Big Mobility Data Analytics: Algorithms and Techniques for Efficient Trajectory Clustering

Department of Informatics

School of Information and Communication Technologies

University of Piraeus

A thesis submitted in partial fulfillment of the requirements for the degree of *Doctor of Philosophy* by

Panagiotis Tampakis

Piraeus, October 2019

Big Mobility Data Analytics: Algorithms and Techniques for Efficient Trajectory Clustering

A thesis submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

in the Department of Informatics
of the School of Information and Communication Technologies
at the University of Piraeus
By

Panagiotis Tampakis

To my family and all the people that supported me during my years as a PhD candidate.

Abstract

The unprecedented rate of trajectory data generation that has been observed during the recent years, caused by the proliferation of GPS-enabled devices, poses new challenges in terms of storage, querying, analytics and knowledge extraction from mobility data.

One of these challenges is cluster analysis, which aims at identifying clusters of moving objects according to the similarity degree of their movement. Discovering clusters of moving objects is an important operation when trying to extract knowledge out of mobility data, since by doing so, the underlying hidden patterns of collective behavior can be unveiled. What is even more challenging is treating knowledge discovery techniques, such as cluster analysis, as an integral part of a real DMBS, which can turn out to be practical and useful in real-world application scenarios, where issues like the ease of use (e.g., via a simple SQL interface) are taken into consideration. Furthermore, the support of incremental and progressive cluster analysis in the context of dynamic applications is of great interest, where (i) new trajectories arrive at frequent rates, and (ii) the analysis is performed over different portions of the dataset, and this might be repeated several times per analysis task. However, performing such "expensive" operations over immense volumes of data in a centralized way is far from straightforward, which in turn calls for parallel and distributed algorithms to address the scalability requirements posed by the Big Data era. The bottleneck of performing "expensive" operations, such as cluster analysis, is the underlying join query. Joining trajectory datasets is not only the cornerstone of various trajectory cluster analysis methods, but it is also a significant operation in mobility data analytics with a wide range of applications, such as carpooling, suspicious movement discovery, etc. In this thesis, we aim to address the above challenges.

Towards this direction, we propose a novel in-DBMS *Sampling-based Sub Trajectory Clustering* algorithm, namely *S^2T-Clustering*, which is incorporated in a real MOD engine over an extensible DBMS (PostgreSQL in our prototype implementation) and turns out to solve the problem more effectively than state-of-the-art techniques. Moreover, we introduce the *temporally-constrained subtrajectory cluster analysis* problem. To address it, we propose *ReTraTree*, an indexing scheme which organizes trajectories by using an effective spatio-temporal partitioning technique. Partitions in *ReTraTree* correspond to groupings of subtrajectories, which are incrementally maintained and represented via a hierarchical organization of a small (thus, light-weight in-memory) set of 'representative' subtrajectories. Given this, the problem in hand can be efficiently solved as a query operator on *ReTraTree*, coined *QuT-Clustering*. Our approach further contributes to the mobility data management and mining domain for the additional reason that it has been designed and implemented in a MOD engine. Such functionality enables the application users to perform progressive cluster analysis via simple SQL in a real extensible DBMS. Furthermore, we propose an efficient in-DBMS architecture for progressive time-aware subtrajectory cluster analysis, by utilizing the aforementioned in-DBMS solutions along with a Visual Analytics (VA) tool to facilitate real world analysis.

Towards addressing the challenges posed by the Big Data era, we introduce the *Distributed Subtrajectory Join* query, an important operation in the spatiotemporal data management domain, where very large datasets of moving object trajectories are processed for analytic purposes. To address this problem in a scalable manner, we follow the MapReduce programming model. Finally, we address the problem of *Distributed Subtrajectory Clustering* by building upon the *Distributed Subtrajectory Join* query, in order to tackle the problem in an efficient manner. We propose two alternative trajectory segmentation algorithms and a distributed clustering algorithm where the clusters are identified in a parallel manner.

Περίληψη

Ο πρωτοφανής ρυθμός παραγωγής δεδομένων τροχιάς που παρατηρείται τα τελευταία χρόνια και προκλήθηκε από τον πολλαπλασιασμό των συσκευών με δυνατότητα GPS, δημιουργεί νέες προκλήσεις όσον αφορά την αποθήκευση, την αναζήτηση, την ανάλυση και την εξαγωγή γνώσης από δεδομένα κίνησης.

Μια από αυτές τις προκλήσεις είναι η ανάλυση συστάδων, η οποία στοχεύει στον εντοπισμό συστάδων κινούμενων αντικειμένων σύμφωνα με τον βαθμό ομοιότητας της κίνησης τους. Η ανακάλυψη συστάδων κινούμενων αντικειμένων είναι μια σημαντική λειτουργία κατά την προσπάθεια εξαγωγής γνώσης από δεδομένα κίνησης, διότι με τον τρόπο αυτό μπορούν να αποκαλυφθούν τα υποκείμενα κρυμμένα πρότυπα συλλογικής συμπεριφοράς. Αυτό που είναι ακόμη πιο δύσκολο είναι η αντιμετώπιση των τεχνικών ανακάλυψης γνώσης, όπως η ανάλυση συστάδων, ως ενα αναπόσπαστο κομμάτι ενός πραγματικού DMBS, το οποίο μπορεί να αποδειχθεί πρακτικό και χρήσιμο σε σενάρια εφαρμογών πραγματικού κόσμου, όπου λαμβάνονται υπόψη θέματα όπως η ευκολία χρήσης (π.χ. μέσω μιας απλής διεπαφής SQL). Επιπλέον, η υποστήριξη της σταδιακής και προοδευτικής ανάλυσης συστάδων στο πλαίσιο των δυναμικών εφαρμογών παρουσιάζει μεγάλο ενδιαφέρον, όπου (i)οι νέες τροχιές φθάνουν με συχνό ρυθμό και (ii)η ανάλυση πραγματοποιείται σε διαφορετικά τμήματα του συνόλου δεδομένων και αυτό μπορεί να επαναληφθεί πολλές φορές ανά εργασία ανάλυσης. Ωστόσο, η εκτέλεση τέτοιων 'δαπανηρών' λειτουργιών σε τεράστιους όγκους δεδομένων με κεντρικοποιημένο τρόπο δεν είναι καθόλου εύκολη, πράγμα που με τη σειρά του απαιτεί παράλληλους και κατανεμημένους αλγόριθμους για την αντιμετώπιση των απαιτήσεων που θέτει η εποχή των Μεγάλων Δεδομένων. Το σημείο συμφόρησης της εκτέλεσης τέτοιων 'δαπανηρών' λειτουργιών, όπως η ανάλυση συστάδων, είναι το υποκείμενο ερώτημα 'σύνδεσης'. Η 'σύνδεση' συνόλων δεδομένων τροχιάς δεν αποτελεί μόνο τον ακρογωνιαίο λίθο των διαφόρων μεθόδων ανάλυσης συστάδων τροχιών, αλλά είναι επίσης μια σημαντική λειτουργία σε αναλύσεις δεδομένων κίνησης με ένα ευρύ φάσμα εφαρμογών, όπως ο συνεπιβατισμός, η ανακάλυψη ύποπτων κινήσεων κλπ. Σε αυτή τη διατριβή, στοχεύουμε να αντιμετωπίσουμε τις παραπάνω προκλήσεις.

Προς αυτή την κατεύθυνση, προτείνουμε έναν νέο in-DBMS αλγόριθμο συσταδοποίησης υποτροχιών βασιζόμενο στη δειγματολειψία, ο οποίος ονομάζεται S^{2}T-Clusteringκαι είναι ενσωματωμένος σε ένα πραγματικό MOD μέσω ενός επεκτάσιμου DBMS (της PostgreSQL στην πρωτότυπη υλοποίησή μας), που επιλύει το πρόβλημα πιο αποτελεσματικά από την τελευταία λέξη της τεχνολογίας. Επιπλέον, εισάγουμε το πρόβλημα της *χρονικά περιορισμένη ανάλυση συστάδων υποτροχιών*. Προκειμένου να το αντιμετωπίσουμε, προτείνουμε το *ReTraTree*, ένα σχήμα ευρετηρίου που οργανώνει τροχιές χρησιμοποιώντας μια αποτελεσματική τεχνική χωροχρονικής διαμέρισης. Οι διαμερίσεις στο *ReTraTree*, αντιστοιχούν σε ομάδες υποτροχιών, οι οποίες διατηρούνται σταδιακά και αντιπροσωπεύονται μέσω μιας ιεραρχικής οργάνωσης μίας μικρής (κατά συνέπεια 'ελαφριάς' όσον αφορά τη μνήμη) ομάδας «αντιπροσωπευτικών» υποτροχιών. Δεδομένου αυτού, το υπό μελέτη πρόβλημα μπορεί να λυθεί αποτελεσματικά ως ένα ερώτημα στο *ReTraTree*, το οποίο το ονομάζουμε *QuT-Clustering*. Η προσέγγισή μας συμβάλλει περαιτέρω στον τομέα διαχείρισης και εξόρυξης γνώσης δεδομένων κίνησης για τον πρόσθετο λόγο ότι έχει σχεδιαστεί και εφαρμοστεί σε ένα MOD. Αυτή η λειτουργικότητα επιτρέπει στους χρήστες της εφαρμογής να εκτελούν προοδευτική ανάλυση συστάδων μέσω απλής SQLσε πραγματικά επεκτάσιμα DBMS. Επιπλέον, προτείνουμε μια αποτελεσματική αρχιτεκτονική in-DBMS για την προοδευτική ανάλυση συστάδων υποτροχιών, χρησιμοποιώντας τις προαναφερθείσες in-DBMS λύσεις, σε συνδυασμό με ενα εργαλείο Οπτικής Ανάλυσης (VA) για τη διευκόλυνση της ανάλυσης στον πραγματικό κόσμο.

Προς αντιμετώπιση των προκλήσεων που θέτει η εποχή των Μεγάλων Δεδομένων, παρουσιάζουμε το ερώτημα της *Κατανεμημένης Σύνδεσης Υποτροχιών*, μια σημαντική λειτουργία στον τομέα των χωροχρονικών δεδομένων, όπου πολύ μεγάλα σύνολα δεδομένων τροχιών κινούμενων αντικειμένων επεξεργάζονται για αναλυτικούς σκοπούς. Για να αντιμετωπίσουμε αυτό το πρόβλημα με αποτελεσματικό τρόπο, χρησιμοποιήσαμε το μοντέλο προγραμματισμού MapReduce. Τέλος, αντιμετωπίζουμε το πρόβλημα της *Κατανεμημένης Συσταδοποίησης Υποτροχιών* με βάση το ερώτημα της *Κατανεμημένης Σύνδεσης Υποτροχιών*, προκειμένου να αντιμετωπίσουμε αποτελεσματικά το πρόβλημα. Στη συνέχεια, προτείναμε δύο εναλλακτικούς αλγορίθμους τμηματοποίησης τροχιάς και έναν κατανεμημένο αλγόριθμο ομαδοποίησης, όπου οι συστάδες ταυτοποιούνται με έναν παράλληλο τρόπο.

Contents

List of Figures

List of Tables

Acronyms

AIS Automatic Identification System (radio navigation)

API Application Program Interface

DBMS DataBase Management System

DTW Dynamic Time Warping

GiST Generalized Search Tree

GPS Global Positioning System

GPU Graphics Processing Unit

GUI Graphical User Interface

HDFS Hadoop Distributed File System

KDD Knowledge Discovery in Data

LBS Location-Based Services

LCSS Longest Common SubSequence

MBB Minimum Bounding Box

MOD Moving Objects Database

MR MapReduce

ORDBMS Object-Relational DataBase Management System

RDBMS Relational DataBase Management System

SQL Structured Query Language

VA Visual Analytics

Setting the Scene Part I

1 Introduction

During the recent years, the proliferation of GPS enabled devices has led to the production of enormous amounts of mobility data. This “explosion” of mobility data generation has posed new challenges in the data management community, in terms of storage, querying, analytics and knowledge extraction out of such data. In this chapter, we introduce the problem under consideration, provide the motivation, and present some interesting application scenarios. Furthermore, we present the challenges faced and the contributions of this thesis. Finally, we provide the thesis organization.

1.1 Motivation

During the past two decades, the field of Moving Object Databases (MODs) has emerged for the efficient management of such data, by exploiting existing extensible DBMSs [19, 66]. By doing so, we tackle the problem of storage, querying and indexing. However, knowledge discovery techniques, such as cluster analysis, are not treated as an integral part of MODs. What actually happens is that the data are exported from the MOD to an ad-hoc implementation and subsequently the results are re-imported to the MOD. Treating knowledge discovery techniques, such as cluster analysis, as an integral part can be practical and useful in real-world application scenarios, where where issues like the ease of use use (e.g., via a simple SQL interface) are taken into consideration. Therefore, we argue that this is an important step towards bridging the gap between MOD management and mobility data mining, as state-of-art approaches [52, 99, 35] could make use of the efficiency and the advantage of our proposal to execute in-DBMS clustering via simple SQL.

