# Streaming Data Algorithm Design for Big Trajectory Data Analysis

**Yong Yi Xian**

**A Thesis**

**in**

**The Department**

**of**

**Electrical and Computer Engineering**

**Presented in Partial Fulfillment of the Requirements**

**for the Degree of**

**Master of Applied Science (MASc) at**

**Concordia University**

**Montréal, Québec, Canada**

**June 2017**

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Yong Yi Xian**

Entitled: **Streaming Data Algorithm Design for Big Trajectory Data Analysis**

and submitted in partial fulfillment of the requirements for the degree of

**Master of Applied Science (MASc)**

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
*Dr. Rabin Raut*

_____ External Examiner
*Dr. Lingyu Wang*

_____ Examiner
*Dr. Nawwaf Kharma*

_____ Supervisor
*Dr. Yan Liu*

Approved by _____
William E. Lynch, Chair
Department of Electrical and Computer Engineering

_____ 2017

_____
Amir Asif, Dean
Faculty of Engineering and Computer Science

# Abstract

Streaming Data Algorithm Design for Big Trajectory Data Analysis

Yong Yi Xian

Trajectory streams consist of large volumes of time-stamped spatial data that are constantly generated from diverse and geographically distributed sources. Discovery of traveling patterns on trajectory streams such as gathering and companies needs to process each record when it arrives and correlates across multiple records near real-time. Thus techniques for handling high-speed trajectory streams should scale on distributed cluster computing. The main issues encapsulate three aspects, namely a data model to represent the continuous trajectory data, the parallelism of a discovery algorithm, and end-to-end performance improvement. In this thesis, I propose two parallel discovery methods, namely snapshot model and slot model that each consists of 1) a model of partitioning trajectories sampled on different time intervals; 2) definition on distance measurements of trajectories; and 3) a parallel discovery algorithm. I develop these methods in a stream processing workflow. I evaluate our solution with a public dataset on Amazon Web Services (AWS) cloud cluster. From parallelization point of view, I investigate system performance, scalability, stability and pinpoint principle operations that contribute most to the run-time cost of computation and data shuffling. I improve data locality with fine-tuned data partition and data aggregation techniques. I observe that both models can scale on a cluster of nodes as the intensity of trajectory data streams grows. Generally, snapshot model has higher throughput thus lower latency, while slot model produce more accurate trajectory discovery.

# Acknowledgments

I would like to express my sincere appreciation to my supervisor Prof. Yan Liu, for her constant guidance and encouragement, without which this work would not have been possible. The enthusiasm and immense knowledge she has for her research was contagious and motivational for me. She chose a cutting-edge research topic that I am interested in. Her timely advice, meticulous scrutiny, scholarly advice, scientific approach and financial approach have assisted me to a great extent to accomplish this work. Besides the research, Prof. Liu also concerns her student's career paths. She gave me many useful advices and recommendations. I could not have imagined having a better advisor and academic experience.

I would also like to thank Chuanfei Xu, the co-author of two of our papers published in IEEE International Conference on Big Data, for sharing his insights, as well as significant amount of contribution to the papers. It is my pleasure to work with him and learn from him. His great work keeps me up the momentum of my research.

It has been hard work with time conflicts between work-life, study-life and personal life, although I enjoyed every bit of my research. I must express my very profound gratitude to my family for providing me with unfailing support throughout my years of study and through the process of researching and writing this thesis.

# Contents

# List of Figures

# List of Tables

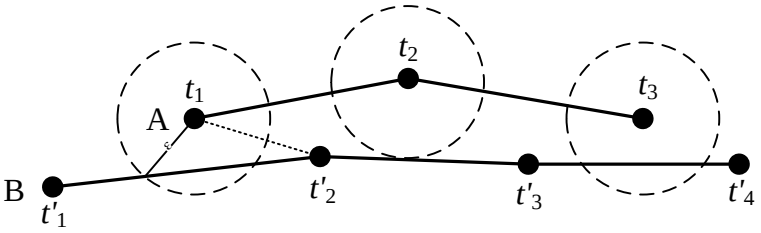
# Chapter 1

# Introduction

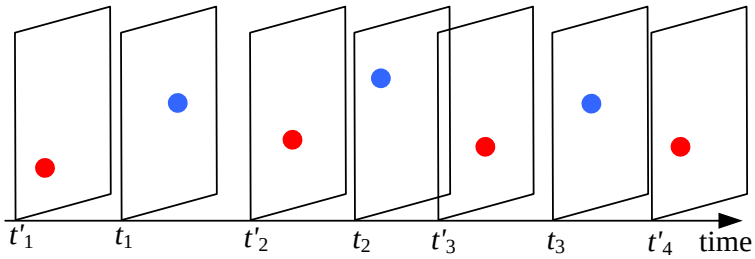
Advances in location-acquisition technologies such as GPS positioning, sensors probing, mobile phones monitoring and many smart devices. These spatial-temporal location data are usually recorded in the format of streaming trajectories [29, 37]. Spatial-temporal trajectory data are constantly generated and collected from tremendous diverse sources. In this thesis, I term trajectory data streams collected from heterogeneous sources as *heterogeneous streaming data*. For example, the Microsoft Geolife project [42] 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 [15, 20, 30, 31, 40, 41]. In these works, snapshots were used to model trajectory data. For trajectories each sampled at different time intervals, a snapshot model with a fixed interval is likely to miss data points that are partial to companions in continuous time series.

A simple case is illustrated in the following example of Figure 1.1. 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 [36] explores the use of real world connected vehicle data set called Safety Pilot Model Delolyment (SPMD) data. The study was




(a) Real trajectories




(b) Snapshot model

Figure 1.1: Example of trajectories

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 Figure 1.1(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 $\epsilon$, if the distance between any two collected locations of these two objects is not more than $\epsilon$, I think these vehicles travel together at this time. In Figure 1.1(a), I observe the distances from any collected location of A to the trajectory of B are always less than $\epsilon$. Therefore, they can be regarded as companions. It should discover them from trajectory data. However, based on the snapshot model (see Figure 1.1(b)), I cannot collect both a location of A and a location of B in the same snapshot. Therefore, trajectory companions is non-deterministic in this case.

Current techniques have two limitations. First, the snapshot modeling methods [15, 20, 30, 31] 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.

Trajectories are independent to each other. Intuitively, operations on individual trajectories or combination of trajectories can be processed in parallel. The parallel computing paradigm MapReduce [7] and its open-source implementation such as Hadoop provides off-the-shelf batch processing for big trajectory data (such as SpatialHadoop [9]). However, Hadoop runs batch mode processing and relies on distributed file system or storage to keep intermediate data. Thus it is not directly applicable for streaming data with fast updates.

I desire a new approach that is able to model the relations of trajectory objects at different timestamps within a time period, and thus reduce the missing rate of trajectory streams. A solution to address the aforementioned problems should discover relations of objects at each snapshot as well as relations to all the objects at adjacent timestamps. Since the size of objects in each trajectory are constantly accumulating in a continuous fashion, this leads to big volume of data when the number of trajectories grow from geographically distributed sources. Hence it is promising to parallel the discovery process on a cloud platform that scales beyond a centralized computing solution. In particular, there are three issues I need to address in parallelism: i) design a discovery method of handling heterogeneous streaming data precisely; ii) parallelize the discovery algorithm on a cloud platform to scale; iii) reduce end-to-end delay by improving data locality.

In this thesis, I devise two traveling group models namely *gathering* and *Trajectory Companion (TCompanion)*. For each model, I will present the algorithms, parallel workflow, optimization techniques, and evaluate the proposed methods and techniques on a public dataset in Amazon EC2 clusters.

In particular, I first present *gathering* that (1) defines a *snapshot clustering* model (2) detects clusterings of moving objects into *crowds* during a certain time intervals that satisfy thresholds on distances and densities; (3) discovers *gatherings* that the participation of moving objects in crowds are constant over a time threshold.

Next, I propose the *TCompanion* model that (1) represents trajectory objects' locations at each timestamp as well as their movements in a time period. (2) define distance metrics to measure trajectories for their companion discovery over time; (3) develop discovery algorithms based on the distance metrics and build a parallel framework and optimization algorithms to cope with data locality and load balancing.

In details, I summarize our technical contributions in this paper as follows.

- Devise traveling pattern called Gathering on both batch and streaming data, that discovers the participation of moving objects in crowds are constant over time
- Devise traveling groups termed Trajectory Companions (TCompanion) on heterogeneous streaming data. Compared to Gathering, TCompanion represents the heterogeneity of data with accuracy.
- Design parallel workflow for both models with a suite of techniques and effective load-balancing and aggregation strategies.
- I propose optimization algorithms in our parallel frameworks to improve end-to-end performance. From our evaluation, TCompanion can discover more accurate traveling companions, while Gathering have higher throughput and lower latency.

The rest of the thesis is organized as follows. Chapter 2 introduces the concepts and preliminaries in the thesis. Chapter 3 provides the related work from both conceptual and technological aspects. Chapter 4 states our research methodology including problem statement, high level system architecture, design factors, as well as evaluation method. Chapter 5 and Chapter 6 cover the *gathering* and *TCompanion* models in detail respectively. Chapter 7 concludes the thesis.

# Chapter 2

# Background

## 2.1 Distributed Computing

A distribution system refers to a network of autonomous nodes (physical machines or virtual machines) that communicate with each other in order to achieve one goal. The nodes in a distributed system are independent and do not physically share memory or processors. Information is exchanged by passing messages between the processors. Distributed computing is widely used because its advantages over traditional centralized computing. By using the combined processing and storage capacity of many nodes, performance levels can be reached that are out of the scope of centralized computing. That is, distributed computing makes it possible to solve big data problems more efficient with a cluster of processing units.

When it comes to designing a distributed environment, three core systemic requirements exist in a special relationship: *Consistency*, *Availability* and *Partition-tolerance*. This concept is so-called the *CAP theorem*, was first introduced by Eric Brewer[4]. Basically, the theorem states that can satisfy only two of these guarantees at the same time, but not all three. *Consistency* means that all nodes see the same data at the same time. *Availability* guarantees that every request receive its response. *Partition Tolerance* implies that system continues to work despite message loss or partial failure. These guarantees are essential to the distributed systems.[5]

## 2.2 MapReduce Model

MapReduce [6] is a programming model for data intensive applications, providing an abstraction which hides all complexity of parallelization. The term *MapReduce* is first used by Jeffrey Dean and Sanjay Ghemawat in Google. Many real world tasks are expressible in this model. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machine. In fact, MapReduce has been efficiently solved a wide range of large-scale computing problems, including risk assessment, recommendation engine, document clustering, machine learning and so on.

In MapReduce paradigm, data are represented as key/value pairs. A job in MapReduce contains three main phases: map, shuffle and reduce. the whole process can be summarized as following:

**Map** $< k_1, v_1 > \longrightarrow list < k_2, v_2 >$

**Reduce** $< k_2, list\{vg\} > \longrightarrow list < k_3, v_3 >$

In the map phase, for each input pair $< k_1, v_1 >$, the map function produces one or more output pair list $< k_2, v_2 >$. In the shuffle phase, these data tuples are ordered and distributed to reducers by their hashed keys. In the reduce phase, pairs with the same key are grouped together as $< k_2, list\{vg\} >$. Then the reduce function generates the final output pairs list $< k_3, v_3 >$ for each group.

## 2.3 Stream Processing Model

Matei Z. *et al* proposed a stream programming model, namely *discretized stream (D-Stream)* [39]. The main idea behind D-Streams is to treat a streaming computation as a sequence of *micro-batch* computations on small time intervals. The input data received during each interval is stored across the cluster to form an input dataset for that interval. Once the time interval completes, this dataset is processed via parallel operations to produce new datasets representing outputs or intermediate state.

*Sliding Window* [2, 21, 39] is one of the common stream processing techniques that works over multiple intervals. The idea is to evaluate not over the entire past stream events, but rather only over sliding windows of recent data from the streams. For example, only data from the past hour could be considered in producing results. Data older than that will be discarded.

In stream processing, it often needs to track data across batches so called a state. While sliding windows compute the results over multiple batches of data, each window would not know the result from previous windows and hence is stateless. Many complex stream processing pipelines must maintain state across a period of time. For example, user behavior analysis for websites requires to maintain information about each user session as a persistent state and continuously update this state based on the user's actions.

Figure 2.1 illustrates micro-batch, sliding window, and stateful result with a streaming word count example. There are 5 events arriving at different timestamps along with input stream. Our goal is to count the word occurrences from the input event stream. I first consider dividing the input stream into micro-batches where each batch interval is 5 minutes. I can obtain the running word counts with one single map/reduce job. Instead of running word counts, I count words within 10 minute windows sliding every 5 mins. That is, word counts in words received between 10 minute windows are 12:00-12:10, 12:05-12:15 and so on. Note that 12:00-12:10 means data that arrived after 12:00 but before 12:10. Finally, a global state can keep track of a state for each word count and update at the end of each batch.

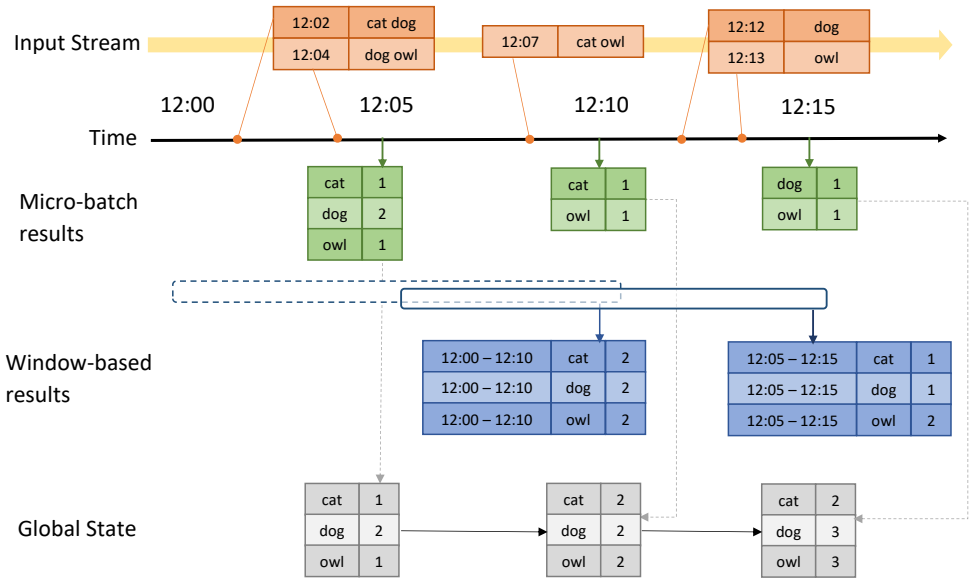



Figure 2.1: Streaming Word Count Example

# Chapter 3

# Related Work

## 3.1 Discovery of Traveling Groups

There exist several concepts with the aim to discover a group of objects that move together for a certain time period, such as *flock* [12], *convoy* [15], *swarm* [20], *companion* [30, 31], and *Gathering* [40, 41]. These concepts can be distinguished based on how the 'group' is defined and whether they require the time period to be consecutive. For instance, *companion* groups objects in the density-based clustering fashion and it requires a group of objects to be density-connected to each other during a consecutive time period. Next I summarize some characteristics of these methods in Table 3.1.

Table 3.1: Different Group Patterns

| | *flock* | *convoy* | *swarm* | *companion* | *Gathering* | the proposed |
|---|---|---|---|---|---|---|
| flexible group patten | × | √ | √ | √ | √ | √ |
| flexible consecutive time | × | × | √ | × | √ | √ |
| stream processing | × | × | × | √ | √ | √ |
| flexible lifetime | × | × | × | × | √ | √ |
| parallelism | × | × | × | × | √ | √ |
| heterogeneous patten | × | × | × | × | × | √ |

In Table 3.1, the characteristics are explained below:

- *flexible group patten* denotes that this method captures the pattern of any shape (i.e., not only circular shape) since it adopts the density-based clustering;

- *flexible consecutive time* denotes a cluster of objects lasting for consecutive timestamps;

- *stream processing* denotes that this method can cope with dynamic trajectory data;

- *flexible lifetime* denotes that in a cluster members joining and leaving is inevitable;

- *parallelism* denotes the method can parallel processing trajectory data;

- *heterogeneous patten* means that the method can discover companions or patterns from heterogeneous trajectory data.

These characteristics are very significant for discovery algorithms of traveling groups or companions. For example, an algorithms without *flexible group patten* may miss lots of traveling groups whose location distributions are not regular. In business promotion, a group member may leave at a timestamp and be back after a period. But the critical part is, though each individual may only leave for a while, and rejoin in the same group. In this case, if an individual leaves and not joins other group, I consider this individual is also in the original group in a time period. To the best of our knowledge, there is no existing method that can capture more generic traveling groups or companions (i.e., satisfy all these characteristics). Therefore, I aim to propose a solution which can discover more generic traveling groups while ensure it is able to process heterogeneous streaming data as well. To our best knowledge, *Gathering* approach is the current state-of-the-art method. This is, I will first adjust *Gathering* in parallel processing manner, and compare TCompanion method with it.

## 3.2 Trajectory Segmentation

In many scenarios, such as trajectories clustering, I need to divide a trajectory into segments for a further process. The segmentation not only reduces the computational complexity but also enables us to mine richer knowledge. I study different segmentation techniques as follow,

- *Equal-Split*. This technique produces MBRs of fixed time interval of length $l$. It is a simple approach with cost linear in the length of a sequence. However, the length of MBR is critical as the increase of splits can result to larger space utilization.

- *k-Optimal.* I can discover the k MBRs of a sequence that take up the least volume, using a dynamic programming algorithm that requires $O(n^2k)$ time ([14]), where $n$ is the length of the given sequence. Since this approach is not reasonable for large databases, I did not consider k-Optimal.

