

(3) Ensemble experiments - The benefit of the algorithm composition is it allows an *ensemble* approach to run experiments with the same input and system level setup. The details of experiments differ from one to one because of the function selection, and choice of distance metrics. Each experiment is a parallel computing pipeline and the discovery analysis can run these pipeline concurrently. This remains our future work to fully automate the ensemble experiment deployment and launching.

8. Conclusion

In this paper, we devised a parallel discovery method called Trajectory Companions on heterogeneous trajectory data stream. The parallelism focuses on data partition and data aggregation to improve data locality and hence reduce the data shuffling overhead of the discovery framework. Our discovery algorithm contains both spatial and temporal functions to measure distances between trajectories over continuously updated streaming windows. In experiments, our method is able to process up to 30,000 updates per second of moving objects within 14 seconds. The modular structure of our analysis framework allows other distance metrics and clustering methods to be applied. It remains our future work to refractor the current method as an algorithm of service on the cloud.

distance metrics, load-balancing strategy. To optimize the framework, we analyzed merging method and proposed IJHP to minimize data shuffling. Experimental results demonstrated that the proposed method has better accuracy and competitive runtime performance to Gathering. Although the distance metrics of PL2PL produces higher precision than P2PL, it also results in higher latency due to more frequent data read and write remotely. In a nutshell, our proposed framework can keep up the grow of data stream with a few seconds delay, which achieves nearly real time streaming data processing.

References

[1] Xian, Y., Xu, C., Elnikety, S. and Liu, Y.: Parallel Discovery of Trajectory Companions from Heterogeneous Streaming Data, *2019 IEEE 43rd Annual Computer Software and Applications Conference* (*COMPSAC*), pp.453–462 (2019).

[2] Becker, B., Gschwind, S., Ohler, T., Seeger, B. and Widmayer, P.: An asymptotically optimal multiversion b-tree, *The VLDB Journal*, Vol.5, No.4, pp.264–275 (Dec. 1996).

[3] Eldawy, A. and Mokbel, M.F.: SpatialHadoop: A MapReduce Framework for Spatial Data, *31st ICDE 2015*, pp.1352–1363 (2015).

[4] Foley, T. and Sugerman, J.: Kd-tree acceleration structures for a gpu raytracer, *Proc. ACM SIGGRAPH/EUROGRAPHICS*, pp.15–22 (2005).

[5] Gudmundsson, J. and van Kreveld, M.: Computing longest duration flocks in trajectory data, *Proc. GIS*, pp.35–42 (2006).

[6] Guttman, A.: R-trees: A dynamic index structure for spatial searching, *Proc. SIGMOD*, pp.47–57 (1984).

[7] Hadjieleftheriou, M., Kollios, G., Tsotras, V.J. and Gunopulos, D.: Efficient indexing of spatiotemporal objects, *Proc. 8th International Conference on Extending Database Technology: Advances in Database Technology, EDBT '02*, pp.251–268, Springer-Verlag (2002).

[8] Jeung, H., Yiu, M.L., Zhou, X., Jensen, C.S. and Shen, H.T.: Discovery of convoys in trajectory databases, *Proc. 2010 IEEE International Conference on Data Mining Workshops*, pp.170–177 (2010).

[9] Karau, H., Konwinski, A., Wendell, P. and Zaharia, M.: *Learning Spark: Lightning-Fast Big Data Analytics*, O'Reilly Media, Inc., 1st edition (2015).

[10] Leibiusky, J., Eisbruch, G. and Simonassi, D.: *Getting Started with Storm*, O'Reilly Media, Inc. (2012).

[11] Li, Z., Ding, B., Han, J. and Kays, R.: Swarm: Mining relaxed temporal moving object clusters, *Proc. VLDB Endow.*, Vol.3, No.1-2, pp.723–734 (Sep. 2010).

[12] Magdy, A., Mokbel, M.F., Elnikety, S., Nath, S. and He, Y.: Mercury: A memory-constrained spatio-temporal real-time search on microblogs, *ICDE*, pp.172–183 (2014).

[13] Miller, J., Raymond, M., Archer, J., Adem, S., Hansel, L., Konda, S., Luti, M., Zhao, Y., Teredesai, A. and Ali, M.: An extensibility approach for spatio-temporal stream processing using microsoft streaminsight, *Proc. SSTD*, pp.496–501 (2011).

[14] Neumeyer, L., Robbins, B., Nair, A. and Kesari, A.: S4: Distributed stream computing platform, *Proc. 2010 IEEE International Conference on Data Mining Workshops*, pp.170–177 (2010).

[15] Rabkin, A., Arye, M., Sen, S., Pai, V.S. and Freedman, M.J.: Aggregation and degradation in jetstream: Streaming analytics in the wide area, *Proc. USENIX*, pp.275–288 (2014).