Trajectory clustering is an important operation of knowledge discovery from mobility data. The research so far has focused mainly in methods that aim to identify specific collective behavior patterns among moving objects, such as [48, 45, 44, 60, 51, 50, 92, 107, 28]. However, this kind of approaches operate at specific predefined temporal "snapshots" of the dataset, thus ignoring the route of each moving object between these sampled points. Another line of research, tries to identify patterns that are valid for the entire lifespan of the moving objects [56, 67, 21, 79]. However, discovering clusters of complete trajectories can overlook significant patterns that might exist only for some portions of their lifespan. Subtrajectory clustering is a typical cluster analysis problem, where the goal is to segment trajectories and discover clusters of subtrajectories. Finding a solution to the above described subtrajectory clustering problem is challenging; what is even more challenging, is how one can support incremental and progressive cluster analysis in the context of dynamic applications, where (i) new trajectories arrive at frequent rates, and (ii) the analysis is performed over different portions of the dataset, and this might be repeated several times per analysis task.

However, performing advanced knowledge discovery operations, such as subtrajectory clustering (e.g., [70, 49, 3]), over immense volumes of data in a centralized way is far from straightforward. This calls for parallel and distributed algorithms that address the scalability requirements. The bottleneck of these approaches is that their computation raises efficiency issues due to the fact that all of them are actually based on a spatiotemporal similarity join query. Repeated multidimensional range queries, which are found in the core of a large class of knowledge discovery algorithms can be transformed equivalently in a similarity self-join. Joining trajectory datasets is not only the cornerstone of various methods that aim to identify different kinds of mobility patterns (group behavior, etc.), but is also a significant operation in mobility data analytics with a wide range of applications, such as, carpooling, suspicious movement discovery etc. It is obvious that the need for overcoming this bottleneck becomes more imperative in the era of Big Data, which calls for parallel and distributed solutions that scale beyond the limitations of a single machine. In this context, one challenge is how to partition the data in such a way so that each node can perform its computation independently, thus minimizing the communication cost between nodes, which is a cost that can turn out to be a serious bottleneck. Another challenge, related to partitioning, is how to achieve load balancing, in order to balance the load fairly between the different nodes. Yet another challenge is to minimize the iterations of data processing, which are typically required in clustering algorithms.

1.2 Application Scenarios

To facilitate the discussion about the importance of this thesis and its impact in our society, let us cite some interesting application scenarios.

To begin with, let us consider the trajectory join operation. For instance, in the urban traffic domain, carpooling is becoming increasingly popular. More concretely, consider a mobile application which tries to match users that can share a ride based on their past movements. Here, given a set of trajectories we want to find all the pairs of users that can share a ride for a portion of their everyday routes without significantly deviating (spatially and temporally) from their daily routine (i.e. retrieve all pairs of maximal subtrajectories that move close in space and time).

Another interesting scenario concerns the identification of suspicious movement by a governmental security agency. For instance, given a set of trajectories that depict the movement of suspicious individuals, we would like to retrieve all the pairs of moving objects that move "close" to each other for a duration that exceeds a threshold (moving together for small periods of time could be considered as coincidental) as candidates for illegal activity. In the maritime domain, suspicious group movements of vessels is also of great interest, since they might indicate e.g., illegal transshipment activity.

Trajectory segmentation techniques [62, 70], can directly benefit from the subtrajectory join query since their input, for each trajectory, is the spatiotemporal neighbourhood of each object. Moreover, such a query is in fact the building block for a number of operations than aim to identify mobility patterns, such as co-movement patterns (e.g., flocks [37], convoys [44], swarms [51]).

An even more challenging problem is that of subtrajectory clustering [70, 3]. An interesting application scenario of subtrajectory clustering is network discovery, where given a set of trajectories (e.g from the maritime or the aviation domain) we want to identify the underlying network of movement by grouping subtrajectories that move "close" to each other and use cluster representatives/medoids as network edges.

An additional valuable application scenario of subtrajectory clustering is predictive analytics over mobility data, where the goal is the extraction of valuable knowledge from data and its utilization in order predict future behavioural patterns (i.e. movement) [74, 73]. The general idea is first to identify popular mobility patterns, either global (for the whole dataset) or local (for each moving object separately), by employing some subtrajectory clustering technique that also provides the cluster representatives. Then, when some new position of a moving object is reported, the goal is to try to "match" the new portion of movement with the most similar historical patterns and employ this pattern in order to predict its future location.

Another application of great interest, is that of interactive mobility data exploration and analysis, which can aid/facilitate mobility analysts, in e.g. urban planning and traffic analysis applications. To achieve this, we demonstrate how a MOD engine, built on top of an extensible DBMS, can efficiently incorporate advanced mobility data analytics methods. In more detail, we demonstrate the feasibility of progressive time-aware analytics, in terms of allowing a data analyst to select different time periods to perform his/her analysis, without being obliged to apply from scratch costly preprocessing or iterative clustering procedures.

1.3 Challenges

Concerning the challenges that arise when dealing with the aforementioned problems, the problem of subtrajectory clustering is shown to be NP-Hard (cf. [3]). In addition, the objects to be clustered are not known beforehand, but have to be identified through a trajectory segmentation procedure. Efforts that try to deal with this problem in a centralized way do exist [49, 70, 3], however, applying centralized algorithms for subtrajectory clustering over massive data in a scalable way is far from straightforward. This calls for parallel and distributed algorithms that address the scalability requirements. In this context, one challenge is how to partition the data in such a way so that each node can perform its computation independently, thus minimizing the communication cost between nodes, which is a cost that can turn out to be a serious bottleneck. Another challenge, related to partitioning, is how to achieve load balancing, in order to balance the load fairly between the different nodes. Yet another challenge is to minimize the iterations of data processing, which are typically required in clustering algorithms.

Concerning the trajectory join problem, there have been some efforts to tackle variations of this problem in a centralized way [9, 11, 17]. However, the problem definitions of these approaches differ from one another and are not general enough in order to capture different kinds of mobility patterns. Moreover, in [10, 9] they make the assumption that all the trajectories have the same number of points and that these points are synchronized between any two given trajectories, which is not realistic in real life applications and requires a preprocessing step that can be prohibitive when dealing with Big Data. In addition, in [17] the definition of the trajectory join is not symmetric. However, these solutions discover pairs of entire trajectories and cannot identify matching subtrajectories. In [10], all pairs of "matching" (with respect to a spatial threshold) subtrajectories of exactly δt duration are retrieved, where the goal is to identify maximally "matching" subtrajectories, which is vital for exploiting the output in subsequent steps, e.g., the mining operations mentioned above. Moreover, applying these centralized solutions to a parallel and distributed environment is not straightforward and is often impossible if radical changes to the methods/algorithms do not take place, since there are several non-trivial issues that arise. For instance, how to partition the data in such a way so that each partition can be processed independently and be of even size. As another example, assuming that all the nodes in the cluster have similar processing power, in order to achieve the fastest execution of an algorithm the load must be balanced among the nodes. Furthermore, due to the read/write cost from/to the disk, the number of reads and writes should be minimized and ideally the whole procedure should be performed in "one pass".

1.4 Contributions

The contributions of this thesis are summarized below:

- We propose *S^2T-Clustering*, an efficient in-DBMS sampling-based subtrajectory clustering algorithm. We implement *S^2T-Clustering* as a query operator in an extensible DBMS, namely PostgreSQL, based on access methods that exploit on the GiST indexing extensibility interface.

- We introduce the temporal-constrained subtrajectory cluster analysis problem, we design *ReTraTree*, an efficient indexing scheme for large dynamic MODs, which is based on representative trajectories found in the dataset. As a solution to the problem under study, we devise *QuT-Clustering*, a subtrajectory clustering algorithm running as simply as a query operator upon *ReTraTree*.

- We propose an architecture that builds upon the aforementioned subtrajectory clustering approaches in order to enable interactive mobility data exploration and analysis by utilizing a MOD engine built on top of extensible DBMS.
- We formally define the problem of *Distributed Subtrajectory Join* processing, investigate its main properties, and present a well-designed algorithm, called *DTJb*, and two improvements, namely *DTJr* and *DTJi*, which extends *DTJr* by exploiting an indexing scheme that speeds up the computation of the join.
- We formally define the problem of *Distributed Subtrajectory Clustering*, propose two neighborhood-aware trajectory segmentation algorithms and design an efficient and scalable solution for the specific problem.
- We perform an extensive experimental study, with both synthetic and real datasets from different mobility domains (urban, aviation and maritime) to evaluate the effectiveness and efficiency of the proposed algorithms.

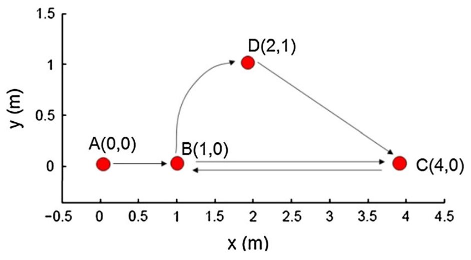
1.5 Datasets

The synthetic and real datasets that were used throughout this thesis are the following.

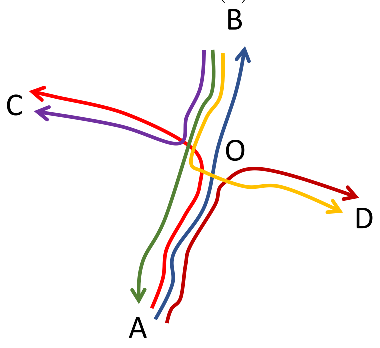
1.5.1 Synthetic Datasets

SMOD - Synthetic MOD (SMOD) consists of 400 trajectories and is used for the ground truth verification in Chapters 3 and 4. The scenario of the synthetic dataset is the following: the objects move upon a simple graph that consists of the following destination nodes (points) with coordinates: A(0,0), B(1,0), C(4,0) and D(2,1). Half of the objects move with normal speed (2 units per second) and another half move with high speed (5 units per second). Figure 1.1 illustrates the 2D map of the SMOD consisting of three one-directional ($A \rightarrow B$, $B \rightarrow D$, $D \rightarrow C$) and one bi-directional road ($B \leftrightarrow C$). All objects move under the following scenario, for a lifetime of 100 seconds:

- (normal movement – 99% of the trajectories) All objects start from point A towards point B; the high-speed objects start at t = 0 sec and the normal-speed objects start at t = 20 sec. When an object arrives at B, it ends its trajectory with a probability of 15%; otherwise, it continues with the same speed to the next point. If there exist more than one option for the next point, it decides randomly about the next destination.
- (abnormal movement – 1% of the trajectories) A few outlier objects follow a random movement in space (other than these roads) with a speed that is updated randomly.



(a)



(b)

Figure 1.1: The 2-D map of (a) SMOD and (b) Intersection

Intersection consist of 409 trajectories and is used for the ground truth verification in Chapter 7. The scenario of the synthetic dataset is the following: the objects move in the xy-plane in six predefined origin-destination pairs. More specifically, as illustrated in Figure 1.1, these pairs are $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow D$, $B \rightarrow A$, $B \rightarrow C$ and $B \rightarrow D$. The trajectories have the same starting time and similar speed.

Table 1.1 summarizes the basic statistics of the synthetic datasets that were employed in this thesis.

Table 1.1: Synthetic Datasets Summary

Statistic	**SMOD**	**Intersection**
# Trajectories	400	409
# Points	35273	2573
Dataset Duration	120 seconds	23 seconds

1.5.2 Real Datasets

IMIS[1] is a real dataset which consists of 699,031 trajectories of ships moving in the Eastern Mediterranean for a period of 3 years. This dataset contains approximately 1.5 billion records, 56GB in total size. This dataset was collected through the Automatic Identification System (AIS) through which ships are obliged to broadcast their position for maritime regulatory purposes.

IMIS$_1$ is a subset of IMIS consisting of the trajectories of 637 ships moving in the Greek seas for one week.

IMIS$_2$ is also a subset of IMIS consisting of the trajectories of 2,181 ships sailing in the Eastern Mediterranean for one week.

Brest[2][76] is a 650MB publicly available AIS dataset of vessels moving in the wider Brest area, consisting approximately of 3.65×10^5 trajectories that correspond 17×10^6 points.

GeoLife [110] consists of the trajectories of 178 users in a period of more than four years; this dataset represents a wide range of movements, including not only urban transportation (e.g., from home to work and back) but also different kinds of activities, such as sports activities, shopping, etc.

SIS[3] is a 27GB proprietary insurance dataset of moving objects around Rome and Tuscany area, that contains approximately 2.2×10^7 trajectories that correspond to 7.2×10^8 points.

LondonLanding is a dataset from the aviation domain that consists of 1118 flights approaching airports of the London metropolitan area.

Table 1.2 summarizes the basic statistics of the real datasets that were employed in this thesis.