- *Greedy-Split*. I assign an MBR to each of the n sequence points and at each subsequent step

I merge the consecutive MBRs that will introduce the least volume consumption. The algorithm has a running time of $O(nlogn)$. Instead of assigning to our space requirements I can assign a total of K splits to be distributed among all objects. This method can provide better results, since I can assign more splits for the objects that will yield more space gain. Also, this approach is more appropriate when one is dealing with sequences of different lengths. The complexity of this approach is $O(K + NlogN)$, for a total of $N$ objects.

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 6.2. For simplicity, I apply Equal-Split in our case.

## 3.3 Geospatial Databases

There exists databases that offer spatial data types (SDTs) in its data model and optimized to store and query data. *Spatial indexing* and *spatial join* are the most common features in these spatial databases, which can be potentially utilized in our system. I study different spatial databases as follow,

SharkDB [34] is a trajectory storing and processing system, which includes three components, the user interface component, the in-memory storage component and the trajectory query processing component. Moreover, SharkDB can make better use of main memory as the permanent storage medium, and also greatly benefits from convenient data compression and parallel processing [33]. Nevertheless, SharkDB stores massive historical trajectories to support processing, so that it cannot apply to high-speed streaming data.

PostGIS[1] is a spatial extension for PostgreSQL relational database system that allows GIS objects to be stored in the database. PostGIS comprises functions for basic analysis of GIS objects and more importantly, it also supports the spatial indexing schemes by adopting B-Tree, R-Tree, and GiST (Generalized Search Tree) indexes. PostGIS is frequently used during analysis of large data set if examination of spatial indexes is a particulary essential task. However, PostGIS, like the other

[1] http://www.postgis.net/

relation databases, the database is difficult to scale out. Thus, PostGIS is not suitable for distributed systems.

Neo4j[2], NoSQL graph based database, supports the storage and manipulation of spatial data through the library Neo4j Spatial[3]. The Neo4j Spatial enables spatial operations on data store. For instance, operations to locate data within a specific region or an area close to point of interest (POI). While Neo4j Spatial is easier to scale out compared to PostGIS, it only supports two dimensional geometry data. Moreover, Neo4j has no notion of Data/Time data type, timestamp can be stored as a long, or as a human-readable String. In our case, I aim to analyze spatial-temporal trajectory data which can be considered as three dimensional data. That is, storing such data in Neo4j may either miss information or require explicit management.

## 3.4 Parallel Platform for Trajectory Data

To solve our problem efficiently, I desire parallel processing platforms. There is lots of high-performance platforms [26, 23, 24, 25, 38] devoting to processing data streams. But, some of them could be used to analyze trajectory data and trajectories mining. I now introduce some systems and parallel platforms that are suitable for processing this type of data.

First, MapReduce Online [28] pipelines the intermediate data between Map and Reduce operators. Compared to traditional MapReduce where a reducer reads the data from mappers in a pull fashion, a mapper in MapReduce Online transmits data to reducers in a push fashion. When a mapper has finished processing a key-value pair, it directly sends the data to the reducers through socket connections. Therefore, the reducers in MapReduce Online do not have to wait until the last map task has finished. However, MapReduce Online lacks the ability to cache data between iterations. For our trajectory problem, I need massive iterative operations, so that MapReduce Online cannot not satisfy our high-speed processing requirements.

Second, Dremel [23] is a system proposed by Google for interactive process of large-scale data sets. It complements MapReduce by providing much faster query processing. Dremel combines a nested data model with columnar storage to improve retrieval efficiency. To achieve this goal,

[2]https://neo4j.com/

[3]https://github.com/neo4j-contrib/spatial

Dremel introduces a lossless representation of record structure in columnar format, provides fast encoding and record assembly algorithms, and postpones record assembly by directly processing data in columnar format. However, it cannot work so well for our complex trajectory analysis tasks.

Furthermore, Storm is an open source low latency data stream processing system [19]. Storm consists of several moving parts, including the coordinator (ZooKeeper), state manager (Nimbus), and processing nodes (Supervisor). Storm implements the data flow model in which data flows continuously through a network of transformation entities. Storm runs in-memory, and is therefore set to process large volumes of data at in-memory speed. In many enterprise applications, Storm is used as their real-time architecture for integrating and processing streaming data continuously. But it does not have the load balancing policy to balance processing time in different machines.

I also introduce S4 that is a real-time and distributed modular platform for processing continuous unbounded streaming data, which has a decentralized and symmetric architecture which all the nodes in a cluster are identical, different to the classic master-nodes architecture [25]. However, compared with S4, the reliability and performance of Storm is higher [25]. Therefore, I do not use S4 platform to process our trajectory data streams.

Next I analyze one of most popular parallel computing platform Spark [16] and Spark Streaming [38]. Spark is a cluster computing system 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 [38]. Moreover, Spark can work on the RDDs for multiple iterations which are required by many machine learning algorithms. Spark Streaming is extended from Spark by adding the ability to perform online processing through a similar functional interface to Spark, such as map, filter, reduce, etc [38]. 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.

In our problem, I need to deal with massive and continuously updating trajectory data clustering operations. Therefore, the following characterizes are necessary,

- *real-time processing* the platform processes trajectory data in a real-time fashion;

- *iterative processing* the platform can execute iterative operations efficiently;

- *recoverability* recoverability from failures can guarantee the platform processes trajectory streams stably;

- *functionality* the platform is multi-functional, such as data mining, learning, and query processing, and so on;

- *trajectory storage* the platform stores trajectory information of objects and partitions them, which can process data in a parallel manner.

Table 3.4 summarizes the parallel platforms based on these characteristics. I can see that Storm and Spark are the most appropriate toward our objectives.

Table 3.2: Comparison of Different Approaches

| | MapReduce Online | Storm | SharkDB | Dremel | Spark |
|---|---|---|---|---|---|
| real-time processing | | ♠ | | ♠ | ♠ |
| iterative processing | ♠ | ♢ | | | ♢ |
| recoverability | ♢ | ♢ | ♢ | ♢ | ♢ |
| functionality | ♢ | ♠ | | ♢ | ♠ |
| trajectory storage | | | ♠ | | |

(♠ indicates primary objective, while ♢ indicates secondary objectives)

According to the comparison of experimental results[4], Spark Streaming can handle more data than Storm in the same time period. Namely, the Spark Streaming system has higher throughput. Therefore, I choose Spark Streaming as the parallel computing platform to solve our problem.

## 3.5 Batch-based Spatial Data Processing

Trajectories consist of spatial points or location, so I analyze papers on spatial data processing. There is a large body of research work on spatial indexes for spatial data, such as R-tree [13], multi-version B-tree [3], quad-tree [27] and so on. R-tree is one of most popular spatial indexes, and especially it is very effective for multi-dimensional data. The R-tree index height-balanced index structure. Objects are represented by minimum bounding rectangles (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 [13]. For trajectory data, Saltenis et al. [32] propose an extension version of R-tree termed

[4]http://www.slideshare.net/ptgoetz/apache-storm-vs-spark-streaming

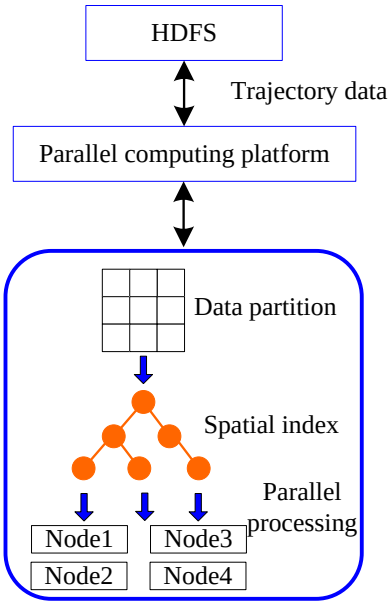



Figure 3.1: Batch-based processing architecture

Time-Parameterized R-tree (TPR-tree) which augments the R-tree with velocities to index moving objects. Another often used index structure is the quadtree. Samet [27] has done a thorough survey of the quadtree and the related hierarchical data structures. On the other hand, Ahmed Eldawy et al. propose a MapReduce framework for spatial data called SpatialHadoop [9] which 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. Generally, the parallel architecture of batch-based trajectory data processing includes 3 key steps: i) data partition, ii) spatial index, and iii) parallel processing (see Figure 3.1). First, I choose a partition approach to divide all trajectory data into some batches. Second, I build a spatial index (e.g., R-tree, quadtree and so on) for these batches of data to improve data search efficiency. Third, I use multiple nodes to process the data in a parallelism manner. All the trajectory data are stored in HDFS or other distributed file systems. Spark driver partitions these data into batches, and use RDD objects to represent data objects. Then master node allocates tasks to work nodes, and these work nodes process the tasks assigned to themselves.

These above batch-based techniques need to archive all historical trajectory data in order to construct spatial indexes. In our streaming problem, I can only buffer some trajectory data in a time

window (several seconds), but unable to store all data from our high-speed trajectory streams. Due to the dynamic nature of streaming trajectories (i.e., the objects' positions are always changing), maintaining spatial indexes (such as R-tree or quad-tree) at each time window incurs high cost [18]. Thus, I do not build the spatial indexes for our streaming problem.

# Chapter 4

# Research Methodology

This chapter provides an overview of our research methodology. I first address the problems I aim to solve. Secondly, I describe the high level system architecture. Then I present a summary of parallel design factors. Finally, I state our principles for evaluating the quality attributes of the our system.

## 4.1 Problem Statements

The objective of the thesis is to design scalable analytic pipelines that discover group of objects moving together pattern over large and continuous trajectory data stream. The design focuses on addressing the following research questions,

(1) *What are the appropriate model of trajectory pattern discovery?*

Choosing an appropriate model becomes the first challenge in our research. Having investigated several existing moving together discovery concepts summarized in Table 3.1, I realize that there is "no one size fit all" model for trajectory pattern discovery. While most existing concepts are either limited by the object grouping pattern or lacking the consecutive timestamp requirement, they were also designed and evaluated in conventional sequential computing fashion. Moreover, Migration of existing concepts to parallel computation is not a trivial effort. Thus, I aim to propose two solutions, namely *Gathering* and *Trajectory Companion*

(TCompanion) which can discover a group of object moving together with *flexible group pattern*, *consecutive time* while ensuring it is able to process *heterogenous* data stream. I will describe Gathering model in Chapter 5 and TCompanion model in Chapter 6.

(2) *What are the algorithmic approaches enabling parallel processing of trajectory data?*

To keep up the pace of incoming trajectory data stream, I adopts a "divide-and-conquer" paradigm to process the trajectory data concurrently. It implies that data is splitted into several smaller subset of data and processing each subset in parallel. Parallel computing has introduced new degrees of freedom to algorithm design approaches. Essentially, *Data partition* is the key enabler for parallelization in the *divide* phase. In this thesis, I focus on discussing data partitioning in terms of algorithm design approaches.

(3) *What are the distance metrics for comparing trajectories?*

A fundamental ingredient of trajectory analysis tasks is the distance/similarity measure that can effectively determine the similarity of trajectories. Unlike other simple data types such as geometric points where the distance definition is straightforward, the distance between trajectories needs to be carefully defined in order to reflect the true underlying similarity. This is due to the fact that trajectories are essentially data attached with both spatial and temporal attributes, which needs to be considered for similarity measures.

(4) *What are the parallelization design factors for efficient analysis?*

I adopt MapReduce programming model for parallel computing. However, MapReduce programs are not guaranteed to be fast. In tuning performance of MapReduce, the algorithmic and non-algorithmic design factors for mapping, shuffle, and reducing steps has to be taken into account. Factors central to this thesis are load balancing and data locality.

## 4.2 System Architecture

Figure 4.1 illustrates the high level architecture of our system which contains three layers: data ingestion, processing, data storage layers.

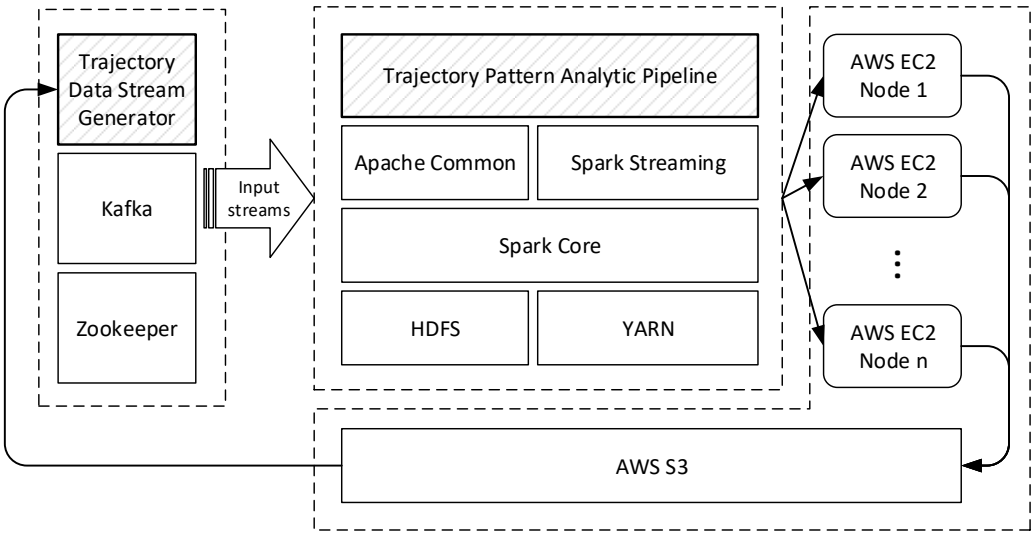



Figure 4.1: System Architecture

*Data ingestion layer*: Input data stream for Trajectory pattern discovery is generated by the trajectory Data Stream Generator. The generator is responsible for reading dataset files on a distributed file system (e.g. HDFS or AWS S3) and delivering this data in a manner simulating its arrival in real-time. The generator runs on a Kafka cluster that employs Zookeeper as the communication layer to coordinate the nodes within the cluster. On the trajectory data stream generator side, I implement a Kafka producer to push input data into a Kafka topic. This would allow consumer on the processing layer to pull the data off a topic.

*Processing layer*: Processing layer is key component of the overall architecture because this is where the algorithms and logics are implemented. The trajectory pattern analytic pipeline is built on top of Apache Spark module stack. Our system mainly requires Spark Core and Spark Streaming. As the name suggests, Spark Core is the heart of Spark and is responsible for management functions such as task scheduling. Spark Streaming ensures applications written for batches of historical data can be repurposed to analyze streaming data with little modification. The spark program is deployed on YARN cluster, where YARN (Yet Another Resource Negotiator) is a cluster management technology that separates the resource management and processing components.

*Data storage layer*: Data storage is a layer provides simple access to data stored in persistent storage providers, Amazon Simple Storage Server (AWS S3) in our case. This layer is where the

input, output, and intermediate data live. I first upload the public dataset to AWS S3 bucket so that it is ready to be consumed by the ingestion layer. At any stage of the processing pipeline may store its intermediate data back to the AWS S3. This would typically occur when Spark RDDs is set to persist on disk or checkpointing RDDs.

## 4.3 Parallel Computing Design Considerations

In this section, I will discuss two design considerations that are significant to parallel computing, namely load balancing, and data locality.

### Load Balancing

While MapReduce model enables us to write distributed application without having to worry about the underlying distributed computing infrastructure, it gradually exposes some shortcomings. Typically, handling skewed data can cause the imbalance of the workloads. After mapper processes data, the result will be sent to reducer by partition function. An inappropriate partition algorithm may result in poor network quality, the overloading of some reducers and the extension of the execution time of job. That is, using an inappropriate algorithm to process skewed data will form a negative impact on the system performance. Distributed computing frameworks such as Spark do not guarantee balanced workloads at each transformation step. In order to solve load imbalance problems and improve performance of cluster, I propose partition algorithms to guide the process of assigning data.

### Data Locality

Data locality is another crucial consideration for the performance of task scheduling in distributed computing system. From distributed computing perspective, data locality, as known as data placement or proximity of data source, means that computation should happen as close as where the data is stored to reduce data transfer over the network. Spark's scheduler already has a concept of locality-sensitive scheduling such that tasks are executed locally when possible. However, it is difficult to avoid data shuffling especially in the case of data aggregation. There are many ways

to join two datasets in Spark. However, not all the methods result in the same performance. Also, different method may perform differently depending on use cases. Therefore, I perform extensive analysis on those merging methods in this thesis in order to select the most appropriate ones suitable for our system.

## 4.4 Evaluation Method

For both Gathering and TCompanion models, the main goal is to evaluate the following quality attributes, namely precision and recall, performance, scalability, effectiveness of optimization strategies, and stability.

### 4.4.1 The Dataset and Evaluation Setttings

All the evaluations are conducted on Amazon Web Services (AWS). I use one master node and eight slave nodes in AWS. Table 4.1 shows the deployment settings in our experiments. Details of cluster configuration and deployment procedure will be described in Appendix A.

Table 4.1: Deployment settings

| **Factor** | **Description** |
|---|---|
| compute unit | AWS EC2 t2.large |
| core number (in each unit) | 2 |
| memory (in each unit) | 8GB |
| storage | Amazon S3 |
| computing platform | Apache Spark 1.5.1[1] |

To study our algorithms on streaming data, I utilize Kafka+Spark Streaming[2] to process our heterogeneous streaming data. In our streaming data, over 10 thousands new locations will be arrived per second. I use Kafka to let Spark cluster receive each micro batch of data (i.e., data in several seconds), but do not need to store trajectory in HDFS. Then the Spark cluster will handle our streaming data step by step.

I use a real GPS trajectories dataset[3]. This dataset was collected in (Microsoft Research Asia) Geolife project by 178 real users in a period of over four years (from April 2007 to October 2011),

[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

which is recoded a broad range of these users' outdoor movements, including not only life routines like go home and go to work but also some entertainments and sports activities, such as shopping, sightseeing, dining, hiking, and cycling. In this dataset, 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, and have a variety of sampling rates (e.g. every 1–5 seconds). In this dataset, I ignore the date attribute of each GPS trajectory, so that they can be regarded as the trajectories with in one day. Then I use them to simulate a heterogeneous trajectory stream in which each location record contains the information of latitude, longitude and timestamp of an object.