[16] Samet, H.: The quadtree and related hierarchical data structures, *ACM Comput. Surv.*, Vol.16, No.2, pp.187–260 (June 1984).

[17] Condie, T. and Conway, N., et al.: Mapreduce online, *Proc. NSDI*, pp.313–328 (2010).

[18] Tang, L.-A., Zheng, Y., Xie, X., Yuan, J., Yu, X. and Han, J.: Retrieving k-nearest neighboring trajectories by a set of point locations, *Proc. 12th International Conference on Advances in Spatial and Temporal Databases, SSTD'11*, pp.223–241 (2011).

[19] Tang, L.-A., Zheng, Y., Yuan, J., Han, J., Leung, A., Hung, C.-C. and Peng, W.-C.: On discovery of traveling companions from streaming trajectories, *ICDE 2012* (Apr. 2012).

[20] Tang, L.-A., Zheng, Y., Yuan, J., Han, J., Leung, A., Peng, W.-C. and Porta, T.L.: A framework of traveling companion discovery on trajectory data streams, *ACM Trans. Intell. Syst. Technol.*, Vol.5, No.1, pp.3:1–3:34 (Jan. 2014).

[21] Šaltenis, S., Jensen, C.S., Leutenegger, S.T. and Lopez, M.A.: Indexing the positions of continuously moving objects, *Proc. SIGMOD*, pp.331–342 (2000).

[22] Xian, Y., Liu, Y. and Xu, C.: Parallel gathering discovery over big trajectory data, *2016 IEEE International Conference on Big Data* (*Big Data*), pp.783–792 (Dec. 2016).

[23] Xian, Y., Xu, C. and Liu, Y.: Implementing trajectory data stream analysis in parallel, *2016 IEEE International Conference on Big Data* (*Big Data*), pp.2431–2436 (Dec. 2016).

[24] Xu, W., Juri, N.R., Gupta, A., Deering, A., Bhat, C., Kuhr, J. and Archer, J.: Supporting large scale connected vehicle data analysis using HIVE, *2016 IEEE International Conference on Big Data*, pp.2296–2304, (2016).

[25] Yuan, J., Zheng, Y., Zhang, C., Xie, W., Xie, X. and Huang, Y.: T-drive: Driving directions based on taxi trajectories, *ACM SIGSPATIAL GIS 2010*, Association for Computing Machinery, Inc. (Nov. 2010).
[26] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S. and Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, *Proc. USENIX*, p.2 (2012).
[27] Zheng, K., Zheng, Y., Yuan, N.J. and Shang, S.: On discovery of gathering patterns from trajectories, *IEEE International Conference on Data Engineering* (*ICDE 2013*), IEEE (Apr. 2013).
[28] Zheng, K., Zheng, Y., Yuan, N.J., Shang, S. and Zhou, X.: Online discovery of gathering patterns over trajectories, *IEEE Trans. Knowledge Discovery and Data Engineering* (2014).
[29] Zheng, Y., Zhang, L., Xie, X. and Ma, W.-Y.: Mining interesting locations and travel sequences from gps trajectories, *Proc. World Wide Web*, pp.791–800 (2009).

Yong Yi Xian received his M.A.Sc. (2018) and B.Eng. (2012) degrees from Concordia University, Montreal, all in computer engineering. In 2011, he started his career in Presagis, Montreal, Canada, and gained experience in GIS and aerospace modeling and simulation industry. He later joined Bloomberg LP in 2018 as a senior engineer and has been involved in building cloud-based, large scale, and natural language processing enabled data processing and reporting systems for information governance, surveillance, and trade reconstruction. His areas of specialty and interest include big data analytics, distributed computing systems, and natural language processing.

Yan Liu is an Associate Professor in Faculty of Engineering and Computer Science, Concordia University. She has over 15 years research experience of developing data intensive algorithms on distributed and parallel computing systems. Before her faculty position, she was a senior research scientist in Pacific Northwest National Laboratory (PNNL) in Washington State, delivering high performance and scalable data analysis platforms for domains of power systems, scientific computing and engineering simulation. Her recent research focuses on parallel and distributed machine learning, and automatically scaling back-end computing resources also by means of machine learning.

Chuanfei Xu received his B.E. degree in 2007 from Shenyang University of Technology, his M.E. degree and Ph.D. degree in 2009 and 2013 from Computer Science, Northeastern University, China. His research interests include spatial database management, uncertain data management and natural language processing (NLP).

Sameh Elnikety is a systems researcher at Microsoft, focusing on experimental server systems. His research interests span several areas including distributed computing, databases and operating systems. His main research theme is making large cloud services more efficient, responsive and reliable.

Elie Neghawi is a Ph.D. Student in Faculty of Engineering and Computer Science at Concordia University and a Senior IT Technology Consultant at SAP, focusing his research interests on distributed computing and machine learning algorithms. His main research theme is distributed machine learning efficient and reliable.