[1]IMIS dataset has been kindly provided by IMIS Hellas for research and educational purposes. It is available for downloading at http://chorochronos.datastories.org

[2]https://zenodo.org/record/1167595#.XKHTyaRRVPa

[3]This private dataset was kindly provided by Gruppo Sistematica SpA

Table 1.2: Real Datasets Summary

Statistic	# Trajectories	# Points	Area	Dataset Duration
IMIS	699031	1.5×10^9	Eastern Mediterranean	3 years
IMIS$_1$	5110	443657	Greece	1 week
IMIS$_2$	5110	449680	Eastern Mediterranean	1 week
Brest	365000	17×10^6	Brest	6 months
GeoLife	18668	24×10^6	China and USA	4 years
SIS	2.2×10^7	7.2×10^8	Rome and Tuscany	2.5 years
London Landing	1118	95396	London	1 day

1.6 Thesis Organization

The rest of this thesis is organized as follows:

Chapter 2 presents some background knowledge and a literature review on the topics of this thesis.

Part II deals with in-DBMS solutions to problems related with the management and mining of mobility data. More specifically, Chapter 3 introduces *S^2T-Clustering* [70], an efficient in-DBMS sampling-based subtrajectory clustering algorithm. Subsequently, Chapter 4 presents an efficient indexing scheme for large dynamic MODs, called *ReTraTree*, along with a query operator upon *ReTraTree*, which tackles the problem of *temporal-constrained subtrajectory cluster analysis* [69]. Chapter 5 introduces an architecture that builds upon the solutions presented in Chapters 3 and 4 in order to enable interactive mobility data exploration and analysis.

Inspired by the limitations of the approaches that were presented in Part II and under the light of the Big Data era and the immense volumes of generated data, Part III deals with the management and mining of mobility data in a distributed way. In more detail, Chapter 6 presents an efficient load balanced, index-based solution to the problem of *Distributed Subtrajectory Join* processing, which is the cornerstone of various methods that aim to identify different kinds of mobility patterns. Chapter 7, presents an efficient and scalable solution for the problem of *Distributed Subtrajectory Clustering*, which builds upon the findings of Chapter 6.

Finally in Part IV we conclude the thesis. In particular, in Chapter 8 we summarize the thesis and in Chapter 9 we discuss future work directions.

2 Background

In this chapter, we provide an overview of the topics related to this thesis and a survey of the related work. Initially we discuss the key concepts in mobility data management, such as modelling, storing, querying and indexing mobility data. Subsequently, we provide an extensive literature review on joining trajectory datasets, which is a significant operation in mobility data analytics and the cornerstone of various methods that aim to identify different kinds of mobility patterns. We then discuss about mobility data mining methods, such as co-movement and sequential patterns, trajectory clustering and data-driven predictive analytics.

2.1 Fundamentals

During the recent years, the proliferation of GPS enabled devices has led to the production of enormous amounts of mobility data. This "explosion" of data generation has posed new challenges in the data management community. In this chapter we present the fundamentals about mobility data.

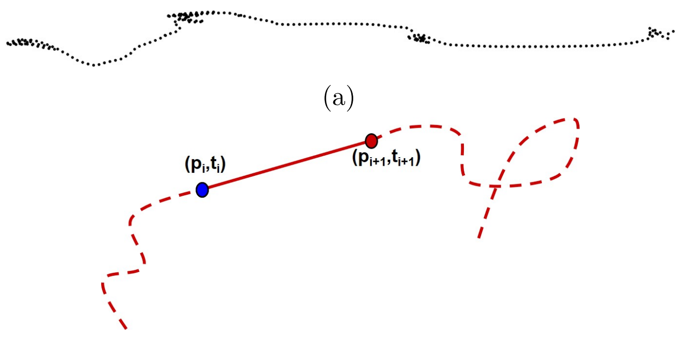
2.1.1 About Mobility Data

Mobility data refers to data, representing the movement of objects, like people, animals, cars, vessels, aircrafts, hurricanes etc. It is important to clarify from the very beginning that 'location' mentioned in the previous examples varies in size and resolution; it could vary from points (for instance, locations recorded by GPS devices are points with a tolerance of a few meters) to a very broad area (for instance, locations recorded by mobile phone providers could be regions of several sq.km. area). This inaccuracy in positioning is not necessarily a problem. It depends on the application whether an area of order of sq.km. is adequate positioning or the object should be identified within no more than a few meters. For example, resolution of a few meters is mandatory for effective navigation (where the road signs and turns are in front of us); on the other hand, answers to information services, such as localized weather report may be valid for a resolution of several km. Due to its popularity and its high accuracy with respect to 'location' recorded, hereafter we focus on GPS data.

Concerning the analysis of mobility data, mobility data analytics aim to describe the mobility of objects, to extract valuable knowledge by revealing motion behaviors or patterns, to predict future mobility behaviors or trends and in general, to generate various perspectives out of data, useful for many other scientific fields. To serve its purpose, mobility data analytics follows a series of steps. Having assured the collection and efficient storage of mobility data, the next step for an analyst is to familiarize with the mobility data by employing a number of techniques (e.g., statistics, data visualization and visual analytics) to form a compact and complete picture of the available mobility data. Afterwards, the analyst, depending on the application requirements, proceeds to the appropriate preprocessing steps. The goal is to bring mobility data in a form that serves its later usage by various processes and algorithms that respond to the given questions. Data preparation is essential for successful mobility data analytics, since low-quality data typically result in incorrect and unreliable conclusions. Finally, mobility data are ready for the application of knowledge extraction methods that will satisfy the given application requirements. There are already several analytical methods and algorithms available from the scientific community and an analyst has the capability either to employ some of the existing techniques or implement some ad-hoc solutions that better serve the problems needs.

2.1.2 Modeling Mobility Data

Due to discretization, a moving object is represented by a trajectory, which is a sequence of sampled time-stamped locations $(p_i,\ t_i)$ where p_i is a 2-dimensional point–pair $(x_i,\ y_i)$ – and t_i is the recording timestamp of p_i, as illustrated in Figure 2.1(a). In order to simulate the continuous movement of objects, a common representation of a trajectory is a 3-dimensional polyline where vertices correspond to time-stamped locations $(p_i,\ t_i)$ and linear interpolation is assumed between $(p_i,\ t_i)$ and $(p_{i+1},\ t_{i+1})$, as depicted in Figure 2.1(b).



(b)

Figure 2.1: (a) An example of a raw trajectory and (b) an example of linear interpolation.

The basic assumption followed by this formula is that the velocity vector (i.e., speed and direction) remains constant during time interval $[t_i, t_{i+1})$, which (unfortunately) results in discontinuous evolution of those movement parameters, exactly at the recorded timestamps t_i. The result of the above discussion is that the "trajectory" of a moving object can be modeled according to the concept of sliced representation: this model decomposes the temporal development of a value into fragments called 'slices', such that within a slice this development can be described by some kind of "simple" function.

The above discussion assumes movement in the (unconstrained) Euclidean space. However, in several applications, moving objects are assumed to move along transportation networks. Formally, a transportation network is modeled as $G = (V, E)$, where V is a set of m vertices $\{v_1, v_2, \ldots, v_m\}$, representing e.g., road junctions, and $E \subset V \times V$ is a set of n edges $\{e_1, e_2, \ldots, e_n\}$, with each edge connecting two vertices, v_{from} and v_{to}, and representing e.g., routes between road junctions. Following this, a trajectory can be modeled as a sequence of edges and/or vertices along with some offsets and the corresponding time.

2.2 Mobility Data Management

An integral part of any mobility data analytics effort is to assure the collection and efficient storage of mobility data. Towards this direction there have been proposed several mobility-aware queries and the corresponding indexing structures to ensure their efficient execution. Moreover, there have been some efforts to integrate such functionality with real world RDBMSs.

2.2.1 Querying Mobility Data

According to [71], queries over mobility data can be classified queries as *location-oriented* versus *trajectory-oriented* queries with the distinction emphasizing on the assumed model of trajectory objects: the former considers movement as a sequence of sampled points whereas the latter considers movement as a continuous evolution (trajectory).

Location-oriented queries are essentially spatial, though taking into consideration that the spatial objects are changing their locations with time. Typical examples include:

- *Continuous Nearest Neighbor* (CNN) query: CNN query retrieves the nearest (among a set of candidate points) of every location on a polyline (actually, the trajectory of a moving object from a starting to an ending point).
- *Sequenced Route* (SR) query: SR query finds the shortest path from a starting s to an ending point e, visiting a sequence of facilities from a set of facility classes.

Trajectory-oriented queries are the most popular in MOD literature and can be broken down to *coordinate-based* and *trajectory-based* queries.

Coordinate-based queries, are focused in filtering the trajectory database setting conditions on the (space- and/or time-) coordinates of the segments that compose the trajectories. Figure 2.2 illustrates these types of queries. Such queries are:

- *timeslice queries*,
- *spatiotemporal range queries* and
- *nearest-neighbor queries* in three versions, *point NN*, *trajectory NN*, and *historical continuous point NN*.

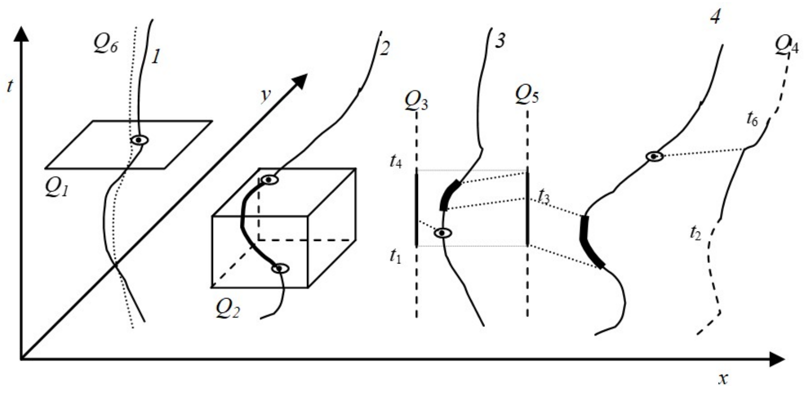


Figure 2.2: (Examples of timeslice, spatiotemporal range and nearest-neighbor queries [71].

On the other hand, in *trajectory-based* queries, it is essential to have knowledge of the entire trajectory (or, at least, a subtrajectory of it) in order to be able to provide the answer, e.g., “find the trajectories that are the most similar to a given trajectory”, as well as topological and navigational queries with respect to a stationary object, e.g., “find the trajectories that entered (crossed, left, bypassed, etc.) a given region during a given time interval” and “. . . were located west of (south of, etc.) a given region . . . ”, respectively) as well as their counterparts when the reference object is another trajectory (“find the trajectories that met (followed, were in front of, etc.) a given trajectory”).

Moreover, combined coordinate- and trajectory-based queries are expected to be of great interest in MOD. Consider the following example: “find trajectories that entered a given region during a given time interval, stayed inside the region for a given time period, and then left the region at a speed higher than a given speed threshold”. For sure, interesting query processing issues arise here. Last but not least, an emerging family of trajectory-oriented queries is motivated by traffic analysis; examples include path queries (“find the trajectories that have moved along an entire path”) and traffic queries (“find places on the road network where traffic jams appear”), etc.

2.2.2 Indexing Mobility Data

The ubiquity of R-trees in spatial databases has been expanded also in the domain of mobility data. To name but a few representative approaches, the 3D-Rtree for the purposes of spatiotemporal indexing was proposed in [94], while it was adapted to organize trajectories of moving objects in [75], where the TB-tree and STR-tree were introduced. The overhead introduced by representing trajectory segments as MBBs in a R-tree like structure was studied in [39]. MV3R-tree [93] is another efficient proposal for indexing the past movement of mobility data, consisting of a multi-version R-tree (extending the idea of multi-version B-tree) and a small auxiliary 3D-Rtree pointing to the leaf nodes of the former. A recent approach in trajectory indexing includes TrajStore [18], which is actually a storage scheme consisting of distinct spatial and temporal indexes. PA-tree [58] is a parametric index that organizes the coefficients of continuous polynomials approximating movement functions. All the above state-of-the-art indexing techniques make use of clustering methods so as to take advantage of their properties in the organization of the data (e.g., improve the compactness of the MBBs in R-tree-like structures).

2.2.3 In-DBMS Mobility Data Management

During the past decade, the field of MOD has emerged as a strong candidate for the efficient management of trajectory data exploiting on the robust architecture of extensible DBMS; Secondo [19], Hermes [66] and MobilityDB [111] are typical examples of this paradigm. Nevertheless, extending a DBMS does not reduce the complexity of understanding their concurrency and recovery protocols, and as such, does not reduce the implementation effort of an external access method when compared to a built-in one, assuming that identical levels of concurrency, robustness and integration are desired [46]. Actually, complexity is the main reason that almost none of the numerous access methods for mobility data that have been proposed in the literature, [39, 75, 92] to name but a few representatives, have been integrated in a real Object-Relational DBMS. Even GiST [40], which has been proposed to provide access method extensibility has only recently started to be used in the context of mobility data by Hermes@PostgreSQL [98] and MobilityDB [111]. Mainly due to the above reasons, although a lot of research has been carried out in the field of MOD regarding efficient indexing and query processing, almost no related work exists in the field of mobility data mining in-DBMS [72].