### 4.4.2 Ground Truth

To measure the returned results from different algorithms, I need to know which results are *ground truth*. Since in our dataset, lots of moving objects are far away from others, I only sample part of trajectories of objects within a small region (e.g., 1km x 1km). Our sampling process includes 3 steps: 1) choose a fixed number of objects (e.g., 4 objects) in this small region; 2) sample locations of these objects a time period; 3) pick up the objects that have at least one location is close to trajectories of the original chosen 4 objects (i.e., less than the distance threshold). This reason is that this process can ensure the chosen trajectories are more likely to be related with others. As result, I generate 12 trajectories from our dataset in this sampling process. I would like to observe which trajectories should be real companions (ground truth) in the sampling dataset.

I visualize the sampling trajectories using spatiotemporal visualizer[4], where I can define a time window by adjusting the slider in the UI. The visualizer, however, does not reveal the factors such as heading directions and speed of a moving object. That is, I assume an object moving along its trajectory with constant speed. As shown in Figure 4.2, some objects' trajectories are always close but does not imply they are in companion without considering the time constrain. Consider the two trajectories within the zoom area, each trajectory has two timestamps (shown in diamond and circle accordingly), indicating the object traveled to the north form 6:15am to 6:25am. The second object traversed the same path about 2 hours later (8:30am - 8:40am). Technically, these trajectories are

[4]https://github.com/hugocore/spatiotemporal-visualizer

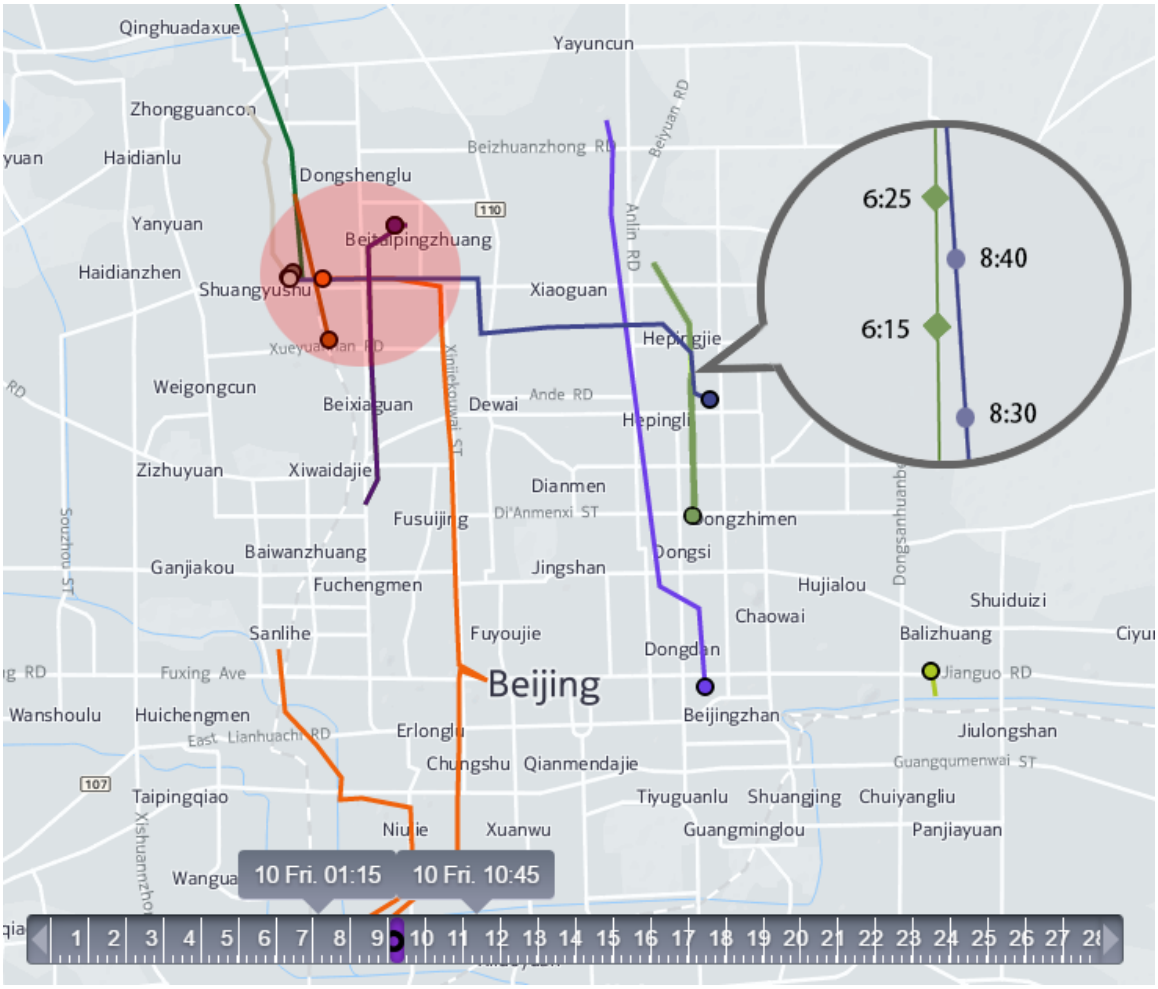



Figure 4.2: Trajectory Visualization

not companions. Thus, I measure each pair of objects as follows:

$$dist(o_i[t_1] - o_j[t_2]) \leq \epsilon, (t_1, t_2 \in \Psi \wedge |\Psi| \geq \tau), \tag{1}$$

where $t_1$ and $t_2$ are any two timestamps in a set $\Psi$, $o_i[t_1]$ and $o_j[t_2]$ denote the locations of $o_i$ and $o_j$ at timestamps $t_1$ and $t_2$ respectively. If there are more than $\tau$ timestame pair (like $t_1$, $t_2$) that satisfies $dist(o_i[t_1] - o_j[t_2]) \leq \epsilon$, $o_i$ and $o_j$ are ground truth companions.

# Chapter 5

# Snapshot Model

## 5.1 Overview

In this Chapter, our goal is to discover gatherings from trajectory data. Generally, there are two types of trajectory data: 1) archived trajectory data and 2) trajectory data streams. The archived trajectory data are usually stored-and-scanned that is suitable to batch-based processing. On the other hand, the trajectory data streams are constantly updated. Hence I propose a window-based model that partitions data based on time windows, (e.g., trajectory data in every 10 seconds), and process the partitioned data in micro-batching.

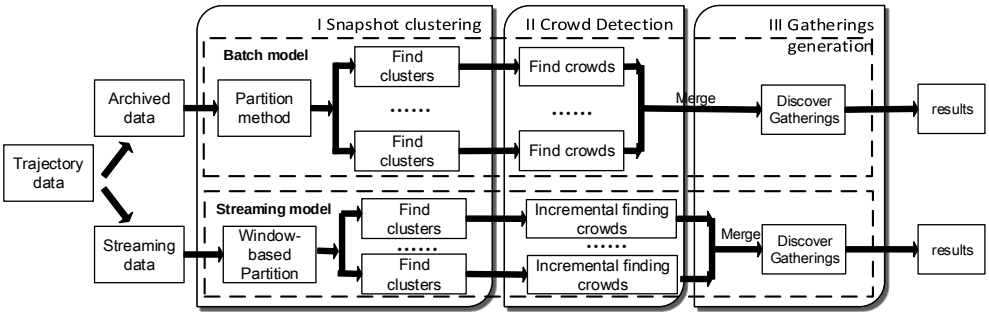



Figure 5.1: Parallel Gathering Discovery Framework

In this way, I define an analysis framework depicted in Figure 5.1 that unifies the processing of both types of trajectories in three phases, namely (1) *snapshot clustering* that clusters moving objects in snapshots; (2) *crowd detection* that finds clusters during a certain time intervals that satisfy thresholds on distances and densities; and (3) *gathering generation* that aggregates participation of

moving objects in crowds over times.

## 5.2 Definitions and Notations

The definition and notations of aforementioned concepts in the framework is provided as follows.

Let $\mathcal{O} = \{o_1, o_2, ..., o_i, ..., o_m\}$ be the set of all moving objects. In the snapshot model, each snapshot is the set of these $m$ objects' location at a timestamp [30]. To discover groups in each snapshot, I adopt density-based clustering algorithms [10] to use in our problem (like [30, 40, 41]). Based on the discovered clusters in snapshots, I formally define concepts in this work below.

**Definition 1.** ***(Crowd)****: Let $k_c$ be the lifetime threshold which denotes the minimum duration time of each crowd, $\Delta t$ be time threshold and $\mu$ be the density threshold, and $\delta$ be the cluster-distance threshold between any two clusters. A crowd $\mathcal{C}_r =< cl_{t_a}, cl_{t_{a+1}}, ..., cl_{t_b} >$ is a sequence of snapshot clusters in a time interval $I$ between $[t_a, t_b]$, i.e.. $I = t_b - t_a$, which satisfies the following requirements:*

*1)* $I \geq k_c \wedge \forall i(a \leq i \leq b)$, $t_{i+1} - t_i \leq \Delta t$;

*2)* $|cl_{t_i}| \geq \mu$;

*3)* $Dist(cl_{t_i}, cl_{t_{i+1}}) \leq \delta$.

For instant, I set $k_c = 2$, $\mu = 2$ and $\delta = 5$ (assume the maximum distance between $o_1, o_2, o_3$ in the $t_1$ snapshot and $o_1, o_2$ in the $t_3$ snapshot is not more than 5). $< cl_{t_1} = \{o_1, o_2, o_3\}, cl_{t_3} = \{o_1, o_2\} >$ is a crowd.

**Definition 2.** ***(Participator)****: Given a crowd $\mathcal{C}_r$, an object $o$ is called a participator of $\mathcal{C}_r$ if it appears in at least $k_p$ snapshot clusters of $\mathcal{C}_r$. Let $\mathcal{C}_r(o)$ denote the set of snapshot clusters in $\mathcal{C}_r$ that contains object $o$, i.e., $\mathcal{C}_r(o) = \{cl_t | cl_t \in \mathcal{C}_r, o(t) \in cl_t\}$. Then the participators of $\mathcal{C}_r$ are the object set Par($\mathcal{C}_r$) = $\{o|\ |\mathcal{C}_r(o)| \geq k_p\}$.*

In our example, if $k_p = 2$, Par($\mathcal{C}_r$) = $\{o_1, o_2\}$.

**Definition 3.** ***(Gathering)****: A crowd $\mathcal{C}_r$ is called a gathering iff there exists at least $m_p$ participators in each snapshot cluster of $\mathcal{C}_r$, i.e., $\{o|o(t) \in cl_t, o \in \mathcal{C}_r(o)\} \geq m_p$.*

I set $m_p = 2$. Then $< cl_{t_1} = \{o_1, o_2, o_3\}, cl_{t_3} = \{o_1, o_2\} >$ is a gathering. Intuitively, a gathering trends to discover a sequence of clusters that contain similar objects and distances between consecutive clusters can be bounded.

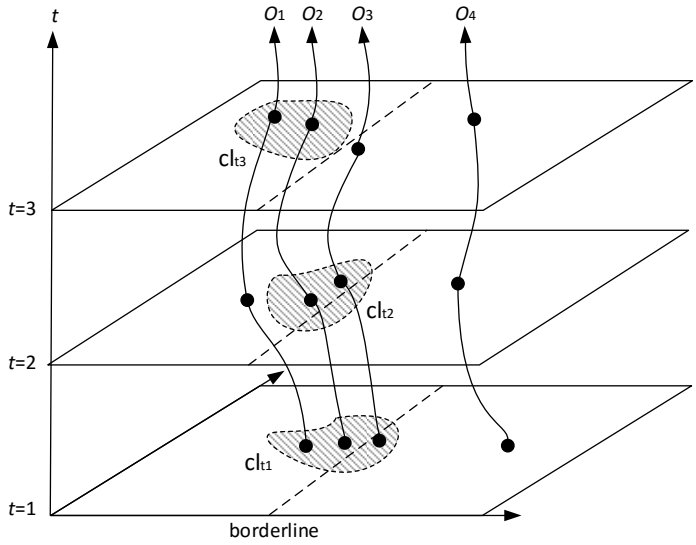



Figure 5.2: An Illustrating Example of Gatherings of Trajectories

## 5.3 Implementation

Based on the above definition and the conceptual framework, I further design the workflows of gathering discovery in a parallel manner.

For archived trajectory data, the batch-based approach includes 4 steps as follows.

- *Data partition*–In this step, I present a grid-based partition method to partition data within each snapshot into $n$ sub-partitions.
- *Group objects into clusters*–I use DBSCAN clustering approach to group objects in each sub-partition and then merge clusters in these sub-partitions.
- *Crowd detection*–I propose efficient join methods in the parallel framework to join clusters and thus generate crowds.

- *Discover gatherings*–I design the discovering algorithm to find gatherings from crowds.

The streaming model also first groups objects in each time window (such as every 10 seconds). For the crowd detection phase, instead of storing all the historian trajectories up-to-date I address an incremental crowd detection approach to generate crowds. *Incremental* means the latest windows of date should be appended to the current window Finally, it returns the gatherings in the adjacent time windows. I summary the key steps of this workflow below:

- *Window-based partition*–I first buffer data in a time window, and use our grid-based partition method to partition the buffered data into $n$ sub-partitions as well.
- *Group objects into clusters*–Similar to our batch model, I also use DBSCAN clustering approach to group objects in each sub-partition and then merge clusters in these sub-partitions.
- *Incremental crowd detection*–In this step, I propose the incremental algorithm to generate crowds from clusters while avoiding scanning all the data.
- *Discover gatherings*–I present the discovering algorithm to find gatherings from crowds.

The design of discovery method focuses on data partition and data shuffling to improve the scalability as the workload in terms of data size increases. In this section, I present the techniques applied in three phases of the analysis framework.

### 5.3.1 Trajectory Data Partition

First of all, data within each snapshot are partitioned into $n$ sub-partitions for parallel processing. Fixed grid partitioning is a spatial indexing technique [1],[8] suitable for partitioning trajectory data as it does not incur any extra storage or computation overhead other than replicated boundary objects. I employ the fixed grid partition due to its straightforwardness, where the entire space is partitioned into equal sized grids represented by minimum bounding rectangles (MBRs). Suppose that the given two dimensional space $\mathcal{S}$ = a $\times$ b and $\omega$ be the grid spacing, and grid Id of the partition is denoted by $gid$. Given a spatial point $\{x, y\}$ where $0 \leq x < a, 0 \leq y < b$, then the grid $\mathcal{G}$ is represented as $\mathcal{G}(i,j) = \left\{ i\frac{a}{N} \leq x < (i+1)\frac{a}{N}, i\frac{b}{M} \leq y < (i+1)\frac{b}{M} \right\}$ where $N$ is number of grid column and $M$ is the number of grid row. That gives that $i = \left[\frac{xN}{a}\right], j = \left[\frac{yM}{b}\right]$.

For instance, in the public Geolife dataset that is used in this study, the world space is divided by an uniform grid with $\omega$ = 0.1 degree (about 11 km, average width of large city or district), -180° $\leq x \leq$ 180°, -90° $\leq y \leq$ 90°. Hence, this gives $N$=3600 and $M$=1800. If a given point coordinate is (39.971573,116.33191), I can immediately conclude that the point is belong to the grid cell $[399, 1163]$.

The partition affects the clustering of the data objects, especially those near to the partitioning boundaries. I use an example to illustrate this issue. As shown in Figure 5.2, the objects $o_1$, $o_2$ and $o_3$ belong to one cluster $cl_{t_1}$ given the whole snapshot. However, if the snapshot is partitioned, and $o_1$ and $o_2$ are within one sub-partition, and $o_3$ is within another sub-partition, only $o_1$ and $o_2$ are grouped into a cluster, while $o_3$ is left out. This is not consistent with the clustering on the whole snapshot. Therefore the partition can potentially lead to missing objects and less precise results.

I resolve this issue by using the method called *multi-assignment* [17]. In this method, each object near to a boundary is replicated to the neighbouring partitions. If an object $o$ is very close to the partitioning borderlines (*i.e.*, the distance is not more than $\epsilon$), this object $o$ and other objects within the neighbouring partition may also generate a cluster. In such case, I should assign $o$ to multiple sub-partitions, so that the clustering results are not missed.

### 5.3.2 Snapshot Clustering

I develop the Algorithm 1 to cluster the data in each partition. The map function first extracts temporal and spatial information from inputs and emits a key-value pair as $<< t, gid >, p >$, where $t$ denotes the timestamp of the point $p$, $gid$ denotes the grid ID and $p$ represents one point (including $x$- and $y$-coordinates). Next, the grid partition method ensures object points with the same timestamp located within the same geo-spatial boundary are grouped in the same sub-partitions. Finally, I employ DBSCAN [10] to identify clusters which satisfy the distance and density constraints. Euclidean distance is the chosen distance measure for constructing clusters. Distance threshold $\epsilon$ affects the size of clusters and subsequent computation. I have selected the value $\epsilon$ ranging from 0.0005 to 0.005 degree (approximately 10 to 100 meters) which is common when a gathering happens.

Spark is a parallel processing framework that supports distributed operations on key-value pairs.

**Algorithm 1:** Grid Based Partition

**Input :** object set $\mathcal{O}$, grid spacing $\omega$, distance threshold $\epsilon$, density threshold $\mu$
**Output:** clusters per timestamp per partition

```
1  Map(documentId a, document d)
2  for each record r ∈ doc d do
3      Extract time t, point p from r
4      EMIT(timestamp t, point p)
5  gridPartitioner(timestamp t, point p)
6  for each grid do
7      gid ← getPartitionId(p)
8      EMIT(timestamp t, gridId gid, objectId oid)
9  Reduce(timestamp t, gridId gid, objects [o_1, o_2, ...])
10 clusters← DBSCAN(ϵ, μ, objects)
11 for all cluster cl ∈ clusters do
12     EMIT(timestamp t, gridId gid, cluster cl)
```

I present the workflow of our snapshot clustering algorithm using Spark in Figure 5.3. In this workflow, the $repartition()$ transformation plays a significant role since it ensures a partition does not span multiple machines. This means aggregation operations such as $reduceByKey()$ can be done locally without shuffling data across network, which significantly improve the runtime performance for a partition that contains a large dataset.