2.3 Joining Trajectories

Joining trajectory datasets is a significant operation in mobility data analytics and the cornerstone of various methods that aim to identify different kinds of mobility patterns (group behavior, etc.). Trajectory joins are closely related to three topics in the spatial and spatiotemporal database management literature. These are, (a) centralized trajectory joins, (b) distributed spatial and multidimensional joins, and (c) distributed trajectory joins. Moreover trajectory joins can be categorized in *distance joins*, *k-nn joins* and *similarity joins*.

2.3.1 Distance Join

In [9] the effort focuses in identifying pairs of trajectories that move close enough, with respect to a spatial threshold, during a user specified temporal window, which is kind of limiting, since it might be of interest to identify "matches" of different duration (at least δt) during the whole lifespan of the datasets. Furthermore, in [9], no temporal tolerance is considered which can lead in missing pairs of trajectories that move with some temporal displacement. In [11], the authors try to solve the same problem in a streaming environment. In [10] the authors extend their work by not binding to the temporal dimension the interval in which two trajectories should move "together". Hence, all pairs of "matching" (with respect to a spatial threshold) subtrajectories of exactly δt duration will be returned. This definition, although more general, it still suffers from the rest of the aforementioned problems. Moreover, in these approaches there is an assumption made, that all trajectories have the same number of points which are synchronized. However, an assumption like that is not realistic in real life scenarios and supposes a preprocessing step that can be prohibitive when dealing with Big Data. A slightly different definition is provided in [17] where the goal is to identify all pairs of moving objects that, for some time intervals, move closer than a given spatial threshold. Here, the duration of the "matches" is not fixed. However, the trajectory join definition, here, is asymmetric and time relaxation is not considered, so the distance between two objects refers to their distance at the same time point t. Furthermore, the minimum duration of the "matches" cannot be limited which can lead to pairs with very small duration that might not be useful for some applications. Finally, the solution provided is focused to an instantiation of the problem, called Window Trajectory Distance Join, which limits the problem to a user-specified time window.

2.3.2 k-nn Join

More specifically, [104] and [100] address the problem where given a reference trajectory and an integer k they want to discover the k most similar trajectories to the reference trajectory. This is achieved by broadcasting the input trajectory to all the nodes and calculating the LCSS distance. The problem of k-nn join by using the MapReduce framework is tackled in [29]. More specifically, given two sets of trajectories R and M, an integer k and a time interval $[t_s, t_e]$, the algorithms proposed there return the k nearest neighbors from R for each object in M during this interval. In more detail, a five step procedure (five MR jobs) is adopted where the data are reprocessed, subtrajectories are extracted, the time dependent upper bound is computed, candidates are found and the trajectories are joined. Similarly, in [100] given a query trajectory they try to find the k most similar ones.

2.3.3 Similarity Join

A similar but different problem is the one of trajectory similarity join, where the goal is to retrieve all pairs of trajectories that exceed a given similarity threshold as in [88] and [22]. However, both of them return as a result pairs of trajectories and not subtrajectories. An approach very similar to ours is presented in [13], where, given a pair of trajectories they try to perform partial matching, finding the most similar subtrajectories between these two trajectories. Different variations of the problem are presented, where the duration of the “match” is specified beforehand or not. Nevertheless, the problem in [13] is not a join operation and temporal tolerance is not considered. To sum up, all of the above approaches are centralized and applying them to a parallel and distributed environment is non-trivial.

Recently, the algorithms proposed in [80, 81] find all pairs of network-constrained trajectories that exceed a similarity threshold in a parallel manner. However, the parallelization proposed there handles each trajectory separately by assuming that all data need to be replicated for each trajectory, which makes such a solution inapplicable to the Big Data setting. Finally, these approaches (a) assume that the underlying network is known, which is not something trivial in some domains (e.g., maritime or aviation) and (b) work at the entire trajectories and cannot identify matching subtrajectories.

2.3.4 Spatial & Multidimensional Joins

A special class of joins which is very relative to our problem is that of spatial join. There have been several efforts to tackle this issue using the MapReduce framework. In particular, [105], which is based on the traditional PPBSM algorithm [64], partitions the input data into small evenly disjoint tiles at *Map* stage and joins them at *Reduce* stage by further partitioning the data into strips and performing a plane sweeping algorithm along with a duplication avoidance technique. [105] uses no indexes and is a non-invasive approach to the underlying system. Instead, it performs an in-memory Reduce-side Join. As far as it concerns objects that intersect with more than one partitions, these are replicated to each of the partitions that overlap and proper care is taken in order to avoid duplicate results. Other works that try to deal with the problem of spatial join by using the MapReduce framework are presented in [84], [5], [26] and [77]. In more detail, [5] first partitions the data by focusing on recursively breaking high density tiles into smaller ones. Objects that exceed the borders of a partition are replicated and a post-processing step is employed in order to eliminate duplicate results. The join process takes place at the *Reduce* phase by utilizing the R*-Tree indexes that are created and loaded in-memory at query time for each of the relations. Another system that copes with the problem of spatial join is [26]. This approach first re-partitions the data by taking into account load balancing and spatial locality. In more detail the data are sampled and an index (Grid File, R-Tree or R+-Tree) is created which will set the boundaries of each partition. During, the re-partitioning phase global and local indexes are created and stored in HDFS. The join takes place in the *Map* phase by utilizing the local indexes. Concerning borderline objects, a duplicate avoidance method is applied. An approach that enhances [26] with the functionality of identifying closest pairs of points is presented in [34].

Multidimensional similarity join is also related to our work. In [87, 86] the problem of distance range join is studied, which is probably the most common case of similarity join. In this approach the data is iteratively partitioned similarly to the Quickjoin algorithm [43], which results in having multiple MR jobs in order to get the final results. The problem of high dimensional similarity joins on massive datasets using MapReduce is tackled in [54]. In [78] the problem of ϵ-distance similarity self-join on vector data is tackled by employing in the *Map* phase a fixed size grid with cell width ϵ and assign the data to the corresponding cells. In order to compute the ϵ-neighborhood of each cell only the adjacent cells are needed. In order to reduce the replication of data they avoid taking into account all the adjacent cells. Despite this, the algorithm used to perform the join in the *Reduce* phase is still a Nested Loop Join. [32] is an extension of [78] for medium- to high-dimensional spaces where the full d-dimensional space is broken down to k dimension groups, the join is performed in each group and then the results are merged. Unlike [78], they try to cope with the skewness of data by starting with a very fine grid and merging cells until a balanced grid is created. Similarly to [78] the join process is still a Nested Loop Join.

2.4 Mining Mobility Data

In recent years, an increased research interest has been observed in knowledge discovery out of mobility data. Towards this direction, several mining methods have been proposed, which can be categorized to *co-movement pattern discovery*, *trajectory clustering*, *sequential patterns* and *periodic patterns*.

2.4.1 Co-movement Pattern Discovery

An interesting line of research includes works that aim to discover several types of collective behavior among moving objects, forming a group of objects that moves together for a certain time period. One of the first approaches in this direction introduced the concept of flocks. A flock [48, 12, 96] in a time interval I, where I spans for at least k successive timepoints, consists of at least m objects, such that for every timepoint in I, there is a disk of radius r that contains all m entities. If the objects change during the given interval, a kind of varying-flock is formed. Based on this idea, the notion of a moving cluster was introduced [45], which is a sequence of clusters $c_1, \ldots, c_k$, such that for each timestamp i, c_i and c_{i+1} share a sufficient number of common objects. An extension of the flocks pattern is the convoy pattern [44, 60] that is a group of objects that has at least m objects, which are density-connected with respect to a distance threshold e, during k consecutive timepoints. However, trajectories of real-world moving objects may meet together at some, nevertheless non-consecutive timepoints. To meet this real-world requirement, a swarm[51] is a collection of moving objects with cardinality at least m, that are part of the same cluster for at least k timepoints. It is important to note that the k timestamps are not required to be consecutive. The traveling companion [92] pattern is an approach for the online detection of convoy and swarm patterns from trajectories that arrive as a stream to the system. The gathering pattern [107, 108] relaxes the constraints of the above-mentioned patterns by allowing the membership of a group to evolve gradually. Each cluster of a gathering should contain at least p participators, which are the objects appearing in at least c clusters of this gathering. The gathering pattern is used to detect events, thus, it requires that the region and its shape where the gathering takes place is more-or-less stable. Another approach that relaxes the globally consecutive timestamp constraint is the platoon pattern [50], which only requires that the timestamps are locally consecutive. In other words, platoon patterns allow gap(s) in timestamps, but the consecutive time segments must have a minimum length.

However, all of the aforementioned approaches are centralized and cannot scale to massive datasets. In this direction, the problem of efficient convoy discovery was studied both in centralized [60] and distributed environment by employing the MapReduce programming model [59]. An approach that defines a new generalized mobility pattern is presented in [28]. In more detail, the general co-movement pattern (GCMP), is proposed, which models various co-movement patterns in a unified way and is deployed on a modern distributed platform (i.e., Apache Spark) to tackle the scalability issue. Moreover, GCMP detector is implemented in UlTraMan [23], an efficient platform for trajectory data management and analytics techniques. An approach that tries to tackle the problem of generalized co-movement pattern detection in an online fashion, on streaming trajectories, is proposed in [16], by utilizing Apache Flink, which is designed for efficient distributed streaming data processing.

Even though all of these approaches provide explicit definitions of several mined patterns, their main limitation is that they search for specific collective behaviors, defined by respective parameters. Furthermore, most of the aforementioned approaches operate at specific predefined temporal “snapshots” of the dataset, thus ignoring the route of each moving object between these “snapshots”.

2.4.2 Trajectory Clustering

Another line of research, tries to discover groups of either entire or portions of trajectories considering their routes. In [33], the authors proposed probabilistic algorithms for clustering entire trajectories using a regression mixture model. Subsequently, unsupervised learning is carried out by using EM algorithm to determine the cluster memberships in the model. Except from this probabilistic approach, researchers have followed two other directions. The first transforms trajectories to a multi-dimensional space and then apply well-known clustering algorithms such as k-means, BIRCH [106], CURE [38], DBSCAN [27] and [8], which are tailored to work with point data, thus applying them to trajectory data is not possible. Unfortunately, it has been shown [56] that such an approach based on k-means and hierarchical clustering algorithms leads to results of very poor quality.

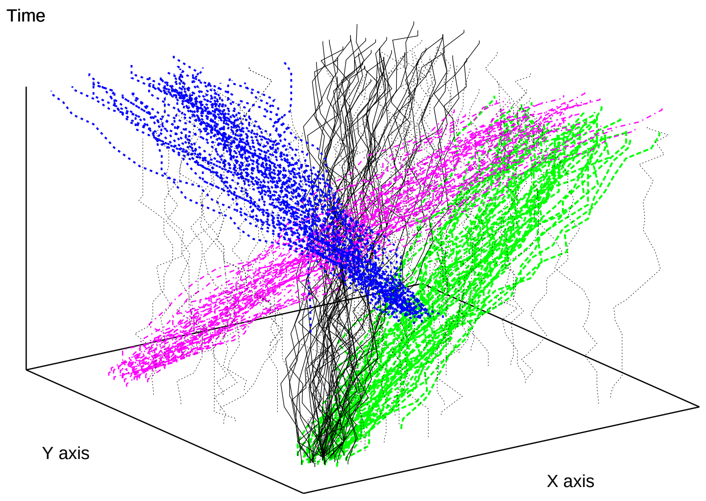


Figure 2.3: An example of T-OPTICS result [56]: The algorithm is able to separate four clusters (in black, green, blue, purple) as well as detect a few outliers (in grey).

Alternatively, another approach is to define an appropriate similarity function and embed it to an extensible clustering algorithm. In this direction, there are several approaches whose goal is to group whole trajectories, including T-OPTICS [56] (Figure 2.3), that incorporates a trajectory similarity function [31] into the OPTICS [8] algorithm. the vector field k-means trajectory clustering technique [30] whose central idea is to use vector fields to induce a notion of similarity between trajectories letting the vector fields themselves define and represent each cluster. CenTR-I-FCM [67], a variant of Fuzzy C-means, proposes a specialized similarity function that aims to tackle the inherent uncertainty of trajectory data. Both of the last two approaches propose specialized similarity functions having as goal to tackle the inherent uncertainty of trajectory data. Lately, another entire-trajectory clustering approach tackling uncertainty has been introduced in [42] where a pattern mining framework has been proposed for discovering trajectory routes that represent the frequent movement behaviors of a user. The approach exploits on a similarity measure for trajectories with silent durations (i.e., the time durations when no data points are available to describe the movements of users). This is used in a clue-aware clustering algorithm, where clues are some spatially and temporally close data points that capture certain common partial movement behaviors of the user. In [101] a multi-kernel-based estimation process leverages both multiple structural information within a trajectory and the local motion patterns across multiple trajectories in order to address challenges in case of large variations within a cluster and ambiguities across clusters. Another line of research identifies clusters of trajectories [47] that move over a fixed network. However, this assumption is valid only for vehicles moving in an urban environment.