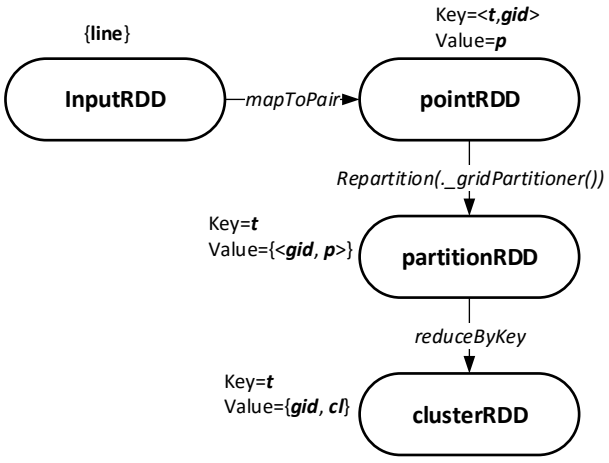



Figure 5.3: Gathering: Spark Worfklow for Finding Clusters

Next, the clusters are further merged by Algorithm 2. In our workflow, all clusters discovered from the same timestamp are dispatched to an identical Spark partition. Now the problem is without

**Algorithm 2:** Merging Clusters

**Input :** clusters per timestamp per partition
**Output:** merged clusters per timestamp

```
1  Map(timestamp t, gridId gid, cluster cl)
2  EMIT(timestamp t, cluster cl)
3  Reduce(timestamp t, clusters [cl_1, cl_2, ...])
4  for all cluster cl ∈ clusters do
5      for all cluster cl′ ∈ clusters do
6          if cl ≠ cl′ then
7              C_i ← intersect(cl, cl′) //Find common objects
8              if C_i not empty then
9                  C_j ← join(cl, cl′)
10                 Emit(timestamp t, C_j)
```

knowing the boundary objects in advance, it is necessary to perform a full self join of the cluster RDDs in order to compare each cluster pair to determine whether they should be merged. Consider an example, suppose the snapshot RDD contains $s_1 =< 1, < 3, 1 >>$, $s_2 =< 1, < 4, 4 >>$, $s_3 =< 1, < 3, 3 >>$, $s_4 =< 1, < 3, 6 >>$, $s_5 =< 1, < 4, 3 >>$. Intuitively, all these snapshots can be grouped into the same partition since they all share the same key. That is, the partition containing these snapshot becomes $\mathcal{P} = \{< 1, < 3, \{1, 3, 6\} >>, < 4, \{3, 4\} >>\}$. From the result of merging clusters, it is can be identified that object 3 is the boundary object between grid 3 and grid 4. As shown in Figure 5.4, the $clusterRDD$ is spitted into two RDDs, namely $clusterWithBoRDD$ and $clusterWithoutBoRDD$. As the names suggest, the former one filters the clusters that consist of one or more boundary objects; the latter RDD is the result of subtraction from $clusterRDD$ to $clusterWithBoRDD$. Afterward, a self-join transformation is applied to $clusterWithBoRDD$ to produce $mergedClusterRDD$. Finally I obtain the $finalClusterRDD$ by combing two result RDDs.

The Spark workflow of merging clusters are illustrated in Figure 5.4.

### 5.3.3 Crowd Detection and Gathering Discovery

The partitions defined in Algorithm 2 is preserved to ensure the level of parallelism. Since objects are replicated to neighbouring partitions, clusters may span more than one partitions. Therefore, I again replicate a cluster to all its covering partitions. Next, I perform Secondary Sort to group

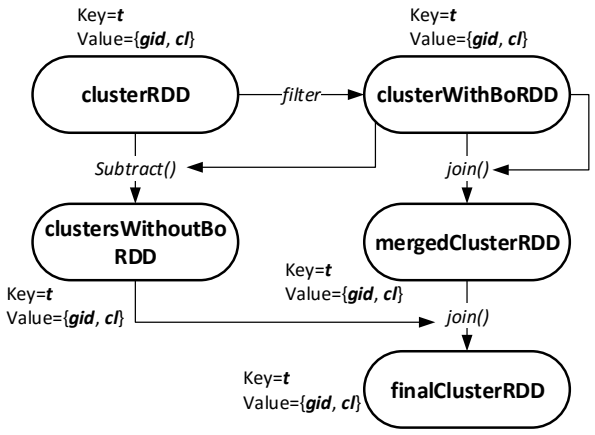



Figure 5.4: Gathering: The Spark Workflow for Merging Clusters

clusters the same partition, and sort clusters by timestamps. I create a composite key $< t, gid >$ to sort all of the records on the composite key, and then use a custom partitioner and grouping function to ensure that all the records with the same $gid$ appear in the same partition.

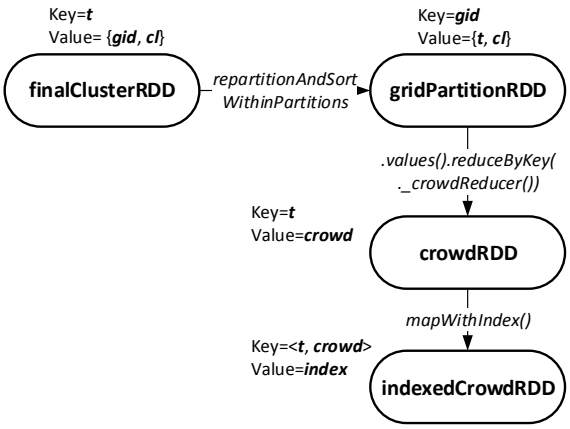



Figure 5.5: Gathering: Spark Workflow for Crowds Detection

Now in each partition I have a set of $< t, cluster >$ pair sorted by timestamps in an ascending order. I further pass the data set into a reducer function, namely $crowdReducer()$ to analyze the crowd candidates. Consider an example, given a set of time-cluster pair in a partition $cl_{t1} =< 1, \{1, 2\} >, cl_{t2} =< 3, \{2, 3, 4\} >, cl_{t3} =< 6, \{2, 4\} >, cl_{t4} =< 7, \{2, 4, 5\} >$. Assume $\Delta t$=2, $\mu$=2, $Dist(cl_{t1}, cl_{t2}) < \delta$ and $Dist(cl_{t3}, cl_{t4}) < \delta$, Table 5.1 illustrates the crowds discovery in deteail. I first initialize two empty sets, namely $CrowdCandidates$ and $Crowds$, then iterate

through the time-cluster pairs. I put the current pair into the candidate set and compare with the previous candidates. (e.g. from Iter.1 to 2) If the candidate set can not be extended, I save the set into $Crowds$ (e.g. from Iter.2 to 3). As a result, I can discover two crowds in our example.

Table 5.1: Illustration of Crowd Discovery

| Iter. | Crowd Candidates | Crowds |
|---|---|---|
| 1 | $[1, \{1, 2\}]$ | |
| 2 | $[1, \{1, 2\}], [3, \{2, 3, 4\}]$ | |
| 3 | $[6, \{2, 4\}]$ | $[1, \{1, 2\}], [3, \{2, 3, 4\}]$ |
| 4 | $[6, \{2, 4\}], [7, \{2, 4, 5\}]$ | |
| 5 | | $[6, \{2, 4\}], [7, \{2, 4, 5\}]$ |

The last phase of our framework is discovering all gatherings from the crowds obtained. Recall Definition 2 and Definition 3, a crowd is called a gathering if it satisfies the temporal and spatial requirements.

Figure 5.6 illustrates the details. The $indexedCrowdRDD$ is inverted index such that the key is composed of crowd index and object ID, and the value is the timestamp. Hence I use the call of $reduceByKey()$ to count the occurrence of a target object in the specific crowd. Next, I filter out the objects which do not meet the requirement Par($\mathcal{C}_r$) = $\{o|\ |\mathcal{C}_r(o)| \geq k_p\}$. To obtain the participators, I group the objects shared the same crowd index. Eventually, the $participatorRDD$ is passed into the second filter applied to $indexedCrowdRDD$ in order to produce $gatheringRDD$.

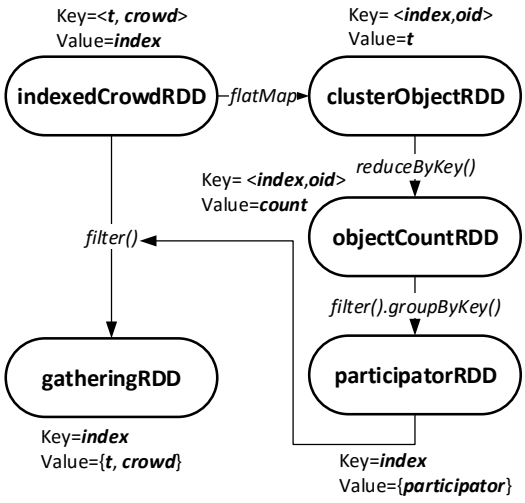



Figure 5.6: Gathering: The Spark Workflow for Gathering Discovery

### 5.3.4 Streaming-based Gatherings Discovery

Compared to the batch-based approach, I need to process the streaming data at the size of dozens of thousands records arriving per second. Thus I consider previous trajectory data should be appended to the streams periodically. The interval is regarded as a time window. Thus the partition and clustering are performed on each window period.

**Theorem 1.** *Let $k_c$ be lifetime threshold. In the crowd detection process, only the clusters in $k_c$ time prior to the current window need to be appended without missing an object in the crowd.*

*Proof.* Suppose that $cl_t$ is a discovered cluster in the current window. According to the definition of crowds, if any cluster $cl_{t_x}$ can be included in a crowd with $cl_t$, $t - t_x$ must be equal or less than $k_c$. Otherwise if $t - t_x > k_c$, the cluster $cl_{t_x}$ does not satisfy the crowd definition and thus is not considered. Therefore, I can only check the clusters in $k_c$ time prior to the current window. □

Based on above theorem, the discovery algorithm is illustrated in Algorithm 3. The implementation of this algorithm follows three phases of the analysis framework. Each phase shares the similar Spark workflows for finding clusters, merging clusters, detecting crows and discovering gatherings as depicted in Figure 5.3 to Figure 5.6.

**Algorithm 3:** Online Discovery Algorithm

**Input :** Trajectory data stream $\mathcal{S}$, cluster set $\mathcal{C}$ in $k_c$ time
**Output:** Gatherings

```
for each window W_j in S do
    Find each cluster cl_t in W_j
    Put cl_t into C'
    Discover crowds from C ∪ C'
    Delete cluster cl_{t_x} iff t − t_x > k_c
    Generate gatherings from crowds
```

## 5.4 Optimization

The optimization strategy focuses on data locality to minimize the amount of data shuffled in the workflows. Data locality means tasks are executed as close as possible to where the data locates. Poor data locality causes data shuffling. Data shuffling incurs the most significant cost in

the overall processing as it requires frequent data serialization/deserialization, disk I/Os, and even data transmissions among physical worker nodes. Therefore, intermediate data communication from the workflow steps may become performance bottlenecks. In Spark, shuffling usually caused by operations such as $join()$ or aggregations on keys or values. Our solution rely on techniques of partitions, load balancing and tuned operation of $join()$. I discuss the techniques I have identified for the optimization purpose.

### 5.4.1 Partitioning

An aggregation can be processed locally without shuffling data if data are partitioned in a way such that data with the same key or hashing result of the key are in the same partition. Consider the workflow in phase II, the crowd detection algorithm adopts the divide-and-conquer paradigm. Crowds are discovered in each grid cell identified by an identifier $[i, j]$. I have several choices to realize fixed-grid partitioning in Spark, namely hash partition, range partition, and custom partition. All of these partitioning are key oriented: *Hash Partitioning* determines the partition as $key.hashCode()\%numPartitions$; *Range Partitioning* uses a range to distribute to the respective partition if the key falls in a range. I select the *Hash Partitioning*. As I will present shortly in the evaluation, hash partitioning improves performance of the subsequent key based transformation.

### 5.4.2 Data Skew

In distributed computation, data skew can be another performance bottleneck which may diminish the gains made from partitioning. Fixed-grid partitioning is based on a major assumption that data are evenly distributed. This is often not the case while analyzing geo-spatial data. Gatherings tend to happen more frequently in downtown than suburban areas. As a result, it ends up with some partitions only have a few clusters whereas the others contain several thousands of clusters. In the Spark runtime environment, a set of parallel tasks is defined into a stage for execution. The subsequent stages do not begin until all preceding stages have finished. Thus tasks with unbalanced workload tend to dominate the overall delays.

Taking the output from the workflow in crowd detection of Figure 5.5, I outline the techniques used to mitigate the data skew problem as shown in Figure 5.7.

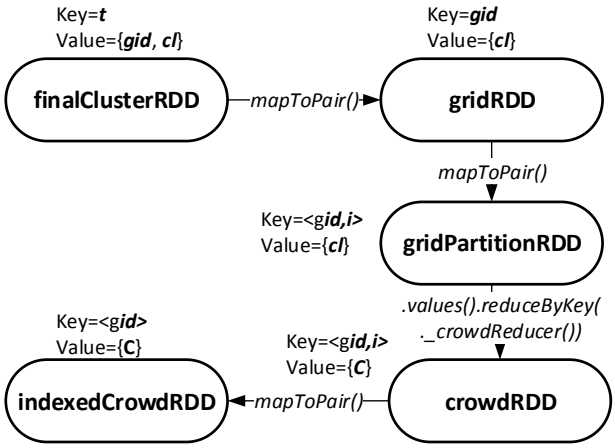



Figure 5.7: Gathering: The Spark workflow for skewed data handling

1) Find skewed partitions by filtering out the keys from the output of the RDD *finalClusterRDD*. Those keys contain grid IDs ($gid$) for those grid partitions that contain clusters more than a threshold.

2) Modify these keys from $< gid >$ to $< gid, i >$ where $i$ is a random number ranged from 0 to $numPartition$.

3) Hash partition keys (in step 2)) according to $key.hashCode()\%numPartitions$.

4) Find crowds in each partition.

5) Remove $i$ from the key and join the result.

This technique aims to break the skewed partitions down to smaller partitions, which can result in more load-balanced for data in each partition. This improvement becomes remarkable if the data is heavily skewed.

### 5.4.3 Efficient Join

Join of two data sets is one of the commonly used operations in our workflow, such as the step of merging clusters. In distributed system, joining data can cause significant delay since data are distributed among many nodes and they need to be shuffled before a join. I consider two types of join methods, namely Shuffle Join and Broadcast Join.

- Shuffle join is the built-in $join()$ transformation that data of two RDDs with the same key are

redistributed to the same partition. Records in each RDD is shuffled across the network.

- Broadcast join is suitable joining a large data set with relatively small data sets. The RDD created from the large data set is broadcast to each worker node so that the join does not require shuffling data of the large data set.

As shown in Figure 5.4, the procedure to merge clusters invokes join operations. In our problem, I have observed based on our experimental dataset that clusters with boundary objects $C_{BO}$ are only approximately 5% of the entire cluster dataset $C$. Since $|C_{BO}| \ll |C|$, I collect $C_{BO}$ into the driver and further utilize Spark's *borardcast* variable to distribute the data to all execution worker nodes. Hence, broadcast join is beneficial because items in $C$ do not required to be shuffled.

## 5.5 Evaluation

Table 5.2: Gathering: Parameter settings

| **Factor** | **Range** | **Default** |
|---|---|---|
| $\delta$ | $\{0.0005, 0.005, 0.001, 0.005\}$ | 0.005 |
| $n$ | $[2, 4, 6, 8, 10, 12, 14, 16]$ | 8 |
| $\Delta_t$ | $[20, 40, 60]$ | 60 |
| $k_c$ | $[60, 80, 100]$ | 100 |

I perform dedicated experimental evaluation for both batch-based and stream-based processing. For the batch-based process, I focus on the quality attributes

*Scalability*: I analyze the latency of in response to the number of partitions and the size of the cluster.

*Effectiviness*: I evaluate the proposed optimizations in terms of data shuffling ratio and latency. I observe whether the optimization is effectively impact the system performance.

For the streaming-based processing, I are interested in the following quality attributes:

*Stability*: I perform a stability analysis by observing the processing time and scheduling delay versus input rate.

*Efficiency*: I analyze the performance of the streaming-based processing under various control parameters such as $\delta$ and $k_c$. The performance is measured in terms of throughput and latency.

Table 6.2 shows our parameter settings.

**The Evaluation Results of Batch-based Processing**

**Scalability Observations.** I vary the number of partitions and the size of the clusters (i.e. the number of the worker nodes in the cluster). Figure 5.8 illustrates the end-to-end execution time measured.

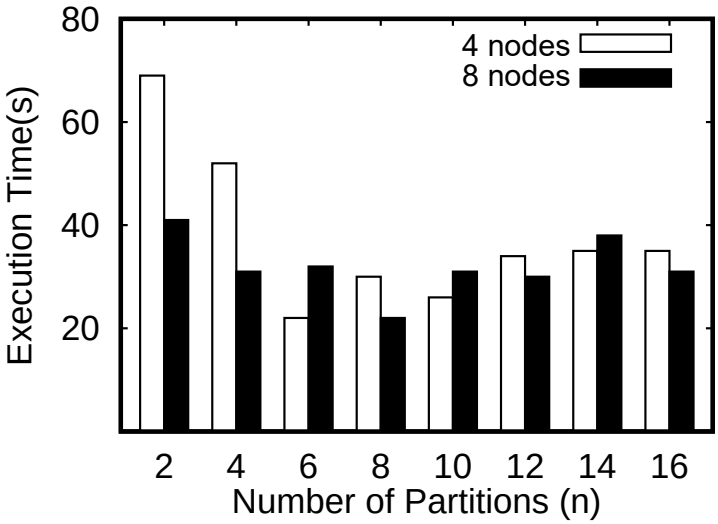



Figure 5.8: Gathering: Scalability Comparison - Numb. of Partitions vs. Size of cluster

I observe that the system achieves the optimal performance when the number of partitions is near the number of cores in the cluster. Each partition is run on one core of the cluster. When the number of partitions are larger than the number of cores, excessive partitions need to wait for available cores till any parallel partition completes. When the number of partitions is low, the workload on each partition running on one core becomes big while other cores are idle.

**Effectiveness Observation.** In this experiment, I aim to observe the effect of different partition methods on data shuffling. I consider three cases, as follows:

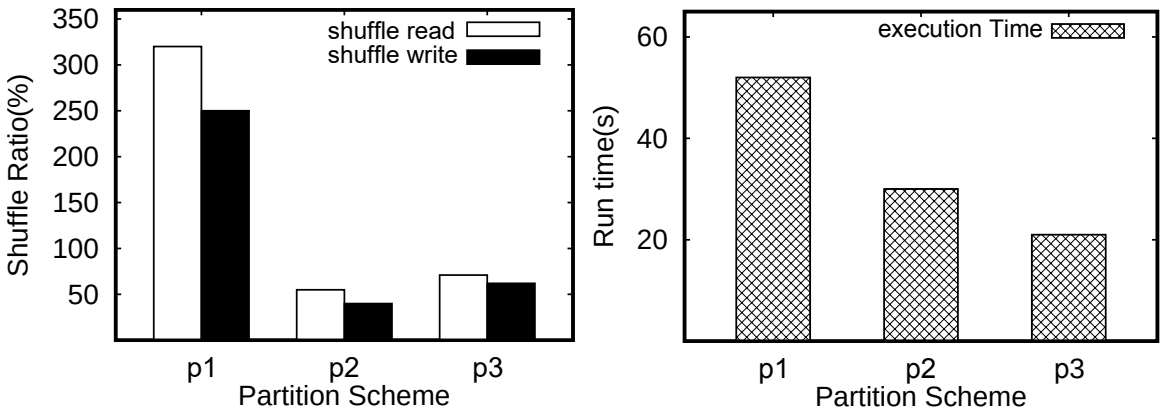
$p_1$: hash partition without $repartition()$ transformation: the partition based on default hashing does not distribute data uniformly.

$p_2$: hash partition with $repartition()$ transformation: The $repartition()$ transformation actually shuffles the original partitions and repartition them. However, it does not guarantee to avoid data skew.

$p_3$: hash partition with skewed data handling : the proposed optimization described in Figure 5.7

is applied.

I fix 8 worker nodes in a cluster and use 8 partitions. The other parameters use the default values. The data shuffle is represented by two metrics: *shuffle read*, and *shuffle write*, measured as the ratio (%) of the input data. *Shuffle read* refers to the sum of serialized *read* data of all executors. Likewise, *shuffle write* is the sum of serialized *write* data of all executors. Both of these metrics are obtained from the utility called Spark UI.

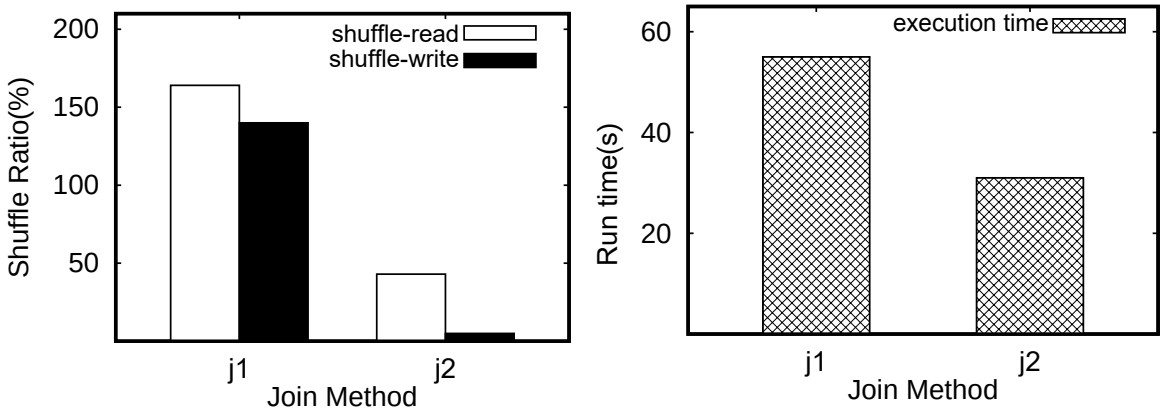



(a) Shuffle Read/Write (b) Execution Time

Figure 5.9: Gathering: Partition Performance Comparison

As Figure 5.9 illustrates, $p_1$ introduces approximately 300% read and 250% write. In contrast, $p_2$ and $p_3$ has dramatically reduced the data shuffled down to approximately 50%. It is expected that $p_3$ has slightly more data shuffled because of the replication of skewed data partitions. The result shows $p_2$ incurs the least data shuffle, while $p_3$ has minimal execution time.

**Join Performance Evaluation.** I compare the performance of two join methods: reduce-side join $(j_1)$, and broadcast join $(j_2)$ shown in Figure 5.10. $j_1$ results in a relatively high shuffle read and shuffle write ratio. One contributing factor is that every join operation requires data with the identical key to be shuffled to the same partition. Unlike $j_1$, $j_2$ generates a small amount of shuffle read and nearly zero shuffle write. The shuffle read may due to the smaller dataset broadcasted to each executor. In this case, two datasets to be joined are in the same partition. That is, no data shuffle is required. From the computation perspective, $j_2$ outperforms $j_1$ by approximately 50%. Therefore, I conclude that map-side join is more efficient in the case of a large dataset joins a smaller dataset.

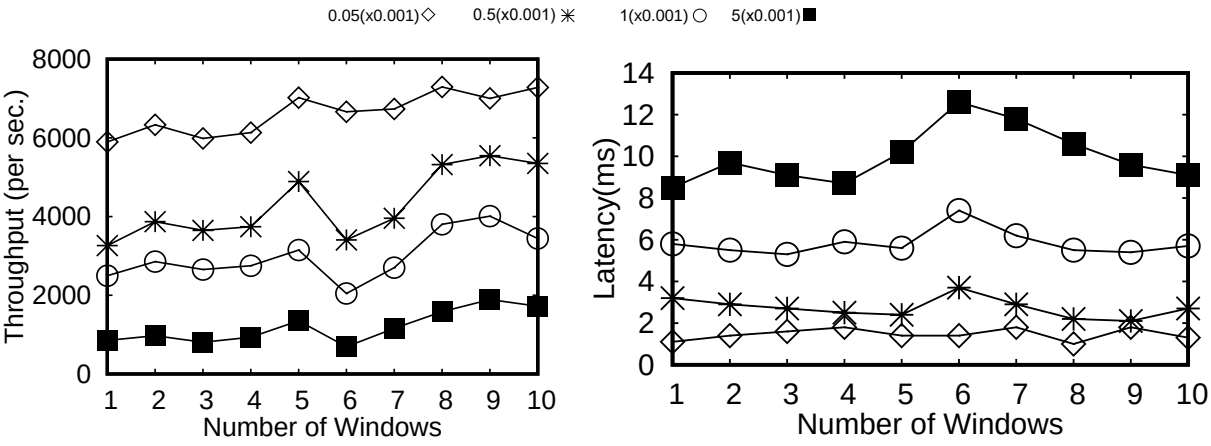



(a) Shuffle Read/Write (b) Execution Time

Figure 5.10: Gathering: Join Performance Comparison

**The Evaluation Results of Streaming-based Processing**

**Effect of Parameters.** In the following experiments, I analyze the throughput and latency of the streaming-based workflow under different parameter settings. I first evaluate the effect of parameters $\delta$ alternated from 0.0005 to 0.005, and other parameters are used default settings. Also, I observe the result in 10 windows, whereby each window length is 10 seconds. Figure 5.11 shows the results under different $\delta$ value.




(a) throughput (b) latency

Figure 5.11: Gathering: Vary $\delta$ from 0.0005 to 0.05 - Streaming Mode

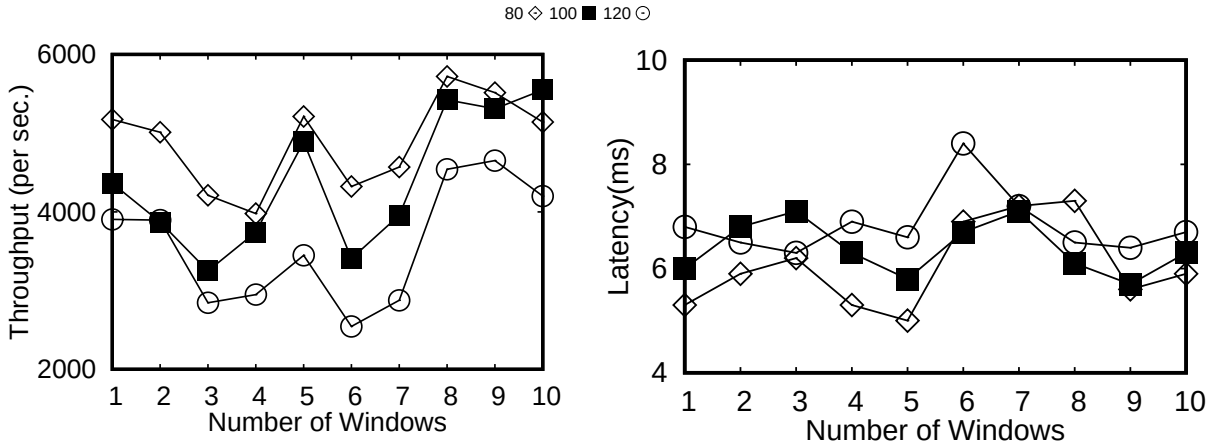
I obtain the following two observations:

1) The throughput is higher and the latency is lower with the smaller $\delta$. The reason is small $\delta$

implies smaller clusters and thus fewer objects to be processed.

2) Except for $\delta$ = 0.0005, the throughput of other $\delta$ values all drop at window 6. This indicates that the data size contains in this window is smaller than other windows (e.g. at noon, most people tend to hang out for lunch). Consequently, it results higher latency.

Figure 5.12 shows the effect of parameter $k_c$ on the performance. I vary the lifetime threshold $k_c$ from 60 to 100 seconds. Notice that length of Spark window and $\Delta t$ are also impacted by $k_c$. The window size needs to be equal or larger than $k_c$. Thus, I set window size to 80, 100, and 120; $\Delta t$ to 40, 60, and 80 respectively. The result shows the throughput drops at around the window 6 that is consistent with the observations in Figure 5.11.

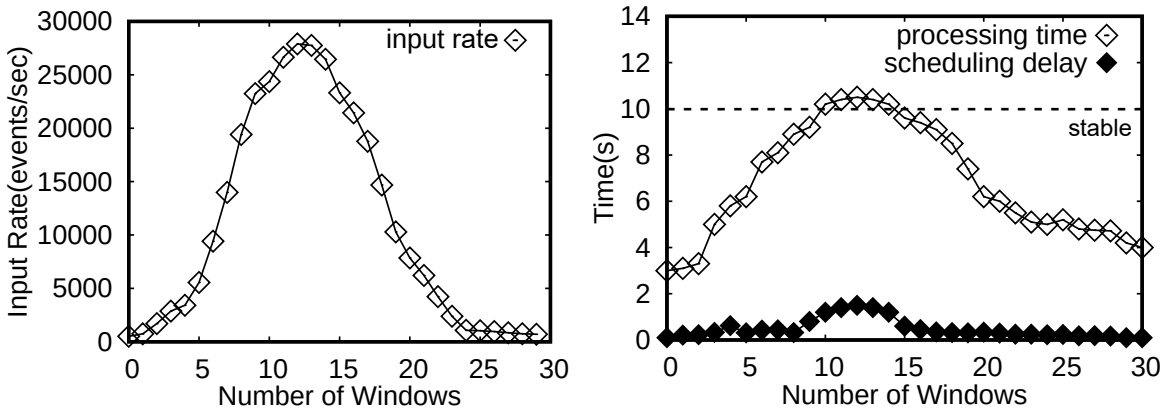



(a) throughput (b) latency

Figure 5.12: Gathering: Vary $k_c$ from 60 to 100 - Streaming Mode

**Stability.** I study the stability of our streaming-based processing system by injecting the entire dataset in 30 windows, where each window size is set to 10 seconds. In this evaluation, I observe the processing time and scheduling delay from Spark UI in response to the input rate. In Spark, an application running on a cluster is said to be stable if the processing time of each micro-batch is less than the window size, as known as batch interval.

In Figure 6.7, I set a horizontal line at 10 seconds indicating the stability threshold, if the processing time is over the threshold, it implies the system has some tasks waiting in the queue and thus results in scheduling delay. From Figure 6.7, I can see that the input rate increase to peak around windows 10 to 14. This leads the processing time larger than 10 seconds, causing a few




(a) input rate (b) processing time, scheduling delay

Figure 5.13: Gathering: Stability Evaluation - Streaming Mode

milliseconds scheduling delay. However, the scheduling delay does not continue to increase after window 14. The delay is maintained to be comparable to the batch size. To summarize, I observe our system keeps up with the growth of data input rate.

# Chapter 6

# Trajectory Slot Model

## 6.1 Overview

A trajectory is the sequence of spatial locations or points that a moving object follows as a function of time. Each point thus consists of a trajectory ID, location (including *latitude*, and *longitude*), and timestamp.

I define *Trajectory Slots* to denote subsets of trajectories in equal time intervals. Each trajectory slot consists of moving objects from all trajectories within the time period of $T$. These objects are of different timestamps. For the visualization simplicity, Figure 6.1 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 $x\text{-}th$ point of object $o_j$, contains the location information as well as the 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.

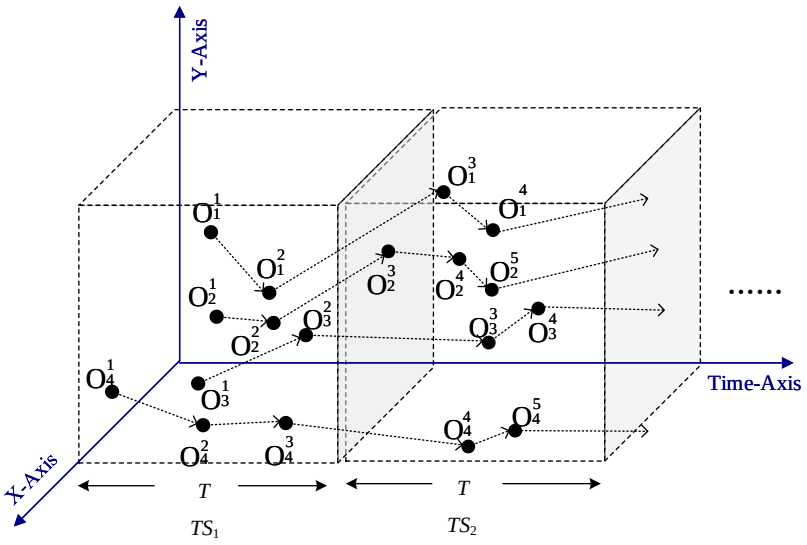



Figure 6.1: Trajectory Slot Model

## 6.2 Definitions and Notations

Table 6.1 summarizes the symbols to be used in this model.

Table 6.1: Gathering: Commonly used symbols

| Symbol | Description |
|---|---|
| $TS$ | trajectory Slot |
| $\mathcal{O}$ | set of moving objects |
| $o_i$ | $i$-$th$ object in $\mathcal{O}$ |
| $o_j^x$ | $x$-$th$ 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) |
| $\epsilon$ | distance threshold |
| $\mu$ | density threshold |
| $\ell$ | companion size threshold |
| $k$ | companion time duration threshold |

### Concept Definition

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

**Definition 4.** ***(Slot Trajectory Coverage)****: Let $\mathcal{O}_{TS}$ be the object set in a trajectory slot $TS$, $\epsilon$ be the distance threshold, and $o_i, o_j \in \mathcal{O}_{TS}$. $o_j$ is a slot trajectory coverage for $o_i$ denoted by $o_j \rhd o_i$ 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 6.2.*

I 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 5.** ***(Coverage Density Reachable)****: $o_i \in \mathcal{O}_{TS}$, the coverage set of $o_i$ contains each object that is a slot trajectory coverage for $o_i$, denoted as $C_s(o_i) = \{o_j \in \mathcal{O}_{TS} | o_j \rhd o_i\}$. Let $\epsilon$ be the distance threshold and $\mu$ be the density threshold, object $o_i$ is coverage density reachable from $o_j$, if $|C_s(o_i)| \geq \mu$.*

In Figure 6.1, assume $\epsilon$-neighbourhood of $o_2$ covers $o_3$ within $TS_1$. According to Definition 4, I compute $o_2 \rhd o_3$. Therefore, $o_3 \in C_s(o_2)$. Likewise, assume the $\epsilon$-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 $\mu$ is set to be 2, $|C_s(o_2)| >= \mu$ and thus $o_2$ is coverage density reachable from $o_1$, $o_3$.

**Definition 6.** ***(Coverage Density Connection)****: For $o_i \in \mathcal{O}_{TS}$, 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 7.** ***(Trajectory Companion)****: Let $k$ be the duration threshold, and $\ell$ 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 3.1.