A slightly different approach is the one of incremental trajectory clustering. The "Trajectory Clustering using Micro- and Macro- clustering" (TCMM) framework [53] is an incremental method that consists of two parts: (i) online micro-cluster maintenance and (ii) offline macro-cluster creation. The online part first simplifies trajectories by partitioning them into 2D line segments to find the spatial clusters of subtrajectories; then, micro-clusters of the partitioned trajectories are computed and maintained incrementally. Micro-clusters hold and summarize similar trajectory partitions at very fine granularity levels. The offline part performs macro-clustering on the set of micro-clusters rather than on all trajectories when a user requests so. The main characteristics of TCMM are: (i) TCMM maintains and operates on summaries of trajectories (i.e. micro-clusters) only; (ii) TCMM applies spatial clustering on directed line segments (using [49]); (iii) the partitioning of the trajectories in TCMM is actually a simplification step taking place per trajectory, i.e. without global criteria; (iv) TCMM targets at the entire lifespan of the database so as to identify global patterns without temporal constraints.

Nevertheless, trajectory clustering is a computationally intensive operation and centralized solutions cannot scale to massive datasets. In this context, [21] introduces a scalable GPU-based trajectory clustering approach which is based on OPTICS [8]. Moreover, [79] attempts to identify frequent movement patterns from the trajectories of moving objects. More specifically, they propose a MapReduce approach by employing quadtree-based hierarchical grid in order to discover complex patterns of different granularity. In [41] the authors tackle the problem of parallel trajectory clustering by utilizing the MapReduce programming model and Hadoop. They adopt an iterative approach similar to k-means in order to identify a user-defined number of clusters, which leads to a large number of MapReduce jobs.

Nonetheless, discovering clusters of complete trajectories can overlook significant patterns that might exist only for portions of their lifespan. To deal with this, another line of research has emerged, that of *Subtrajectory Clustering*. The predominant approach here is TraClus [49] (Figure 2.4), a partition-and-group framework for clustering 2D moving objects (i.e. TraClus ignores the time dimension) that enables the discovery of common subtrajectories. The algorithm first partitions trajectories to directed segments (i.e., subtrajectories) whenever the shape of a trajectory changes significantly, by employing the minimum description length (MDL) principle. Subsequently, the resulting subtrajectories are clustered by employing a modified version of the DBSCAN algorithm, which is applicable to directed segments. Finally, for each identified cluster the algorithm calculates a "fictional" representative trajectory that best describes the corresponding cluster.

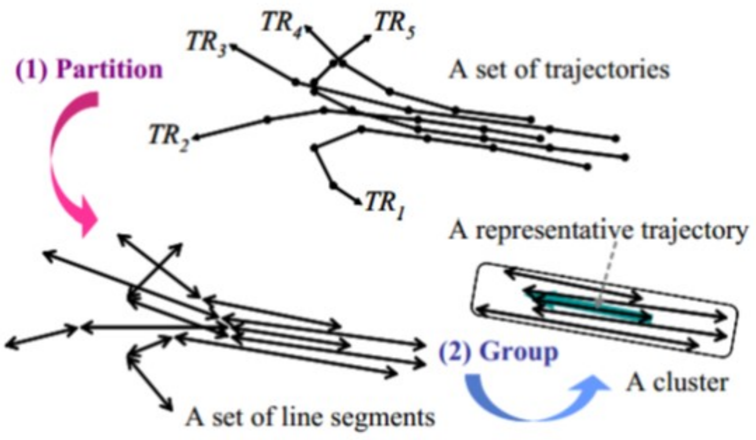


Figure 2.4: Overview of TRACLUS [71].

An alternative viewpoint to the problem of subtrajectory clustering is presented in [3], where the goal is to identify "common" portions between trajectories, with respect to some constraints and/or objectives, cluster these "common" subtrajectories and represent each cluster as a pathlet, which is a point sequence that is not necessarily a subsequence of an actual trajectory. A pathlet can be viewed as a portion of a path that is traversed by many trajectories. In order to solve this problem, the authors in [3] prove that this problem is NP-Hard and propose some approximation algorithms with theoretical guarantees, concerning the quality of the solution and the running time. Similarly, in [112] the goal is to identify corridors, which are frequent routes traversed by a significant number of moving objects. As already mentioned, all of the above subtrajectory clustering approaches are centralized and cannot scale to the size of today's trajectory data.

2.4.3 Sequential Pattern Discovery

Sequential pattern mining is an important data mining problem with broad applications. Given a set of sequences, where each sequence consists of a list of elements and each element consists of a set of items, and given a user-specified *min_support* threshold, sequential pattern mining is to find all frequent sub-sequences, i.e., the sub-sequences whose occurrence frequency in the set of sequences is no less than *min_support.* Additionally, further constraints may be integrated in the sequential pattern process to allow finding more interesting patterns or to indicate more precisely the types of pattern to be found (e.g., time constraints in between two consecutive items in a pattern).

One such method follows the pattern-growth approach. In that the search space is explored using a depth-first search, like in [15, 4]. It starts from sequential patterns of length one and proceeds by recursively appending items to patterns to create larger patterns (pattern-growth).

2.4.4 Data-driven predictive analytics

Predictive analytics is a scientific domain that aims at the extraction of valuable knowledge from data and the utilization of it in order predict future behavioural patterns and trends. When dealing with data that represent the movement of objects, predictive analytics can be of great importance since they can assist an analyst to predict events, such as collision prediction, traffic prediction etc. Towards this direction there have been several efforts, such as [95, 73, 74], that try to predict the future location of an object by utilizing extracted mobility patterns from historical data.

Part II

In-DBMS Centralized Algorithms and Techniques

3 In-DBMS Sampling-based Subtrajectory Clustering

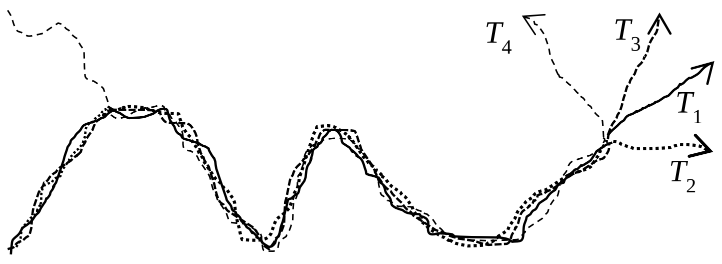
In this chapter, we propose an efficient in-DBMS solution for the problem of subtrajectory clustering and outlier detection in large moving object datasets. The method relies on a two-phase process: a voting-and-segmentation phase that segments trajectories according to a local density criterion and trajectory similarity criteria, followed by a sampling-and-clustering phase that selects the most representative subtrajectories to be used as seeds for the clustering process. Our proposal, called S^2T-Clustering (for Sampling-based SubTrajectory Clustering) is novel since it is the first, to our knowledge, that addresses the pure spatiotemporal subtrajectory clustering and outlier detection problem in a real-world setting (by 'pure' we mean that the entire spatiotemporal information of trajectories is taken into consideration). Moreover, our proposal can be efficiently registered as a database query operator in the context of an extensible DBMS (namely, PostgreSQL in our prototype implementation). The effectiveness and the efficiency of the proposed algorithm are experimentally validated over synthetic and real-world trajectory datasets, demonstrating that S^2T-Clustering outperforms an off-the-shelf in-DBMS solution using PostGIS by several orders of magnitude. The original content of this chapter appears in [70].

3.1 Introduction

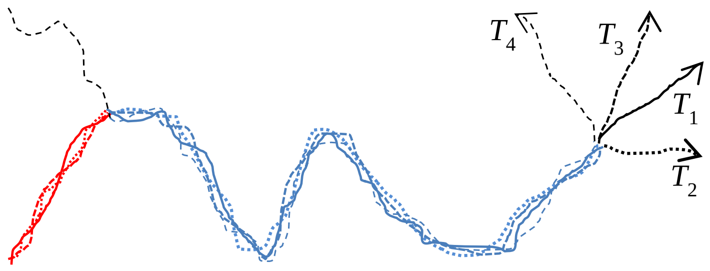
Knowledge discovery in mobility data [36, 72, 109, 102] exposes patterns of moving objects exploitable in several fields. For instance, in both mature (transportation, climatology, zoology, etc.) and emerging domains (e.g., mobile social networks), scientists work with mobility-aware (mostly GPS-based) data, resulting in trajectories of moving objects stored in MODs. Although during the recent years, there have been made significant achievements in the field [36, 72, 109, 102], ongoing research calls for new methods aiming at deeper comprehension and analysis of mobility. For instance - and acting as motivation of this work - enhancing MOD engines, such as Secondo [19] and Hermes [66], with data mining operators is challenging [36, 72] and is subject to the indexing extensibility interface of the corresponding ORDBMS on which they are implemented (see GiST [40, 46], for example). In the literature of trajectory-based mobility data mining, one can identify several types of mining models used to describe various collective behavioral patterns. As such, there exist works that identify various types of clusters of moving objects [33, 49, 56, 66] and variations [12, 44, 50, 107]. Related line of research is the one that builds representatives out of a trajectory dataset, either by generating artificial data [49, 67] or by sampling the dataset itself [68, 62].

Focusing on trajectory clustering, the majority of related work proposes a variety of distance functions, utilized by well-known clustering algorithms to identify collective behavior among whole trajectories [56, 67, 65]. A parallel line of research tries to discover local patterns in MOD, i.e. patterns that are alive only for a portion of moving objects' lifespan: some of those techniques simplify the given trajectories, however focusing on the spatial and ignoring the temporal dimension, such as TRACLUS [49], which is considered as the current state-of-the-art subtrajectory clustering technique.

Figure 3.1 illustrates a working example that motivates our research: a dataset consisting of four trajectories, $T_1, \ldots, T_4$. (In this figure, the time dimension is ignored for visualization reasons.) Among the subtrajectories that compose the dataset, our goal is to identify two clusters (in red and blue, respectively) and five outliers (in black). In particular, the first (red) cluster consists of the tails of trajectories T_1, T_2 and T_3, the second (blue) cluster consists of the main bodies of trajectories T_1, T_2, T_3 and T_4, while the rest portions of the trajectories (namely, the tail of T_4 and all four heads) are recognized as outliers. Such clustering sounds impossible to be achieved by TRACLUS. This is due to the inherent design of that algorithm that, as delineated by the authors, discovers linear patterns only and fails to identify complex (e.g., snake-like) patterns like the ones that appear in Figure 3.1. In other words, when applied to this dataset, TRACLUS would eventually discover five to six linear clusters (one new cluster each time the snake-like motion changes direction). On the contrary, we wish to be able to follow these direction changes without assuming underlying constraints on the complexity of the shape of subtrajectories found nor posing geometrical and temporal constraints, in terms of algorithm parameters, as those required



(a)



(b)

Figure 3.1: (a) a set of 4 trajectories; (b) the set split in 2 clusters (in red and blue) and 5 outliers (in black).

by related work, e.g., [12, 44]. For those having experimented with those techniques, parameters like disc radius, minimum duration and cardinality of patterns, are hard to be set in advance. For instance, a small detour of an object belonging to one of the clusters, would probably result in either the lack of those patterns or the formation of smaller ones. Inspired by the above, in this chapter we study an important problem in the mobility data management and exploration domain [72], that of subtrajectory clustering and outlier detection. Informally, we aim at a methodology that builds clusters around (and detects outliers far away from) appropriately selected subtrajectories that preserve the properties and the mobility patterns hidden in a MOD, as much as possible. Towards this goal, we introduce a novel clustering methodology exploiting on the voting, segmentation and sampling concepts proposed in [62]. More specifically, we devise an efficient voting process that allows us to describe the 'representativeness' of a trajectory in a MOD as a smooth continuous descriptor [62]. Using these descriptors (their 'representativeness'), we result in the automatic segmentation of trajectories into 'homogeneous' subtrajectories. Next, a deterministic sampling procedure selects only those subtrajectories that optimally describe the entire MOD. Finally, we devise a method for subtrajectory clustering driven by the aforementioned representative sample of subtrajectories.

The design of such a clustering methodology is subject to two indispensible requirements that challenged our research: we seek for (a) an efficient and scalable solution that (b) should be able to operate on a real-world DBMS rather than being an ad-hoc implementation using a sophisticated access method. This is in order for the proposal to be practical and useful in real-world application scenarios, where concurrency and recovery issues are taken into consideration. Both requirements call for a MOD engine; therefore, our proposal is implemented as a query operator in Hermes [1], implemented on top of PostgreSQL. To our knowledge, it is the first time in the literature that GiST is used to index trajectory-based mobility data for the above purposes. Therefore, we argue that this is an important step towards bridging the gap between MOD management and mobility data mining, as state-of-art approaches [52, 99, 35] could make use of the efficiency and the advantage of our proposal to execute in-DBMS clustering via simple SQL. Our contribution is summarized below:

- we formulate the problem of subtrajectory clustering (and outlier detection) in a MOD as an optimization problem;
- we propose an efficient solution, the so-called S^2T-Clustering algorithm, driven by a deterministic sampling methodology, with the number of clusters being automatically detected by the algorithm;
- in order to speed up clustering tasks in MOD systems, we implement S^2T-Clustering as a query operator over an extensible DBMS, namely PostgreSQL, based on access methods that exploit on the GiST indexing extensibility interface. (For validation purposes, we also implement S^2T-Clustering using PostGIS, an off-the-shelf in-DBMS alternative solution.)