### Distance Metrics

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

### 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

$$v = \begin{bmatrix} y_m - y_{m-1} \\ -(x_m - x_{m-1}) \end{bmatrix}, \tag{2}$$

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

$$r = \begin{bmatrix} x_{m-1} - x_p \\ y_{m-1} - y_p \end{bmatrix}, \tag{3}$$

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

$$d_m(s_i, p_j) = |\hat{v} \cdot r| = \frac{|(x_m - x_{m-1})(y_{m-1} - y_p) - (x_{m-1} - x_p)(y_m - y_{m-1})|}{\sqrt{(x_m - x_{m-1})^2 + (y_m - y_{m-1})^2}} \tag{4}$$

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,

$$D(o_i, o_j) = \min_{m \in \mathcal{I}} \{d_m(s_i, p_j)\}. \tag{5}$$

**Polyline-to-Polyline Approach**

Similar to the point to polyline approach, the polyline to polyline (PL2PL) approach also partitions the trajectories into time slots. I 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

$$x^{(p)} = \begin{cases} x_1^{(p)} + \frac{x_2^{(p)} - x_1^{(p)}}{t_2 - t_1} \cdot (t - t_1) & (t_1 \leq t < t_2) \\ x_2^{(p)} + \frac{x_3^{(p)} - x_2^{(p)}}{t_3 - t_2} \cdot (t - t_2) & (t_2 \leq t < t_3) \\ ... & \\ x_{n-1}^{(p)} + \frac{x_n^{(p)} - x_{n-1}^{(p)}}{t_n - t_{n-1}} \cdot (t - t_{n-1}) & (t_{n-1} \leq t < t_n) \end{cases} \quad (1 \leq p \leq d). \tag{6}$$

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 [22], this distance should include two parts: spatial distance and temporal distance. In our study, I employ the distance function that considers both space and time factors below.

$$F_\alpha(o_i^{t_a}, o_j^{t_b}) = \sqrt{SpatialDist^2(o_i^{t_a}, o_j^{t_b}) + \alpha \cdot TemporalDist^2(t_a, t_b)}, \tag{7}$$

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

$$SpatialDist(o_i^{t_a}, o_j^{t_b}) = \sqrt{\sum_{p=1}^{d} (x^{(p)} - x'^{(p)})^2} \tag{8}$$

where $x^{(p)}$ and $x'^{(p)}$ are the $p$-th dimension coordinates of $o_i$ and $o_j$ respectively, which can be obtained by Equation 6. $TemporalDist(.,.)$ is a normalized time-distance function, applied only within a time slot $T$, where temporal distance indicates how close between $t_a$ and $t_b$. $\alpha$ $(0 \leq \alpha \leq 1)$ indicates the weight of time factor. If $\alpha$=1, it implies the space and time factors have the same weight. If $\alpha$=0, it means the time factor is ignored. The value of this distance is evaluated as

$$TemporalDist(t_a, t_b) = \begin{cases} \frac{|t_a - t_b|}{T} & |t_a - t_b| \leq T \\ DNE & |t_a - t_b| > T \end{cases} \tag{9}$$

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

$$D(o_i, o_j) = \min_{t_a, t_b \in \mathcal{I}} \{F_\alpha(o_i^{t_a}, o_j^{t_b})\} \tag{10}$$

## 6.3 Implementation

I define a two-phase trajectory processing framework to address the remaining characteristics of stream processing and parallelism in Table 3.1.

- **The coverage density connection discovery phase.** In this phase, I first utilize the trajectory slot model to set up the unit time $T$ of a slot. I 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 I find all the coverage density connections in each sub-partition.

- **The trajectory companion generation phase.** In this phase, I 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 Figure 6.2. In our framework, the procedure includes 4 steps:

- *I : Trajectory partition*–data within each trajectory slot are partitioned into $n$ sub-partitions;

- *II : Find coverage density reachable*–find coverage density reachable for each object in every sub-partition;

- *III : Find coverage density connection*–find all coverage density connections in each sub-partition;

- *IV : Merge*–find coverage density connections in different sub-partition that have same objects and merge them.

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

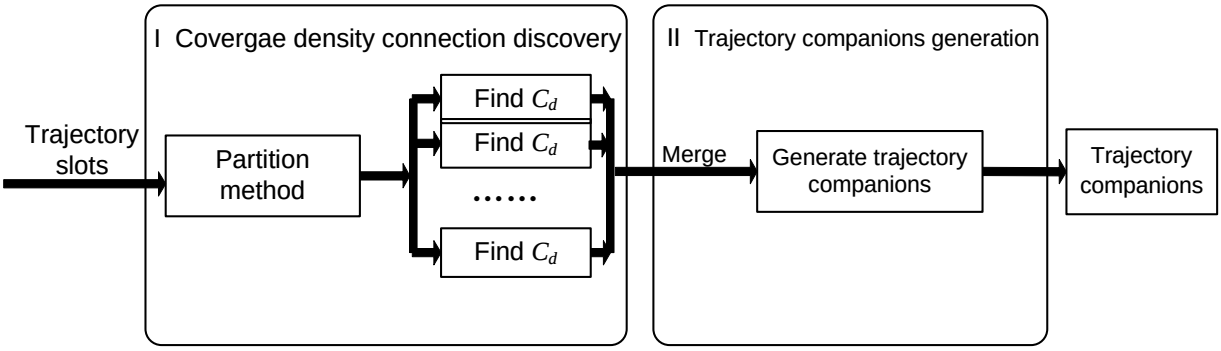



Figure 6.2: TCompanion Two-phase framework

### 6.3.1 Load-balanced Trajectory Partition

When partition the trajectories, I aim to balance the load on each partition in term of the number of objects in each partition. I apply the K-D tree [11] indexing technique. In the K-D tree, there exist nearly equal amount of data in the tree nodes. The steps is illustrated as Algorithm 4.

**Algorithm 4:** K-D Tree Based Partition

**Input :** trajectory data $\mathcal{T}$ in a slot, object set $\mathcal{O}$, number of partitions $n$
**Output:** $\{P_1, ..., P_n\}$ and $\{PL_1, ..., PL_n\}$

```
1  m ⟵ 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 ⟵ 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 vu in T do
14     if vu and i-th region have common points then
15         Put vu into PL_i
16 Return P_i and PL_i
```

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 6.5.

### 6.3.2 Parallel Discovery of Coverage Density Connection

Algorithm 5 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 4.

**Algorithm 5:** 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
```

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, I omit the 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 $L_j$).

I assure the segment pairs within the 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, I can use $reduceByKey()$ function to put segment pairs with the same key together. Finally I 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, I introduce two pruning rules.

**Pruning Rule I**: If the shortest distance between the polyline of $o_i$ and the polyline of $o_j$ is larger than $\epsilon$, 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 $\epsilon$ 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 $\min_{t_a,t_b\in\mathcal{I}}\{\sqrt{\alpha \cdot TemporalDist^2(t_a,t_b)}\} \leq D(o_i,o_j)$, $\min_{t_a,t_b\in\mathcal{I}}\{\sqrt{\alpha \cdot TemporalDist^2(t_a,t_b)}\} > \epsilon \Longrightarrow D(o_i,o_j) \geq \epsilon$.

**Algorithm 6:** Polyline to Polyline Parallel Distance Calculation

**Input :** a set of polylines $\mathcal{L}$ within a time slot $[t_i, t_{i+T}]$
**Output:** a set of polyline density reachable pair $R$

$\mathcal{S} \leftarrow \emptyset$ // a set of polyline segments
$\mathcal{SP} \leftarrow \emptyset$ // a set of polyline segment pairs
$D_{pl} \leftarrow \emptyset$ // a set of polylines distance pair $< (L_i, L_j), d >$
$D\prime_{pl} \leftarrow \emptyset$ // a set of polylines minimum distance pair $< (L_i, L_j), d_{min} >$
**for** *each polyline in* $\mathcal{L}$ **do**
    **for** *each segment* $s \in pl$ **do**
        calculate coefficient of the linear equation
        add $s$ into $\mathcal{S}$
$SP \leftarrow$ find all segment pairs from $\mathcal{S}$
**Map**$(<< i, j >, < s_i^f, s_j^g >>)$
**for** *each sub-partition* $\mathcal{P}$ **do**
    **for** *each segment pair* $< s_i^f, s_j^g >$ *in* $\mathcal{P}$ **do**
        bound segments with $MBR_i^f$ and $MBR_j^g$
        $d \rightarrow dist_{min}(MBR_i^f, MBR_j^g)$
        calculate the minimum distance between $MBR_i^f$ and $MBR_j^g$ // Pruning Rule I
        **if** $d > \epsilon$ **then**
            prune segment pair $< s_i^f, s_j^g >$
        // Pruning Rule II
        **if** $|MBR_i^f.starttime - MBR_j^g.endtime| > \epsilon \wedge |MBR_i^f.endtime - MBR_j^g.starttime| > \epsilon$ **then**
            prune segment pair $< s_i^f, s_j^g >$
        calculate exact distance between two segments as $dist(s_i^f, s_j^g)$
**Reduce**$(< i, j >, d_{min} = \min[dist(s_i^f, s_j^g), ...])$
**for** *each polyline pair* **do**
    **if** $d_{min} < \epsilon$ **then**
        add $(L_i, L_j)$ to $R$
Return $R$

### 6.3.3 Trajectory Companions Generation on Streaming Data

In Phase II, I need to find all possible combination of objects in order to obtain trajectory companions. Consider this example, given a slot-objects key-value pair $< 1, \{1, 2, 3\} >$ read as objects 1,2,and 3 are density connected at slot 1. our goal is to generate all subsets of the set $\{1, 2, 3\}$ such that $\{\}\{1\}$,$\{1, 2\}$,$\{1, 2, 3\}$,$\{2\}$,$\{2, 3\}$,$\{1, 3\}$. The algorithm is described in Algorithm 7 and its complexity yields $O(2^n)$. If a density connections in a slot contains 20 objects, it requires more than 1 million iterations. This computation incurs high cost and definitely a performance bottleneck.

**Algorithm 7:** Power set generation

**Input :** a set of object IDs $S$, size threshold $l$
**Output:** power set $P(S)$

```
1  R ← ∅
2  n ← 1 ≪ S.size
3  for each object o_i in a sub-partition do
4      pos ← 0
5      bitmask ← i
6      s ← ∅
7      do
8          if bitmask ∧ 1 = 1 then
9              add pos into s
10         bitmask ≫ 1
11         increase pos by 1
12     while bitmask > 0;
13     if s.size >= l then
14         add s into R
```

Technically speaking, the power set computation can be parallelized. The idea I have so far is to distribute the set to a cluster such that each executor computes a subset size of $k$. Next, I merge the result together to form a power set. Continuing on our example, a given set $\{1, 2, 3\}$ (where $k$=3) can be mapped into $< \{1, 2, 3\}, 1 >$, $< \{1, 2, 3\}, 2 >$, $< \{1, 2, 3\}, 3 >$. The second integer in the tuple indicates the size of subset, meaning that should generate $\{1, 2\}$, and $\{2, 3\}$ for $< \{1, 2, 3\}, 2 >$. Algorithm 8 illustrates the steps to discover $k$ size subsets. The complexity of the algorithm is $O(n^k)$. One of the drawbacks for the algorithm is that it incurs imbalanced workload for various $k$. Suppose that $n$=10, computing the subsets of size 3 needs 1000 iterations, whereas size of 6 needs 1 million iterations. As result, some executors would have heavy workload while

the others are idle.

**Algorithm 8:** Power set generation of size $k$

**Input :** a set of object IDs $S$, size $k$, index $i$, current set $c$, final set $R$
**Output:** subset of the power set $P(S)$

```
if c.size == k then
    add c into R
    return
if i == S.size then
    return
m ← position of i in S
add m into c
return self(S, k, i+1, c, R)
remove m from c
return self(S, k, i+1, c, R)
```

I 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, I introduce a new concept of *promising companion candidate*.

**Definition 8.** ***(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 $\ell$.*

According to this definition, I 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 I 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 $\mathcal{DC}$. I 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. I conclude the Online Discovery Algorithm in Algorithm 4.

**Algorithm 4:** Online Discovery Algorithm

**Input :** $\{\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_i, ...\}$ and $k$
**Output:** trajectory companions

```
for each T_i (not empty) do
    if i mod k ≠ 0 then
        Call Algorithm III to discover ∀c_d within T_i
        Put ∀c_d into a set DC
    else
        Find p_c in DC
        Keep p_c into memory
        Call Algorithm III to discover ∀c_d in T_i
        Put ∀c_d into a set DC
        Delete ∀c_d within T_{i-k} from DC
        Merge p_c and ∀c_d in T_i into trajectory companions
        Return trajectory companions
```

## 6.4 Merging Methods and Analysis

To discover promising companion candidates and trajectory companions, I need to find the same objects from all the connections and intersect them. Assume an average $M$ coverage density connections in each slot and I 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, I propose intersecting and merging methods and formally analyze their effectiveness to improve the runtime performance on the Spark Streaming platform. In particular, I 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. I introduce an example below to best analyze the method as follows in section 6.4.1 to 6.4.4.

**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.

### 6.4.1 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 $|\mathcal{C}|$ is the average number of coverage density connections that contain the same object in one slot. The number of pairs equals to $n \cdot |\mathcal{C}| \cdot |\mathcal{O}|$, where $|\mathcal{O}|$ 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.

### 6.4.2 Self-cartesian Set Method

Cartesian operation returns the cross product of two RDDs or RDD to its identical self. Given the above example, I 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 I ignore the self-contained pair. Similarly, the same procedure repeats for the rest of rows. Therefore, the total cost of data shuffling, $\Gamma_1$, is calculated as

$$\Gamma_1 = (|\mathcal{DC}| - 1) \cdot |\mathcal{DC}| \simeq |\mathcal{DC}|^2. \tag{11}$$

Next, I intersects each pair $(c_{di}, c_{dj})$. I 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.

### 6.4.3 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, I then compute $c_{di} \cdot \mathcal{DC} = \big\{(c_{di}, c_{dj}) | i \neq j \text{ and } c_{dj} \in \mathcal{DC}\big\}$. In our example, I 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, $\Gamma_2$, is calculated as two rounds transmission of $\mathcal{DC}$ among $w$ worker nodes such that,

$$\Gamma_2 = 2 \cdot w \cdot |\mathcal{DC}|. \tag{12}$$

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 $\mathcal{DC}$.

### 6.4.4 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. I 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 $\beta$ ($0 < \beta \leq 1$) denotes the portion of the coverage density connections that need to be shuffled,

$$\Gamma_3 = \beta \cdot |\mathcal{DC}|. \tag{13}$$

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

## 6.5 Evaluation

In this section, I experimentally evaluate the performance of the proposed solutions on a real dataset. In order to compare with our competitor, I are interested in the quality attributes

*Precision and Recall*: I analyze the precision, recall and F1-score in response to the size of time slot.

*Efficency*: I analyze the performance of gathering, TCompanion-P2PL and TCompanion-PL2PL measured in terms of throughput and latency.

Next, I focus on comparing P2PL and PL2PL algorithms by quality attributes

*Scalability*: I analyze the throughput in response to the size of the cluster.

*Data Intensity*: I are interested in data shuffle rate response to different size of data per time slot.

*Stability*: I perform a stability analysis by observing the processing time and scheduling delay versus input rate.

Table 6.2 shows the parameter settings.

Table 6.2: TCompanion: Parameter settings

| Factor | Range | Default |
|---|---|---|
| $\epsilon$ | $\{0.00005, 0.0001, 0.0005, 0.001, 0.005\}$ | 0.0001 |
| $n$ | $[2, 8]$ | 2 |
| $T$ | $[40, 80]$ | 60 |

**Precision and Recall Evaluation**

In Section 4.4.2, I used 12 selected sample trajectories and the predefined ground truth results to evaluate the precision and recall for each algorithm. Suppose I 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

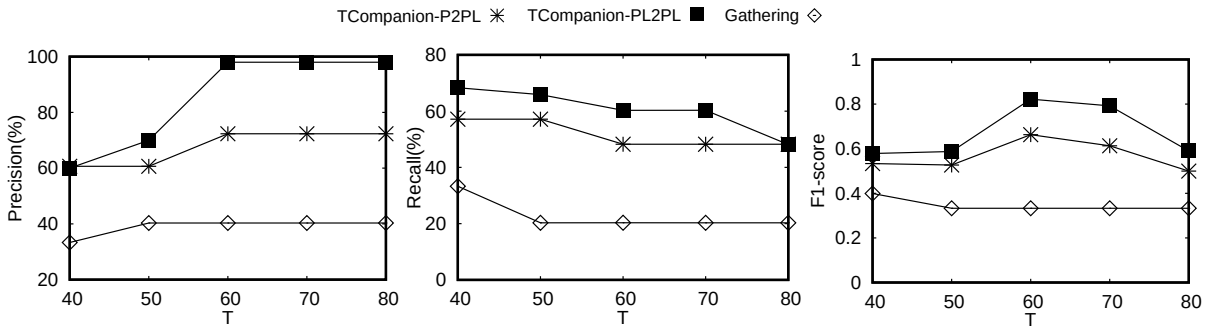
$$Precision(x) = \frac{|\tilde{TC}(x) \cap TC(x)|}{|\tilde{TC}(x)|} \times 100\% \tag{14}$$

Also, the recall is computed as

$$Recall(x) = \frac{|\tilde{TC}(x) \cap TC(x)|}{|TC(x)|} \times 100\% \tag{15}$$

Lastly, the F1-score is computed as

$$F_1(x) = \frac{Precision(x) \times Recall(x)}{Precision(x) + Recall(x)} \times 100\% \tag{16}$$




(a) Precision (b) Recall (c) F1-score

Figure 6.3: Gathering vs. TCompanion: Precision, Recall and F1-score

Figure 6.3 shows the precision, recall and F1-score of our proposed algorithm (TCompanion) and the competitor algorithm (Gathering)[35],[40]. I 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. I 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 recall and higher F1-score. When $T = 60$, the F1-score is highest. I set 60 seconds as default value in the following experiments.

**Performance Comparisons**

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

$$\text{throughput} = \frac{\text{total number of locations}}{\text{procesing time}}$$

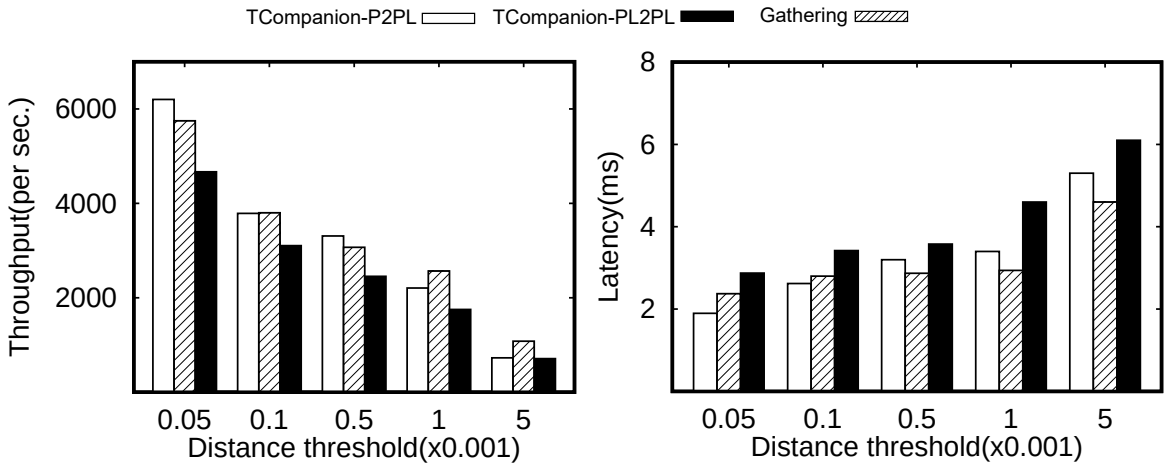
and the average latency of processing each location record is

$$\text{latency} = \frac{\text{procesing time+waiting time}}{\text{total number of locations}}$$

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, I set $n = 2$ for the performance evaluation experiments.

Figure 6.4 illustrates results by varying the distance threshold $\epsilon$. Both throughput and average latency of the two algorithms are comparable. TCompanion has more distance computation since it considers all the timestamps within one time slot. I optimize the runtime performance of TCompanion by reducing the data shuffling cost (see section 6.4).

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, I focus on experiments in the following sections to further identify key factors contributing to system level quality attributes.

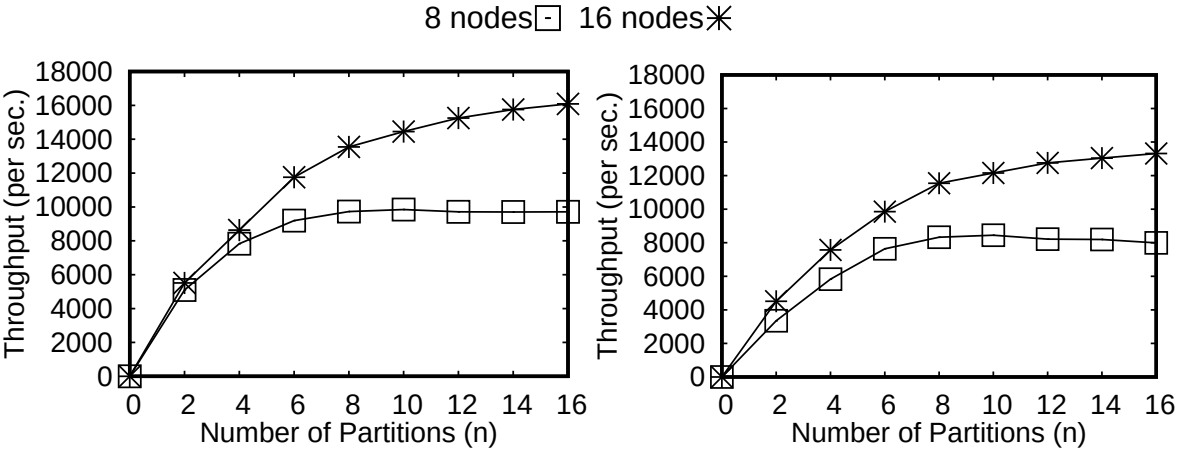



(a) throughput (b) latency

Figure 6.4: TCompanion: Throughput and latency comparison

**Scalability Evaluation**

I 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, I deploy 8 to 16 nodes. Figure 6.5 plots the throughput under different size of the cluster.




(a) P2PL (b) PL2PL

Figure 6.5: TCompanion: Scalability Comparison

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 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 TCompanion with both distance metrics scale as the number of executor nodes and the number of partitions grow together.

### Stability Evaluation

I 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.

I inject the entire dataset to 30 windows into the workflow shown in Figure 6.7(a). I 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, I set the batch interval as 10 seconds. In Figure 6.7(b) and (c), I set a horizontal line at 10 second indicating the stability threshold.

Over the span of 300 seconds (5 minutes), I 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 Figure 6.7(a), I 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 Figure 6.7(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 the 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 underlying 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, which remains our future work.

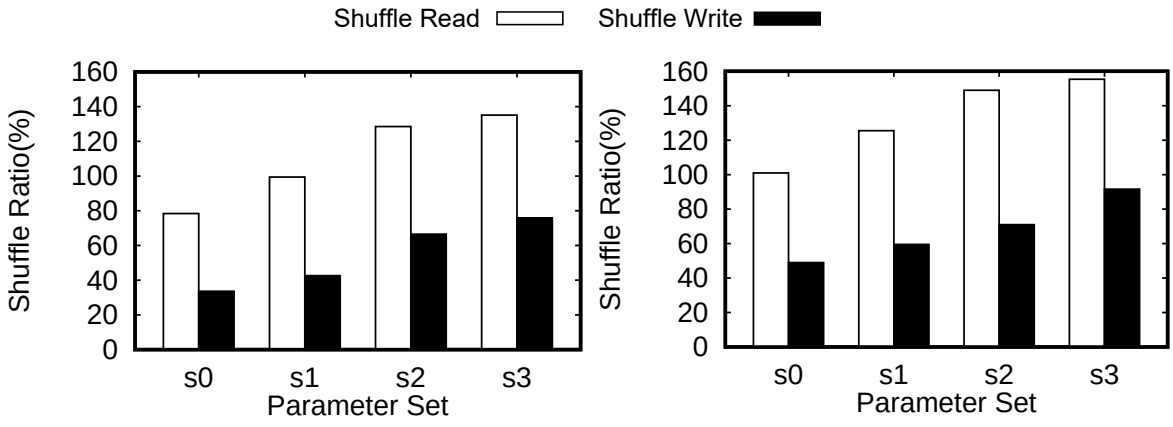
### Data Intensity Evaluation

In this experiment, I 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. I tune four set of parameters, shown in Table 6.3 to obtain different size of data to process per time slot. For example, when I increase the distance threshold ($\epsilon$) 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. I also decrease density threshold ($\mu$) and size threshold ($l$). This increases the data density in the companion discovery phase of the algorithm.

Table 6.3: TCompanion: Parameter settings of Data Intensity Evaluation

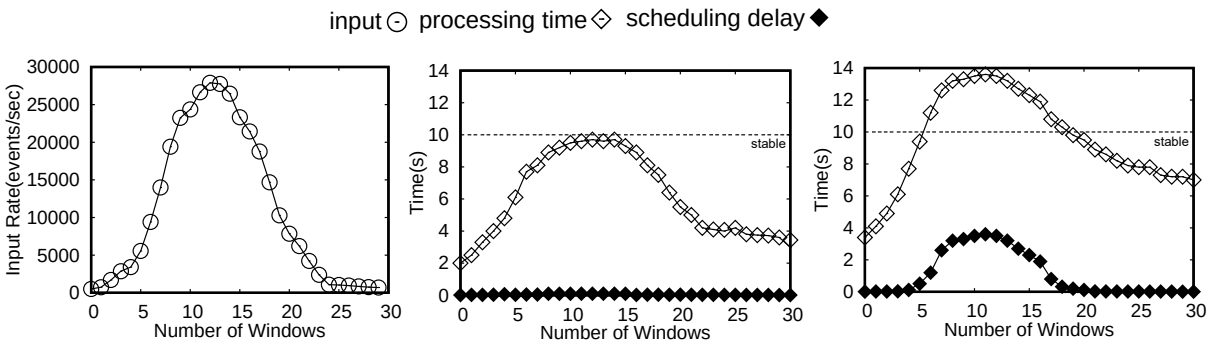
| **Parameter Set** | $\epsilon$ | $k$ | $l$ | $n$ | $T$ | $\mu$ |
|---|---|---|---|---|---|---|
| $S1$ | 0.001 | 3 | 3 | 8 | 60 | 3 |
| $S2$ | 0.005 | 3 | 3 | 8 | 60 | 3 |
| $S3$ | 0.005 | 3 | 3 | 8 | 100 | 3 |
| $S4$ | 0.005 | 3 | 2 | 8 | 100 | 2 |

As Figure 6.6 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 Figure 6.4 in section 6.5).




(a) P2PL (b) PL2PL

Figure 6.6: TCompanion: Data Shuffling Comparison




(a) input rate (b) P2PL response (c) PL2PL response

Figure 6.7: TCompanion: Stability Comparison

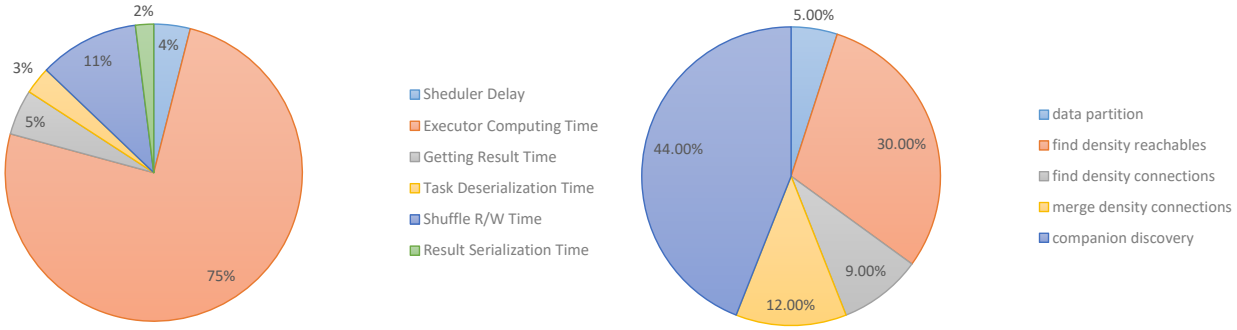
### Execution Time Decomposition

I further decompose the execution time of TCompanion(PL2PL) to understand which steps in the workflow contribute most to the time cost. Figure 6.8 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 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 6.8 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 9 line 3). Its complexity yields $O(2^n)$.

### Effect of Parameters on TCompanion

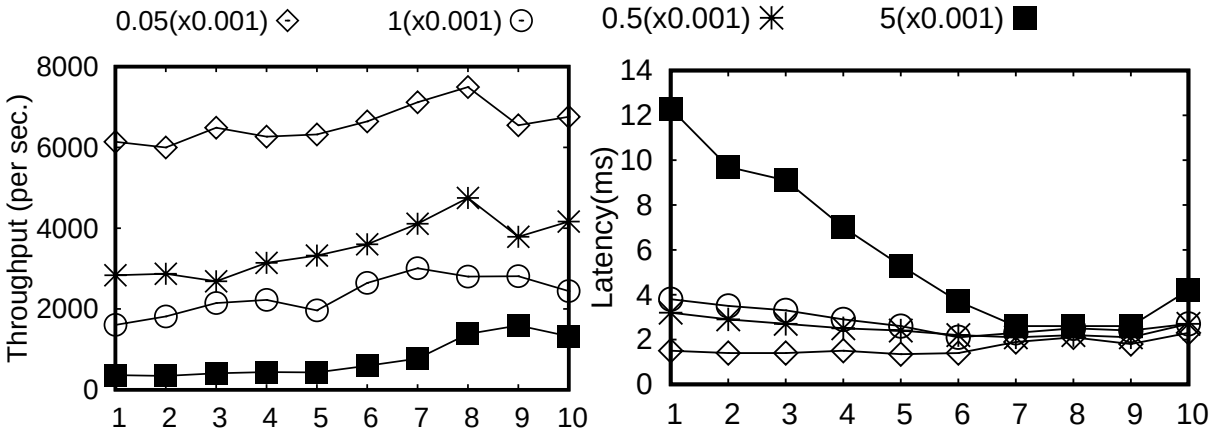
I analyze the performance of TCompanion (PL2PL) under parameter settings since the distance metrics of polyline-to-polyline produces higher precision. I run the algorithm in 10 windows with each duration of 60s that is in total 10 minutes. I tune the distance threshold $\epsilon$ and observe its effects on throughput and latency. I vary the distance threshold $\epsilon$ from 0.00005 to 0.005, with the geospatial meaning of 10 to 100 meters. Other parameters use default settings. Figure 6.9 shows the throughput is higher and the latency is lower by decreasing $\epsilon$. One reason is lager $\epsilon$ covers more objects, thus




(a) Spark's metrics (b) All steps in the workflow

Figure 6.8: TCompanion: Scalability Comparison

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.




(a) throughput (b) latency

Figure 6.9: TCompanion: Vary $\epsilon$ from 0.00005 to 0.005

# Chapter 7

# Conclusion

In this thesis, I devise two parallel discovery methods called Gathering and 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 frameworks.

This work first designed a snapshot based parallel framework to discover gatherings on different types of trajectory data. I focus on discussing the Spark workflow for each phase of Gathering Discovery Framework including snapshot clustering, crowd detection, and gathering generation. To improve performance, I proposed some optimization techniques with Apache Spark and Spark Streaming respectively. Experimental results demonstrated the effectiveness of the proposed methods and techniques on a public dataset in Amazon EC2 clusters.

I also propose a slot based trajectory companions discovery algorithm (TCompanion) that contains both spatial and temporal functions to measure distances between trajectories over continuously updated streaming windows. Its implementation focus on load balanced workload as well as analyzing an optimal merging methods to reduce data shuffling. In experiments, TCompanion 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.

# Appendix A

# Cluster Configuration and Applications Deployment

This appendix is served as a step-by-step guidance of the environment setup of Amazon Web Service (AWS) EC2 cluster, as well as deployment of our applications to the cluster. Technically, our applications including Data Generator and Analytic Pipeline can be deployed on any public cloud Infrastructure-as-a-Service providers (such as Google Cloud Platform[1], Azure[2], and Digital Ocean[3]) or even private cloud. Here, I only focus on the description of the configuration and deployment steps dedicated to this thesis.

### Configuring Apache Spark Cluster on AWS EMR

In AWS, Apache Spark can be installed alongside the other Hadoop application available in Amazon EMR, and it can also leverage the EMR file system to directly access data in Amazon S3. In this section, I will describe the procedure to set up a Spark cluster via Amazon EMR. The following steps is based on the precondition that an AWS account is already created. If you don't have an AWS account, you will need to create one before you will be able to proceed.

(1) Create Amazon EC2 key pair to connect to the nodes in the cluster over a secure channel using

[1] https://cloud.google.com/solutions
[2] https://azure.microsoft.com
[3] https://www.digitalocean.com

the Secure Shell (SSH) protocol.. Detailed steps can be found in AWS EC2 User Guide[4].

(2) Instantiate Apache Spark clusters via AWS EMR. I will need to create two clusters, data generator cluster ($C_g$), and analysis pipeline cluster ($C_a$) respectively. Table A.3 illustrates the cluster configuration. Detailed steps can be found in Amazon EMR Management Guide[5].

| Category | Option | Value |
|---|---|---|
| General configuration | Cluster name | *My Cluster* |
| | Launch mode | *Cluster* |
| Software configuration | Vendor | *Amazon* |
| | Release | *emr-4.2.0* |
| | Applications | *Spark: Spark 1.5.1 on Hadoop 2.6.0 YARN with Ganglia 3.6.0* |
| Hardware configuration | Instance type | *t2.large* |
| | Number of instances | *Data Generator: 1; Analysis Pipeline: 9* |
| Security and access | EC2 key pair | *(Select the key pair name defined in Step 1)* |

Table A.1: AWS - EMR Configurations

Now, I should have two clusters running on AWS. The public DNS or ip address of the master nodes can be found on the EMR cluster list. Let the ip address of Data Generator master be $ip_g$, Analytic Pipeline master be $ip_a$. I need these ip addresses to remote access to the instances in the next step.

### Configuring and Deploying Data Generator

The Data Generator is written in Java. The complete source code can be found on the GitHub repository[6] and the driver code of Spark are attached in Appendix B. The application dependencies is managed by Maven. Table A summarizes the maven dependencies.

| groupId | artifactId | version |
|---|---|---|
| *org.apache.kafka* | *kafka_2.10* | *0.9.0.1* |
| *org.apache.spark* | *spark-core_2.10* | *1.6.1* |
| *org.slf4j* | *slf4j-api* | *1.7.10* |
| *org.slf4j* | *slf4j-simple* | *1.7.10* |
| *junit* | *junit* | *4.11* |
| *org.cloudera.spark.streaming.kafka* | *spark-kafka-writer* | *0.1.0* |

Table A.2: Data Generator - Maven Dependencies

The following steps show how to build, configure, and run the Data Generator:

[4]http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html
[5]http://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-gs.html
[6]https://github.com/samsonxian/Trajectory-Companion-Data-Generator

(1) SSH connect to the master node of $C_g$ with $ip_g$. More detail related to how to connect to the Master Node Using SSH refer to Amazon EMR Management Guide [7].

(2) On Master Node of cluster $C_g$, following the instruction described in the README file.

### Configuring and Deploying Trajectory Pattern Analytic Piplines

The Trajectory Pattern Analytic Piplines are written in Java. The complete source code can be found on the GitHub repository[8] and the driver code Gathering and TCompanion are attached in Appendix B, and Appendix C respectively. The application dependencies is managed by Maven. Table A summarizes the maven dependencies.

| groupId | artifactId | version |
|---|---|---|
| *org.apache.kafka* | *kafka_2.10* | *0.9.0.1* |
| *org.apache.spark* | *spark-core_2.10* | *1.5.1* |
| *org.apache.spark* | *spark-streaming_2.10* | *1.5.1* |
| *org.apache.spark* | *spark-streaming-kafka_2.10* | *1.5.1* |
| *log4j* | *log4j* | *1.2.17* |
| *org.apache.commons* | *commons-math3* | *3.1.1* |

Table A.3: Trajectory Pattern Analytic Pipeline - Maven Dependencies

The following steps show how to build, configure, and run the Trajectory Pattern Analytic Piplines:

(1) SSH connect to the master node of $C_a$ with $ip_a$.

(2) On Master Node of cluster $C_a$, following the instruction described in the README file.

[7] http://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-connect-master-node-ssh.html
[8] https://github.com/samsonxian/Trajectory-Companion-Finder

# Appendix B

# Source code for Trajectory Data Stream Generator