The rest of the chapter is organized as follows: Section 3.2 formulates the problem of subtrajectory clustering (and outlier detection). Sections 3.3 and 3.4 present our proposal and its in-DBMS realization, respectively. Experimental results that evaluate S^2T-Clustering using synthetic and real trajectory datasets from urban and vessel traffic domains are provided in Section 3.5. Section 3.6 concludes the chapter.

3.2 Problem Formulation

Let $D = T_1, T_2, \ldots, T_N$ be a dataset consisting of N trajectories of moving objects (we assume that the objects move in the xy-plane). Let $p_{k,i} = (x_{k,i}, y_{k,i}, t_{k,i})$ be the i-th sampled point, $i \in 1, 2, \ldots, L_k$ of trajectory T_k, $k \in 1, 2, \ldots, N$, where L_k denotes the length of T_k (i.e. the number of points it consists of), the pair $(x_{k,i}, y_{k,i})$ and $t_{k,i}$ denote the 2D location and the time coordinate of point $p_{k,i}$, respectively. We consider linear interpolation between two successive sampled points, $p_{k,i}$ and $p_{k,i+1}$, so that each trajectory turns out to be a sequence of 3D line segments, $e_{k,i} = (p_{k,i}, p_{k,i+1})$, of cardinality L_{k-1}, where each segment represents the continuous movement of the object during sampled points. Table 3.1 summarizes the definitions of the symbols used in this chapter.

Table 3.1: Table of Symbols used in Chapter 3

Symbol	**Definition**
D	A dataset, $D = T_1, \ldots, T_N$, of N trajectories
T_k	k-th trajectory of D
$p_{k,i}$	i-th point of trajectory T_k, $p_{k,i} = (x_{k,i}, y_{k,i}, t_{k,i})$
L_k	Number of points forming trajectory T_k
$e_{k,i}$	i-th (3D) line segment of T_k, $e_{k,i} = (p_{k,i}, p_{k,i+1})$
LP_k	Number of subtrajectories partitioning T_k
P_k	Set of the subtrajectories partitioning T_k
$P_{k,i}$	i-th subtrajectory of trajectory T_k
P	Set of subtrajectories in dataset D, $P = \cup P_k$
V_k	Voting descriptor of trajectory T_k
V	Set of voting descriptors in dataset D, $V = \cup V_k$
$VP_{k,i}$	Voting descriptor of subtrajectory $P_{k,i}$
$Nl_{k,i}$	Normalized lifespan descriptor of subtrajectory $P_{k,i}$ w.r.t. lifespan of T_k
C	Clustering of subtrajectories in M clusters, $C = C_1, \ldots, C_M$, $C_i \subset P$, $Ci \cap Cj = \emptyset$, $i \neq j$
S	Sampling set of representatives, $S = R_1, \ldots, R_M$, $S \subset P$, with subtrajectory R_j representing cluster C_j
M	Cardinality of C (and S)
$SR(S)$	Representativeness function of S
$V(P_{k,i}, R_{k,i})$	Voting descriptor of $P_{k,i} \in P - S$ w.r.t. subtrajectory $R_j \in S$
Out	Set of outlier subtrajectories, $Out = P - C$

Informally, the objective of subtrajectory clustering is to partition trajectories into subtrajectories and then form groups of similar ones, while at the same time, separating those that cannot fit in a group (called outliers). However, searching for entire trajectory similarity may be misleading since real-world trajectories may be long and consisting of heterogeneous portions of movement [24]. On the other hand, clustering at the subtrajectory level sounds much more effective.

Rephrasing the previous discussion, if we consider trajectory T_k as a sequence of successive subtrajectories $P_{k,i}$ of arbitrary length ($P_{k,i}$ is the i-th subtrajectory of trajectory T_k), the objective of subtrajectory clustering (and outlier detection) is to partition subtrajectories into groups of similar ones and isolate the ones (called outliers) that are very dissimilar from the others. To achieve this, assuming a cluster is represented by its representative (or centroid) subtrajectory, we define clustering as an optimization problem where the optimization criterion is to maximize the following expression:

$$SRD = \sum_{\forall R_j \in S} \sum_{\forall P_{k,i} \in C(R_j)} \overline{V(P_{k,i}, R_j)} \tag{3.1}$$

The formula to be maximized, namely Sum of Representativeness of Dataset (SRD), uses set $S = \{R_1, \ldots, R_M\}$ of the representative subtrajectories and the corresponding clusters $C(R_j)$ built around them, and is calculated upon $\overline{V(P_{k,i}, R_j)}$, i.e. the mean similarity (or average number of votes, according to our terminology) of subtrajectory $P_{k,i}$ with respect to R_j. Given the above formulation, the problem in hand is formalized as follows:

Problem 3.1. ***(subtrajectory clustering in a MOD)***: *Assuming a dataset $D = T_1, T_2, \ldots, T_N$ consisting of N trajectories, where each of them is considered as a sequence P_k of successive subtrajectories of arbitrary length, the problem of subtrajectory clustering is defined as the task of partitioning the set $P = \cup P_k$ of subtrajectories into (i) a clustering $C = \{_C 1, \ldots, C_M\}$ of M clusters, $C_i \subset P$, $C_i \cap C_j = \emptyset$, $i \neq j$ (i.e. hard clustering), where each cluster is represented by its representative subtrajectory $R_j \in P$, $j = 1, \ldots, M$, and (ii) a set Out of outliers, by maximizing Equation 3.1.*

It is important to note that maximizing Equation 3.1 is not trivial at all since one has to define, among others, (i) the criterion according to which a trajectory is segmented into subtrajectories, (ii) the technique for selecting the set of the most representative subtrajectories, (iii) whose cardinality M is unknown, to name but a few challenging sub-problems.

3.3 The S^{2}T-Clustering Algorithm

In this section, we propose a solution for Problem 3.1 defined above, which is called S^{2}T-Clustering (for Sampling-based subtrajectory Clustering). Our proposal (listed in Algorithm 3.1) consists of two phases: first, we apply the so-called Neighborhood-aware Trajectory Segmentation (aka NaTS) method that is able to detect homogenized subtrajectories applying trajectory voting and segmentation; then, we apply the so-called Sampling, Clustering, and Outlier detection (aka SaCO) method that selects the most representative among the subtrajectories detected in the previous phase in order for them to serve as the seeds of the clusters to be produced.

Algorithm 3.1 S^{2}T-Clustering

```
1: Input: trajectory dataset $D = \{T_1, T_2, \ldots, T_N\}$, voting influence $\sigma$,
   threshold $\epsilon$
2: Output: sampling set $S$, clustering $C$, set of outliers $Out$.
   // initialization phase
3: Reset set $V$ of voting descriptors in $D$
   // NaTS phase (Neighborhood-aware Trajectory Segmentation)
4: for each trajectory $T_k \in D$ do
5:    Update set $V$ of voting descriptors in $D$ w.r.t. $T_k$ and $\sigma$
6:    Partition $T_k$ in set $P_k$ of subtrajectories w.r.t. $V_k$
   // SaCO phase (Sampling, Clustering, and Outlier detection)
7: Find sampling set $S$ consisting of the $M$ most representative subtrajectories
8: Using set $S$ and threshold $\epsilon$, partition $P = \cup P_k$ in a set $C$ of $M$ clusters
   and a set $Out$ of outliers
9: return $(S, C, Out)$
```

It is important to note that the number M of representatives (hence, the number of clusters) is not user-defined; rather, it is the algorithm that estimates it (in Line 7). As for parameters σ and ϵ that appear in Algorithm 3.1 (Line 5 and Line 8, respectively), σ controls how fast the voting influence decreases with distance, whereas ϵ acts as a lower bound threshold of similarity between representative and non-representative subtrajectories, thus deciding whether a (non-representative) subtrajectory will be flagged as outlier or not. These parameters will be explained in detail in the subsections that follow.

3.3.1 NaTS: Neighborhood-aware Trajectory Segmentation

We extend the concept of density-biased sampling (DBS), which was originally proposed for point datasets [45], to be applied to trajectory segments. According to DBS, the local density for each point of a set is approximated by the number of points in a surrounding region, divided by the volume of the region. In our case, adopting a voting process of trajectories in a MOD as defined in [62], we define the representativeness of a 3D trajectory segment $e_{k,i}$ of a given trajectory T_k to be the number of 'votes' this segment collects from other trajectories with respect to their mutual distance. The overall voting collected by a segment (a value ranging from 0 to N) has the physical meaning of the number of other trajectories that co-exist with the trajectory that segment belongs to, both spatially and temporally. Intuitively, the voting results can be post-processed in order for us to be able to identify homogeneous (with respect to representativeness) subtrajectories.

Formally, let V_k be the voting trajectory descriptor along the line segments of T_k, consisting of a series of L_{k-1} components. Each component $V_{k,i}$ of this vector corresponds to the number of votes ("representativeness" value) that segment $e_{k,i}$, $i \in \{1, \ldots, L_{k-1}\}$, collected by the segments of the other trajectories. This representativeness value is based on a distance function $d(e_{k,i}, e_j)$ between two line segments $e_{k,i}$ and e_j, $k \neq j$. This distance function is defined as the definite integral of the time-varying distance $D_j(t)$ between the two segments during their common lifespan $[t_{j,start}, t_{j,end})$, following the approach proposed in [31]:

$$d(e_{k,i}, e_j) = \int_{t_{j,start}}^{t_{j,end}} D_j(t)dt \tag{3.2}$$

As D_j follows a trinomial, this integral is efficiently approximated by the Trapezoid Rule:

$$(D_j(t_{j,start}) + D_j(t_{j,end})) \cdot (t_{j,start} - t_{j,end})/2$$

and can be computed in $O(1)$, as it has been already proved in [31].

Given the above distance function, the representativeness value is provided by the following voting function.

$$V(e_{k,i}, e_j) = e^{-\frac{d^2(e_{k,i}, e_j)}{2 \cdot \sigma^2}} \tag{3.3}$$

As already mentioned, parameter $\sigma > 0$ controls the "voting influence", i.e. how fast $V(e_{k,i}, e_j)$ decreases with distance. It also holds that $V(e_{k,i}, e_j)$ is bounded in $[0, 1]$: it gets value 1 when the distance of the two segments is zero (i.e. the segments are identical) while very high distance results in voting value close to zero.

After the voting process takes place, the trajectory segmentation process gets into action. The goal of this step is to partition each trajectory into homogeneous representativeness subtrajectories, irrespectively of their shape complexity (recall the discussion about the snake-like trajectories in Figure 3.1). In order to perform neighbourhood-aware trajectory segmentation, we adopt the Trajectory Segmentation Algorithm (TSA), proposed in [62]. In other words, the result of the voting process is given as input to TSA, which provides as output the subtrajectories along with their voting descriptors. More technically, let $P_{k,i}$, $i \in \{1, \ldots, LP_k\}$, be the i-th subtrajectory of T_k, where LP_k denotes the number of partitions of T_k. Then, $VP_{k,i}$ is the voting descriptor formed by the representativeness values of the segments that belong to $P_{k,i}$. In other words, $VP_{k,i}$ shows how many trajectories find themselves to be similar to $P_{k,i}$. The interested reader is referred to [62] for the technical details of TSA.

Back to the example of Figure 3.1, the NaTS phase results in segmenting trajectory T_1 into three subtrajectories (coloured red, blue, and black, respectively, in Figure 3.1(b)); similar for the other trajectories of the dataset. Thus, the overall result of this phase consists of 12 subtrajectories along with their voting descriptors.

3.3.2 SaCO: Sampling, Clustering, and Outlier detection

As already mentioned, trajectory segmentation aims to provide homogeneous subtrajectories according to their representativeness, i.e. with respect to their local similarity with other trajectories. On the other hand, the goal of subtrajectory clustering is to partition the dataset into groups (clusters) of similar subtrajectories. Therefore, in our proposal, we first select the appropriate sampling set S and then tackle the problem of clustering according to the following idea (quite popular, also in traditional data clustering): each subtrajectory in the sampling set is considered to be a representative around which a cluster will be formed. So, our goal is that the sampling set should contain highly voted trajectories of the MOD which, at the same time, would cover the 3D space occupied by the entire dataset as much as possible in order for Equation 3.1 to be maximized.

In order to achieve this goal, we propose the sampling to be done by maximizing a formula (see Equation 3.4) that would take into account the votes $VP_{k,i}$ collected by each subtrajectory. Formally, let S denote the sampling set, so that $S_{k,i}$ is one, if subtrajectory $P_{k,i}$ belongs to the sampling set, and zero otherwise. According to the previous discussion, the number of subtrajectories that are represented in the sampling set S, should be maximized. This is formalized in Equations 3.4-3.6.

$$SR(S) = \sum_{k=1}^{N} \sum_{i=1}^{LP_k} S_{k,i} \cdot SR_{gain}(k,i) \tag{3.4}$$

where

$$SR_{gain}(k,i) = \sum_{j=1}^{|P_{k,i}|} VP^{P}_{k,i,j} \cdot Nl_{k,i,j} \cdot (1 - VP^{S}_{k,i,j}) \tag{3.5}$$

$$Nl_{k,i,j} = lifespan(e_{k,i,j})/lifespanT_k \tag{3.6}$$

More precisely, $SRgain(k,i)$ expresses the gain in $SR(S)$ if we add $P_{k,i}$ in S, $|P_{k,i}|$ denotes the number of line segments of $P_{k,i}$, $VP^{P}_{k,i,j}$ and $VP^{S}_{k,i,j}$ denote the votes in P and the votes in S, respectively, of the j-th line segment of $P_{k,i}$ and are calculated according to Equation 3.3. As for $Nl_{k,i}$, it denotes the normalized lifespan descriptor of subtrajectory $P_{k,i}$ with respect to lifespan of T_k, namely $Nl_{k,i,j}$ is the fraction of the duration of the j-th line segment of $P_{k,i}$ with respect to whole lifespan of T_k.

For this purpose, we follow the ideas included in the subtrajectory Sampling Algorithm (SSA), proposed in [62]. However, SSA is not appropriate for an efficient in-DBMS solution, which is one of our main objectives. Thus, we keep the main characteristics of the algorithm and adapt it in order to meet our specifications (described in detail in Section 3.4.2). In principle, the input of sampling algorithm is the set P of all subtrajectories P_k, the set voting $VP_{k,i}$ and the normalized lifespan $Nl_{k,i}$ vectors of these subtrajectories, all provided by the NaTS phase. The output of the sampling step is the subtrajectory sampling set S consisting of M samples. Back to the example of Figure 3.1, this step results in selecting two subtrajectories (samples), one out of the three red and one out of the four blue subtrajectories.

As already mentioned, the population M of the samples is not user-defined; in contrary, it is dynamically estimated by SSA algorithm. As such, it provides a deterministic solution, in contrast to other probabilistic [45, 57] or user-supervised, explorative sampling techniques [7].

What follows is the clustering step, which takes into account the sampling set S and the vector of votes (i.e. representativeness) $V(P_{k,i}, R_j)$ between, on the one hand, the non-representative $P_{k,i} \in P - S$ and, on the other hand, the representative subtrajectories $R_j \in S$. Technically, $V(P_{k,i}, R_j)$ consists of $|P_{k,i}|$ elements, where each element represents the voting that takes place between the segments of $P_{k,i}$ and R_j. As illustrated in Equation 3.1, we use the mean value $\overline{(V(P_{k,i}, R_j))}$ of the vector values $V(P_{k,i}, R_j)$. Each of those values is computed by measuring the distance of the corresponding segment of $P_{k,i}$ from its nearest to R_j and then by applying the voting function of Equation 3.3. Thus, it holds that $0 \leq \overline{(V(P_{k,i}, R_j))} \leq 1$.

Concluding the discussion about Algorithm 3.1, in order to find the clusters that maximize Equation 3.1, the subtrajectories that are assigned to cluster $C(R_j)$ represented by subtrajectory $R_j \in S$, are the ones that fulfil the following property:

$$\begin{aligned} C(R_j) = \{P_{k,i} \in P - S : \overline{(V(P_{k,i}, R_j))} \geq \overline{(V(P_{k,i}, R_v))} \\ \forall R_v \in S \wedge \overline{(V(P_{k,i}, R_j))} \geq \epsilon\} \end{aligned} \tag{3.7}$$

and

$$C = \cup C(R_j) \tag{3.8}$$

On the other hand, the subtrajectories that are considered outliers (thus forming the outliers set Out) are those failing to be assigned to a cluster, formally:

$$Out = P - C \tag{3.9}$$

As already discussed, parameter ϵ controls how far from a representative a non-representative should be positioned in order for the latter to be flagged as outlier. Back to the example of Figure 3.1, the clustering process presented above results in two clusters, formed around the red and the blue, respectively, representative subtrajectory found in the sampling step. As a side effect, the black subtrajectories are left out of the two clusters, thus they are flagged as outliers.

3.4 S^{2}T-Clustering In-DBMS

In this section, we present our methodology for the efficient in-DBMS development of S^{2}T-Clustering algorithm proposed in Section 3.3.

3.4.1 NaTS in-DBMS

NaTS phase of S^{2}T-Clustering algorithm (Lines 4-6 in Algorithm 3.1) consists of two steps: (a) voting among trajectory segments and (b) trajectory segmentation based on the resulted voting descriptors. An efficient in-DBMS solution should focus on the voting step (Lines 4-5), since TSA [62] that implements the segmentation step (Line 6) poses no special challenges; it is an efficient in-memory algorithm applied only on the voting descriptor of a single trajectory.

Back to the voting step, to meet its requirement we need an algorithm that takes as input a dataset $D = T_1, T_2, \ldots, T_N$ of trajectories, a trajectory $T_k \in D$ and $\sigma > 0$ parameter, and provides as output a voting descriptor (vector) V_k consisting of L_{k-1} components, each corresponding to segment $e_{k,i}$, $i \in \{1, \ldots, L_{k-1}\}$, of trajectory T_k. For efficiency purposes, [62] implemented the demanding voting process by using an incremental nearest neighbour (INN) algorithm. However, given the specifications posed in the introduction of this chapter, INN is not a choice due to the fact that the access methods supported by real ORDBMS (e.g., the GiST interface in PostgreSQL) do not support the incremental paradigm. This implies that, in our case, we are directed to queries natively supported by ORDBMS, such as typical range and NN queries.

Let us now discuss the design and implementation options we have in-DBMS. Dataset D corresponds to a relation with tuples in the form $< t\_id, s\_id, e_{k,i} >$, where t_id (s_id) is the trajectory (segment, respectively) identifier and $e_{k,i}$ corresponds to the 3D segment, upon which a 3D-R-tree index is built. Nevertheless, this setting is straight-forwardly realized in the well-known PostGIS spatial extension of PostgreSQL using 3D GiST. (Note, however, that PostGIS handles time dimension as simply as a (third) z-spatial dimension, next to x- and y- dimensions.) An important issue has also to do with the realization of Equation 3.3 that provides the voting between two segments: theoretically, a segment may vote (though close to zero) even if it is found very far from the target segment. However, this is not realistic in DBMS implementations. As such, we introduce s_buffer, a spatial threshold for distance between two segments, above which there is no need to calculate this distance. In the case where the application user has limited knowledge about space-time properties of the dataset, this parameter can be tuned to be the maximum value resulting in a very low (close to zero) voting as computed by Equation 3.3. This is achieved as follows: by reversing Equation 3.3, we obtain Equation 3.10 that defines an upper bound for s_buffer.

$$d \leq \sqrt{-2\sigma^2 \cdot ln(\epsilon)} \tag{3.10}$$

Thus, d values higher than the upper bound set in Equation 3.10 are not expected to contribute to the quality of the clustering.

Given the above setting, voting can be implemented using at least two alternatives, called Baseline-I and Baseline-II, respectively. Baseline-I solution performs $\Sigma_k(L_{k-1})$ range queries in the 3D-R-tree, where each query window corresponds to the MBB of a segment, enlarged by s_buffer; hence, the total number of range queries equals to the total number of segments in D, a fact that turns this solution to be expensive in disk accesses. On the other hand, Baseline-II solution performs N range queries in the 3D-R-tree, where each query window corresponds to the MBB of a trajectory, again enlarged by s_buffer; hence, the total number of range queries equals to the number of trajectories in D. Obviously, the second solution is much cheaper in disk accesses regarding the index but, unfortunately, imposes a heavy refinement step because of the volume of the trajectory MBB. Anyway, both approaches need a refinement step to calculate voting descriptor $V_{k,i}$, which involves distance calculations.

In the following paragraphs, we present an alternative (third) approach for addressing the voting step, which is the most demanding step in S^{2}T-

Clustering algorithm and, as such, it needs special care. In particular, we follow a filter-and-refinement approach that utilizes a range-like query, called Trajectory Buffer Query (TBQ). TBQ takes as input a trajectory, enlarges it by s_buffer, and returns the segments that overlap with the sequence of the enlarged MBBs of the trajectory's segments. The TBQ rationale is to efficiently retrieve those segments in D that are "around" a given trajectory, where "around" is defined by s_buffer. Figure 3.2 illustrates the Trajectory Buffer TB_k of a trajectory T_k.

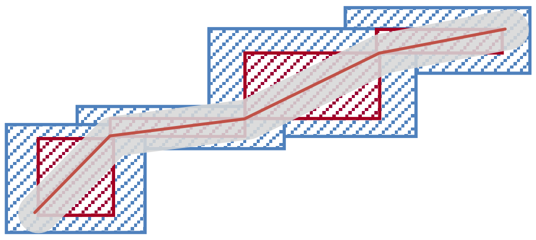


Figure 3.2: The Trajectory Buffer TB_k (i.e. the sequence of the blue MBBs) of a trajectory T_k.

It is obvious that our proposal follows a trajectory-based approach (i.e. similar to the Baseline-II technique), but for each trajectory it minimizes the filtering step by diminishing the dead space of the query, and thus minimizes the expensive refinement step. In turn, this implies changing the default search strategy of the 3D-R-tree over GiST that will reduce the time needed to compare a node entry with the trajectory buffer that is passed as predicate to the index. This is achieved by the Consistent method of the GiST extensibility interface [40], which contains the comparison logic between an index node entry of GiST and the trajectory buffer. Algorithm 3.2 outlines TBQ whereas Algorithm 3.3 presents the adapted Consistent method of the GiST interface.

Algorithm 3.2 Trajectory Buffer Query (TBQ)

```
1: **Input:** pg3D-R-tree root, trajectory $T_k$, parameter $s\_buffer$.
2: **Output:** set of segments that overlap with $TB_k$.
3: $TB_k \leftarrow TrajectoryBuffer(T_k, s\_buffer)$
4: $root.depth\text{-}first\text{-}search(Consistent,\ TB_k)$
```

Recall that *Consistent* decides whether the depth-first search should visit a child of the current entry or not (if the entry belongs to a non-leaf node) or, in case the entry belongs to a leaf node, checks whether to return the

Algorithm 3.3 Consistent

```
1: Input: Trajectory Buffer $TB_k$, current index entry $E$.
2: Output: Boolean.
3: if $E$ is in a leaf node then
4:    if $MBB(E.segment)$ overlaps $MBB(TB_k)$ then
5:       for each $MBB_i \in TB_k$ do
6:          if $E.segment$ overlaps $MBB_i$ then
7:             return true
8: else
9:    if $E.box$ overlaps $MBB(TB_k)$ then
10:      for each $MBB_i \in TB_k$ do
11:         if $E.box$ overlaps $MBB_i$ then
12:            return true
13:   return false
```

segment pointed by the leaf entry. After this remark, the depth-first search driven by *Consistent* in Algorithm 3.3 is easy to be followed: *Consistent* returns true if the MBB of the entry overlaps with one of the MBBs forming the trajectory buffer TB_k (Lines 7 and 12, for leaf and non-leaf nodes, respectively). Before this check takes place, a brute filtering is applied by checking whether the MBB of the entry overlaps the entire MBB of TB_k (Lines 4 and 9, respectively).

3.4.2 SaCO in-DBMS

In this section, we discuss the in-DBMS development of SaCO, i.e. the second phase of S^{2}T-Clustering. SaCO phase (Lines 7–8 in Algorithm 3.1) also consists of two steps: (a) sampling of the most representative subtrajectories (Line 7) and (b) clustering around samples and outlier detection (Line 8).

Regarding the sampling step, we adopt the SSA algorithm [62] as a starting point and we improve it with two crucial modifications, focusing on the efficiency and the quality, respectively, of the samples selected. The first improvement is that the voting method that is inherent in the sampling process follows the much more efficient approach presented earlier rather than the one presented in [62]. The second modification is about the selection of an even better set of representatives; as proposed in [62], SSA selects representatives as long as (a) the top-k number of representatives is less than a user-defined threshold (i.e. parameter M that acts as an upper bound for the selected representatives) and (b) the optimization criterion is satisfied (see

Equations 3.4 and 3.5). In fact, SSA selects the highly voted subtrajectories, while at the same time it tries to penalize subtrajectories that are very close to already selected representatives. Sometimes this automatic penalization fails, resulting to very similar representatives. In contrast, in our case, as the representatives are employed as cluster pivots, when a new representative is selected, it is further examined whether it is similar with one of the already selected representatives. In such a case, it is not selected and the algorithm evaluates the next candidate subtrajectory. The similarity criterion is the same with the one adopted for the clustering, i.e. Equation 3.7.

What follows is the final step, that of clustering and outlier detection. For this purpose, we follow an index-based, greedy approach that takes advantage of the TBQ query, which is applied on the results of the SSA algorithm, so as to form clusters around the sampled subtrajectories. To this end, we propose the so-called *Subtrajectory Clustering Algorithm* (SCA). SCA, listed in Algorithm 3.4, receives as input set P of subtrajectories, set S of representatives, as it was produced by the (modified) SSA, and threshold parameter ϵ. The output of the method is the final result of S^{2}T-Clustering, i.e. sets C and Out, with the clusters and outliers, respectively.