```
public static void main(String[] args) throws Exception {
    if(args.length < 2) {
        System.err.println("USAGE: <propsfile> <inputfile> [debug]");
        System.exit(1);
    }

    boolean isDebug = Arrays.asList(args).contains("debug");

    // Setup the property parser
    PropertyFileParser propertyParser = new PropertyFileParser(args[0]);
    propertyParser.parseFile();

    int messageRate = Integer.parseInt(propertyParser.getProperty
            (Config.KAFKA_PRDOUCER_MESSAGE_RATE));

    // Kafka producer
    Properties props = new Properties();
    props.put("metadata.broker.list",
            propertyParser.getProperty(Config.KAFKA_HOSTNAME_KEY)+":"+
                    propertyParser.getProperty(Config.KAFKA_PORT_KEY));
    props.put("serializer.class", "kafka.serializer.StringEncoder");
    props.put("key.serializer.class", "kafka.serializer.StringEncoder");
    props.put("request.required.acks", "1");
```

```
        // Spark config
        SparkConf sparkConf = new SparkConf().
                setAppName("kafkaDataGenerator");
        if(isDebug) sparkConf.setMaster("local[*]");

        JavaSparkContext ctx = new JavaSparkContext(sparkConf);
        JavaRDD<String> lines = ctx.textFile(args[1]);

        // timestamp in ascending order
        JavaPairRDD<Integer, String> orderedDataRDD =
        lines.mapToPair(new TimestampMapper())
                .sortByKey().cache();

        // get all timestamps
        JavaRDD<Integer> timestampRDD = orderedDataRDD.keys().distinct();
        Set<Integer> timestamps = new TreeSet(timestampRDD.toArray());

        // iterate each timestamp
        for (int timestamp: timestamps) {

            // grab all trajectories per timestamp
            JavaRDD<String> timeDataRDD =
            orderedDataRDD.filter(new TimestampFilter(timestamp))
                    .map(new TrajectoryMapper());

            try{
                JavaRDDKafkaWriter<String> writer = JavaRDDKafkaWriterFactory.fromJavaRDD(timeDataRDD);
                writer.writeToKafka(props, new KafkaSparkProc(
                        propertyParser.getProperty(Config.KAFKA_TOPIC_KEY)));
                Thread.sleep(messageRate);
            }
            catch(Exception e)
            {
                System.err.println(e.getMessage());
                System.exit(1);
            }
        }

        ctx.close();
    }
```

# Appendix C

# Source code for Snapshot Model

```
public class StreamingGPFinder {

 private static String outputDir = "";
 private static double distanceThreshold = 0.01;      // eps
 private static int densityThreshold = 2;             // mu
 private static int timeInterval = 60;                // delta t
 private static int lifetimeThreshold = 100;          // kc
 private static int clusterNumThreshold = 3;          // kp
 private static int participatorNumThreshold = 2;     // mp
 private static int numSubPartitions = 2;
 private static double gridsize = 0.1;               // g
 private static boolean debugMode = false;

 public static void main(String[] args) throws Exception {

     // setup the cli parser
     SGPCliParser parser = new SGPCliParser(args);
     parser.parse();

     if(parser.getCmd() == null) {
         System.exit(1);
     }

     final UserData data = new UserData();
     initParams(parser, data);

     // setup the property parser
```

```
PropertyFileParser propertyParser = new PropertyFileParser(args[0]);
propertyParser.parseFile();
Map<String, String> kafkaParams = new HashMap<>();
kafkaParams.put("metadata.broker.list", propertyParser.getProperty(Config.KAFKA_BROKERS));

SparkConf sparkConf = new SparkConf()
        .setAppName("StreamingGPFinder");
if(debugMode) sparkConf.setMaster("local[*]");

int batchInterval = Integer.parseInt(propertyParser.getProperty(Config.SPARK_BATCH_INTERVAL));
final JavaStreamingContext ssc = new JavaStreamingContext(sparkConf, Durations.seconds(batchInterv
ssc.checkpoint(propertyParser.getProperty(Config.SPARK_CHECKPOINT_DIR));
Set<String> topics = new HashSet(Arrays.asList(
        propertyParser.getProperty(Config.KAFKA_TOPICS).split(",")));

// create direct kafka stream with brokers and topics
JavaPairInputDStream<String, String> inputDStream =
        KafkaUtils.createDirectStream(
                ssc, String.class, String.class,
                StringDecoder.class, StringDecoder.class,
                kafkaParams, topics);

JavaPairDStream windowedInputRDD = inputDStream.window(
        Durations.seconds(batchInterval * 3),
        Durations.seconds(batchInterval * 2));
JavaDStream<String> lines = windowedInputRDD.map(new InputDStreamValueMapper());
lines.foreach(new BatchCountFunc());

// find snapshot per timestamp and data partition
// format: <timestamp, {point}>
JavaPairDStream<Integer, TCPoint> snapshotDStream =
        lines.mapToPair(new SnapshotMapper());

snapshotDStream.foreach(new Function2<JavaPairRDD<Integer,TCPoint>, Time, Void>() {
    @Override
    public Void call(JavaPairRDD<Integer, TCPoint> snapshotRDD, Time time) throws Exception {

        // data partition
        // K = <gridId, timestamp>
        // V = {point}
        FixedGridPartition fgp = new FixedGridPartition(gridsize);
```

```
JavaPairRDD<String, Iterable<TCPoint>> partitionRDD = fgp.apply(snapshotRDD);

// find clusters - find clusters (DBSCAN) in each sub-partition
// K = timestamp
// V = gid, cluster
JavaPairRDD<Integer, Tuple2<String, DBSCANCluster>> clusterRDD =
        GPQuery.getClusterRDD(partitionRDD, data);

// merge clusters - find clusters in different partition that have same objects
JavaPairRDD<Integer, Tuple2<String, DBSCANCluster>> mergedRDD =
        GPQuery.getMergedClusterRDD(clusterRDD, data);

// group clusters by timestamp to form a crowd, given each crowd
// an unique id
// K = <gid, crowd>
// V = crowdId
JavaPairRDD<Tuple2<String, Crowd>, Long> crowdRDD =
        GPQuery.getCrowdRDD(mergedRDD, data).cache();

// find participator
// K = crowdId
// V = {participator}
JavaPairRDD<Long, Iterable<Integer>> participatorRDD =
        GPQuery.getParticipatorRDD(crowdRDD, data);

// convert crowd into the same format as participator
// format: <crowdId, {(objectId, timestamp)}>
JavaPairRDD<Long, Iterable<Tuple2<Integer, Integer>>> crowdToObjectTimestampRDD =
        crowdRDD.flatMapToPair(new CrowdToObjectTimestampPairMapper());

// discover gatherings
// format: <crowdId, {<timestamp, {objectId}>}>
JavaPairRDD<Long, Iterable<Tuple2<Integer, Iterable<Integer>>>> gatheringRDD =
        GPQuery.getGatheringRDD(crowdToObjectTimestampRDD, participatorRDD,
                data);

if(outputDir.isEmpty())
    gatheringRDD.take(1);
else
    gatheringRDD.saveAsHadoopFile(outputDir,
            String.class, String.class, TextOutputFormat.class);
```

```
                return null;
            }
        });

        ssc.start();
        ssc.awaitTermination();
    }
}
```

# Appendix D

# Source code for Slot Model

```
public class StreamingTCFinder {

 private static String outputDir = "";
 private static double distanceThreshold = 0.0001;    //eps
 private static int densityThreshold = 3;             //mu
 private static int timeInterval = 50;                //T
 private static int durationThreshold = 2;            //k
 private static int numSubPartitions = 1;             //n
 private static int sizeThreshold = 2;
 private static boolean debugMode = false;

 public static void main(String[] args) throws Exception {

     // setup the cli parser
     STCCliParser parser = new STCCliParser(args);
     parser.parse();

     if(parser.getCmd() == null) {
         System.exit(1);
     }
     initParams(parser);

     // setup the property parser
     PropertyFileParser propertyParser = new PropertyFileParser(args[0]);
     propertyParser.parseFile();

     SparkConf sparkConf = new SparkConf()
```

```
        .setAppName("StreamingTCFinder");
if(debugMode) sparkConf.setMaster("local[*]");

int batchInterval = Integer.parseInt(propertyParser.getProperty(Config.SPARK_BATCH_INTERVAL));
JavaStreamingContext ssc = new JavaStreamingContext(sparkConf, Durations.seconds(batchInterval));
ssc.checkpoint(propertyParser.getProperty(Config.SPARK_CHECKPOINT_DIR));

Set<String> topics = new HashSet(Arrays.asList(
        propertyParser.getProperty(Config.KAFKA_TOPICS).split(",")));
Map<String, String> kafkaParams = new HashMap<>();
kafkaParams.put("metadata.broker.list", propertyParser.getProperty(Config.KAFKA_BROKERS));

// create direct kafka stream with brokers and topics
JavaPairInputDStream<String, String> inputDStream =
        KafkaUtils.createDirectStream(
                ssc, String.class, String.class,
                StringDecoder.class, StringDecoder.class,
                kafkaParams, topics);

JavaPairDStream windowedInputRDD = inputDStream.window(Durations.seconds(batchInterval));
JavaDStream<String> lines = windowedInputRDD.
        map(new InputDStreamValueMapper());
lines.foreach(new BatchCountFunc());

// partition the entire common.data set into trajectory slots
// format: <slot_id, { pi, pj,... }>
JavaPairDStream<Long, Iterable<TCPoint>> slotsRDD =
        lines.mapToPair(new stc.TrajectorySlotMapper())
                .groupByKey();

// partition each slot into sub-partitions
// format: <slot_id, TCRegion>
JavaDStream<Tuple2<Long, TCRegion>> subPartitionsRDD =
        slotsRDD.flatMap(new KDTreeSubPartitionMapper(numSubPartitions)).cache();

// get each point per partition
// format: <(slotId, regionId), <objectId, point>>
JavaPairDStream<String, Tuple2<Integer, TCPoint>> pointsRDD =
        subPartitionsRDD.flatMapToPair(new SubPartitionToPointsFlatMapper());

// get all polylines per partition
```

```
// format: <(slotId, regionId), {<objectId, polyline>}
JavaPairDStream<String, Map<Integer, TCPolyline>> polylinesRDD =
        subPartitionsRDD.mapToPair(new SubPartitionToPolylinesMapper());

// get density reachable per sub partition
// format: <(slotId, regionId, objectId), {objectId}>
JavaPairDStream<String, Iterable<Integer>> densityReachableRDD =
        pointsRDD.join(polylinesRDD)
                .flatMapToPair(new CoverageDensityReachableMapper(distanceThreshold))
                .groupByKey().filter(new CoverageDensityReachableFilter(densityThreshold));

// remove objectId from key
// format: <(slotId, regionId), {objectId}>
JavaPairDStream<String, Iterable<Integer>> densityConnectionRDD
        = densityReachableRDD
        .mapToPair(new SubPartitionRemoveObjectIDMapper());

// merge density connection sub-partitions
// format: <(slotId, regionId), {{objectId}}>
JavaPairDStream<String, Iterable<Integer>> subpartMergeConnectionRDD =
        densityConnectionRDD
                .reduceByKey(new CoverageDensityConnectionReducer());

// remove regionId from key
// format: <slotId, {objectId}>
JavaPairDStream<Integer, Iterable<Integer>> slotConnectionRDD =
        subpartMergeConnectionRDD
                .mapToPair(new SlotRemoveSubPartitionIDMapper())
                .reduceByKey(new CoverageDensityConnectionReducer());

JavaPairDStream<Integer, Iterable<Integer>> windowedSlotConnectionRDD =
        slotConnectionRDD.window(Durations.seconds(batchInterval * durationThreshold));

// obtain trajectory companion
// format: <{objectId}, {slotId}>
JavaPairDStream<String, Iterable<Integer>> companionRDD =
        windowedSlotConnectionRDD
                .flatMapToPair(new CoverageDensityConnectionSubsetMapper(sizeThreshold))
                .mapToPair(new CoverageDensityConnectionMapper())
                .groupByKey()
                .filter(new TrajectoryCompanionFilter(durationThreshold));
```

```
        if(debugMode)
            companionRDD.print();
        else
            companionRDD.saveAsHadoopFiles(outputDir, "csv",
                    String.class, String.class, TextOutputFormat.class);

        ssc.start();
        ssc.awaitTermination();
    }
}
```

# Bibliography


[1] A. Aji, H. Vo, and F. Wang. Effective spatial data partitioning for scalable query processing. *CoRR*, abs/1509.00910, 2015.

[2] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: The stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 665–665, New York, NY, USA, 2003. ACM.

[3] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *The VLDB Journal*, 5(4):264–275, Dec. 1996.

[4] E. Brewer. Pushing the cap: Strategies for consistency and availability. *Computer*, 45(2):23–29, Feb. 2012.

[5] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[8] A. Eldawy, L. Alarabi, and M. F. Mokbel. Spatial partitioning techniques in spatialhadoop. *Proc. VLDB Endow.*, 8(12):1602–1605, 2015.

[9] A. Eldawy and M. F. Mokbel. SpatialHadoop: A MapReduce Framework for Spatial Data. In *31st ICDE 2015*, pages 1352–1363, 2015.

[10] M. Ester, H. peter Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press, 1996.

[11] T. Foley and J. Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of ACM SIGGRAPH/EUROGRAPHICS*, pages 15–22, 2005.

[12] J. Gudmundsson and M. van Kreveld. Computing longest duration flocks in trajectory data. In *Proceedings of GIS*, pages 35–42, 2006.

[13] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of SIGMOD*, pages 47–57, 1984.

[14] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient indexing of spatiotemporal objects. In *Proceedings of the 8th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '02, pages 251–268, London, UK, UK, 2002. Springer-Verlag.

[15] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen. Discovery of convoys in trajectory databases. *Proc. VLDB Endow.*, 1(1):1068–1080, 2008.

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

[17] M. P. Kleeman and G. B. Lamont. The multi-objective constrained assignment problem. In *Proceedings of GECCO*, pages 743–744, 2006.

[18] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in r-trees: A bottom-up approach. In *Proceedings of VLDB*, pages 608–619, 2003.

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

[20] Z. Li, B. Ding, J. Han, and R. Kays. Swarm: Mining relaxed temporal moving object clusters. *Proc. VLDB Endow.*, 3(1-2):723–734, Sept. 2010.

[21] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable distributed stream join processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 811–825, New York, NY, USA, 2015. ACM.

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

[23] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Commun. ACM*, 54(6):114–123, June 2011.

[24] J. Miller, M. Raymond, J. Archer, S. Adem, L. Hansel, S. Konda, M. Luti, Y. Zhao, A. Teredesai, and M. Ali. An extensibility approach for spatio-temporal stream processing using microsoft streaminsight. In *Proceedings of SSTD*, pages 496–501, 2011.

[25] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, pages 170–177, 2010.

[26] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *Proceedings of USENIX*, pages 275–288, 2014.

[27] H. Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, June 1984.

[28] C. T. and C. N. Mapreduce online. In *Proceedings of NSDI*, page 313?C328, 2010.

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

[30] L.-A. Tang, Y. Zheng, J. Yuan, J. Han, A. Leung, C.-C. Hung, and W.-C. Peng. On discovery of traveling companions from streaming trajectories. In *ICDE 2012*, April 2012.

[31] L.-A. Tang, Y. Zheng, J. Yuan, J. Han, A. Leung, W.-C. Peng, and T. L. Porta. A framework of traveling companion discovery on trajectory data streams. *ACM Trans. Intell. Syst. Technol.*, 5(1):3:1–3:34, Jan. 2014.

[32] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proceedings of SIGMOD*, pages 331–342, 2000.

[33] H. Wang, K. Zheng, J. Xu, B. Zheng, X. Zhou, and S. Sadiq. Sharkdb: An in-memory column-oriented trajectory storage. In *Proceedings of CIKM*, pages 1409–1418, 2014.

[34] H. Wang, K. Zheng, X. Zhou, and S. Sadiq. Sharkdb: An in-memory storage system for massive trajectory data. In *Proceedings of SIGMOD*, pages 1099–1104, 2015.

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

[36] W. Xu, N. R. Juri, A. Gupta, A. Deering, C. Bhat, J. Kuhr, and J. Archer. Supporting large scale connected vehicle data analysis using hive.

[37] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, and Y. Huang. T-drive: Driving directions based on taxi trajectories. In *ACM SIGSPATIAL GIS 2010*. Association for Computing Machinery, Inc., November 2010.

[38] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of USENIX*, pages 2–2, 2012.

[39] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[40] K. Zheng, Y. Zheng, N. J. Yuan, and S. Shang. On discovery of gathering patterns from trajectories. In *IEEE International Conference on Data Engineering (ICDE 2013)*. IEEE, April 2013.

[41] K. Zheng, Y. Zheng, N. J. Yuan, S. Shang, and X. Zhou. Online discovery of gathering patterns over trajectories. *IEEE Transaction on Knowledge Discovery and Data Engineering*, 2014.

[42] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining interesting locations and travel sequences from gps trajectories. In *Proceedings of World Wide Web*, pages 791–800, 2009.