Algorithm 3.4 SCA

```
1: Input: set $P$ of subtrajectories, set $S$ of representatives, parameter $\epsilon$.
2: Output: set $C$ of clusters, set $Out$ of outliers.
3: $Out = P - S$
4: for each $R_j \in S$ do
5:   $C_j \leftarrow \{R_j\}$
6: for each $R_j \in S$ do
7:   $TBQ_j \leftarrow TBQ(Out, R_j, s\_buffer)$
8:   for each $e_{j,f} \in R_j$ do
9:     $TBQ_{j,f} \leftarrow$ overlaps($TBQ_j$, extend($e_{j,f}, s\_buffer$))
10:  for each $\underline{P_{k,i} \in \{TBQ_{j,f}\}}$, $f \in [1, |Rj|]$ do
11:    $V \leftarrow \overline{V}(Pk, i, R_j)$
12:    if $v > \epsilon$ and $v > old\_v_{k,i}$ then
13:      $C_j \leftarrow C_j \cup \{P_{k,i}\}$
14:      flag $P_{k,i}$ as clustered in $Out$
15:      $old\_v_{k,i} \leftarrow v$
16: for each $P_{k,i} \in Out$ do
17:   if $P_{k,i}$ is flagged as clustered then
18:     $Out \leftarrow Out - \{P_{k,i}\}$;
19: return $(C, Out)$
```

Initially, the subtrajectories are organized in two sets (implemented as relations in DBMS), one containing the sampling set sorted by the order of their selection and the other containing the remaining data, while each cluster is initialized by a representative subtrajectory from the sampling set. As such, each representative subtrajectory constitutes the first member (seed) of the corresponding cluster (Lines 3-5). Then, we apply a two-step filtering procedure so as to increase the efficiency of the algorithm. At the first step, for each cluster seed R_j, we apply a TBQ query, which returns the segments that are "close" to the cluster seed (Line 7). Subsequently, for each segment $e_{j,f}$ belonging to the specific representative R_j, we apply a spatiotemporal range query with the same spatial component as that of the TBQ query (Line 9). This spatiotemporal range query is performed in order to identify the segments that are "close enough" to $e_{j,f}$ and, hence, qualify to proceed to the voting procedure with respect to R_j. Subsequently, for each non-clustered $P_{k,i}$, we calculate the average voting that R_j receives (Line 11). By taking into account parameter ϵ discussed earlier, we assign it to cluster C_j mastered by R_j (Line 13) and mark it as clustered (Line 14). Through this process, in the case where $P_{k,i}$ belongs to the result of more than one TBQ searches, it is assigned to the representative that has achieved the highest voting.

3.5 Experimental Study

In this section, we present the results of our experimental study. All experiments were conducted on an Intel Xeon X5675 Processor 3.06GHz with 48GB memory, running on Debian Release 7.0 (wheezy) 64-bit. The proposed algorithms were implemented on top of a PostgreSQL 9.4 server with the default configuration for its memory parameters. We should clarify that in our implementation, which exploits on the extensibility interface given by PostgreSQL, we have defined and implemented from scratch datatypes and operands conforming to the whole discussion so far, resulting in the so-called Hermes@PostgreSQL [1], which is completely independent from PostGIS. This implies that the 3D-R-tree has also been implemented from scratch (on top of GiST); we call it pg3D-R-tree (see the input of TBQ in Algorithm 3.2).

A notable difference of our pg3D-R-tree from the PostGIS implementation of the 3D-R-tree is that, in our case, the entries of the leaf nodes are 3D segments rather than 3D boxes. This is an implicit assumption in the Consistent algorithm (see e.g., Line 4 in Algorithm 3.3), which allows us to avoid additional I/O operations. The outline of our experimental study is as follows: First, we study the robustness of S^{2}T-Clustering by using a synthetic dataset (where we know the ground truth) in order to (a) evaluate the sensitivity of our proposal with respect to various parameters and (b) validate whether our approach succeeds to discover the underlying clusters (and outliers). Then, a set of experiments is performed in order to evaluate the efficiency and scalability of S^{2}T-Clustering. These experiments are performed using three different approaches: the two baseline solutions and our solution based on TBQ, as they were presented in Section 3.4.

3.5.1 Datasets

For our experimental study we utilize the datasets SMOD, IMIS$_1$ and GeoLife that were presented in Section 1.5. Table 3.2 presents the statistics of the three datasets.

Table 3.2: Dataset Statistics

Statistic	**SMOD**	**GeoLife**	**IMIS$_1$**
# Trajectories	400	18668	5110
# Segments	35273	24159325	444570
Dataset Duration (hh:mm:ss)	0:02:00	1932 days 22:59:48	6 days 19:59:53
Avg. Sampling Rate (hh:mm:ss)	0:00:01	0:00:08	0:18:02
Avg. Segment Length (m)	8	72	1545
Avg. Segment Speed (m/s)	7.83	5.01	7.03
Avg. Trajectory Speed (m/s)	2.86	3.91	4.52
Avg. # Points per Trajectory	89	1295	88
Avg. Trajectory Duration (hh:mm:ss)	0:01:28	2:43:15	11:33:45
Avg. Trajectory Length (m)	691	93046	134,148

As already mentioned, SMOD is used for the ground truth verification. In more detail, the ground truth of the clusters that are hidden in SMOD can be inferred by the description of the dataset itself. In particular, eight clusters of subtrajectories (as well as a set of outliers) are identified. Table 3.3 lists the eight clusters along with their spatial (2nd column) and temporal projection (3rd column).

Table 3.3: The ground truth hidden in SMOD

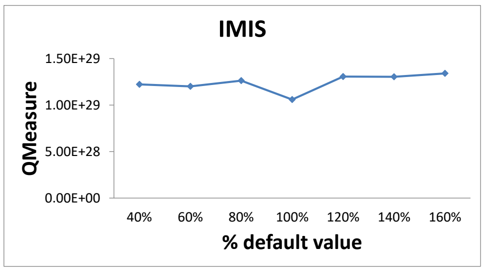
Cluster	Path	Time periods (clusters)
#1, #2	$A \to B$	[0, 0.2], [0.2, 0.7]
#3, #4	$B \to C$	[0.2, 0.8], [0.7, 1.2]
#5, #6	$B \to D$	[0.2, 0.52], [0.7, 1.2]
#7	$C \to B$	[0.8, 1]
#8	$D \to C$	[0.52, 1]

3.5.2 Quality of Clustering Analysis

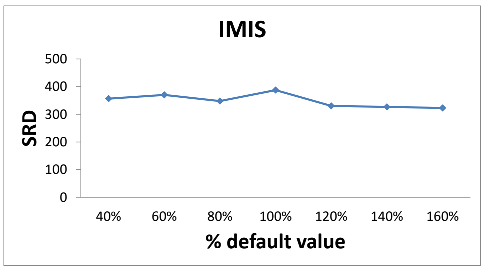
In this section, we perform a sensitivity analysis in order to explore the effect on the quality of clustering when setting different values on certain parameters. The quality of the clustering is calculated through two different measures: QMeasure [49] and SRD (see Equation 3.1). We should mention that the lower the QMeasure the higher the quality; on the other hand, the higher the SRD the higher the quality. Regarding parameter settings, as our approach shares similar concepts with the sampling methodology of [62], we followed the best practices presented in that work. More specifically, parameter σ was set to 0.1% of the dataset diameter while ϵ was set to 10^{-3}. Regarding s_buffer, it was automatically set according to Equation 3.10 as default value and we experimented with values around the default.

The first set of experiments is about the sensitivity of S^{2}T-Clustering with respect to s_buffer. Figure 3.3 illustrates the results over the IMIS$_1$ dataset. In particular, we used the default value (labelled 100% in the x-axis of the charts) as well as 6 values around it (labelled 40%, 60%, 80%, 120%, 140%, 160%). As one can easily observe, the quality of the clustering, measured either by QMeasure or SRD, remains more or less stable and follows the trend of the number of clusters identified. Moreover, in both QMeasure and SRD, the best quality appears when s_buffer is set to its default value (d).

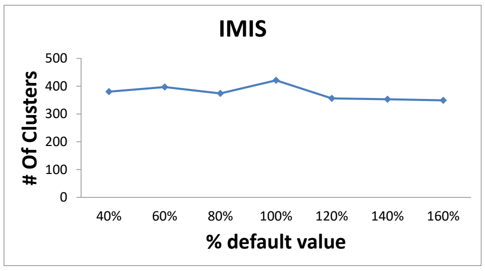
We repeated the same experiment over GeoLife and resulted in similar conclusions. Considering the above analysis, the value for s_buffer used in the remainder of our experimental study is the default value provided by Equation 3.10.



(a)



(b)



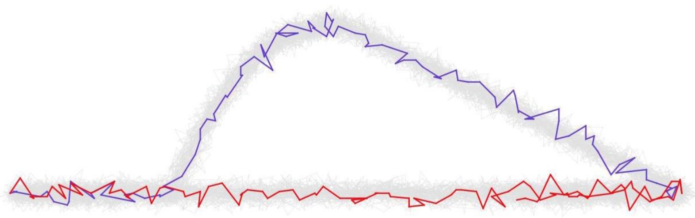
(c)

Figure 3.3: The effect on (a) QMeasure, (b) SRD, (c) the discovered number of clusters, when varying s_buffer parameter around its default value.

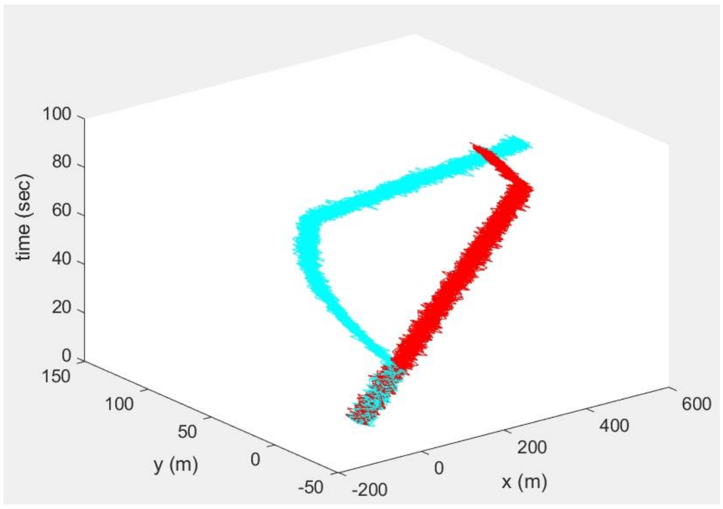
In a second set of experiments, we applied our proposal to the SMOD dataset, which is ideal for the purposes of testing the quality of our algorithm. In order to measure the stability of our method to noise effects, we have added Gaussian white noise of different Signal to Noise Ratio (SNR) levels, measured in db, to the spatial coordinates of SMOD. All the subsequent experiments have been repeated with SNR = 30db and SNR = 50db and the results were the same. Therefore, we present only the case with the SNR =30db.

First, we applied both S^2T-Clustering and TRACLUS [49] over a subset of SMOD that consists only of the trajectories that move throughout the whole lifespan of the dataset, thus limiting the ground truth to two clusters. In Figure 3.4(a) and Figure 3.4(c) we visualize only the representatives of each cluster, while in Figure 3.4(b) we provide a 3D illustration of the data used in the case of Figure 3.4(a). Note that S^2T-Clustering discovers the two clusters, while TRACLUS discovers several linear patterns; see Figure 3.4(a) vs. Figure 3.4(c).

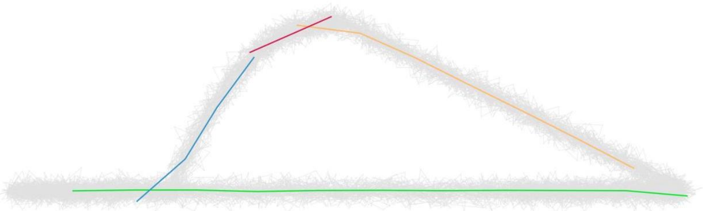
Subsequently, we applied both S^2T-Clustering and TRACLUS to the entire SMOD, for which we have knowledge of the ground truth. In Figure 3.5(a) and Figure 3.5(c), we present the results of the S^2T-Clustering and TRACLUS, respectively. Moreover, in order to better comprehend the temporal dynamics



(a)

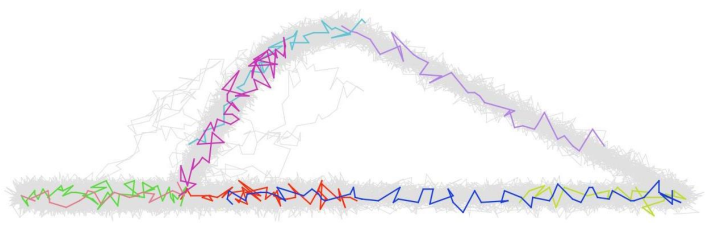


(b)

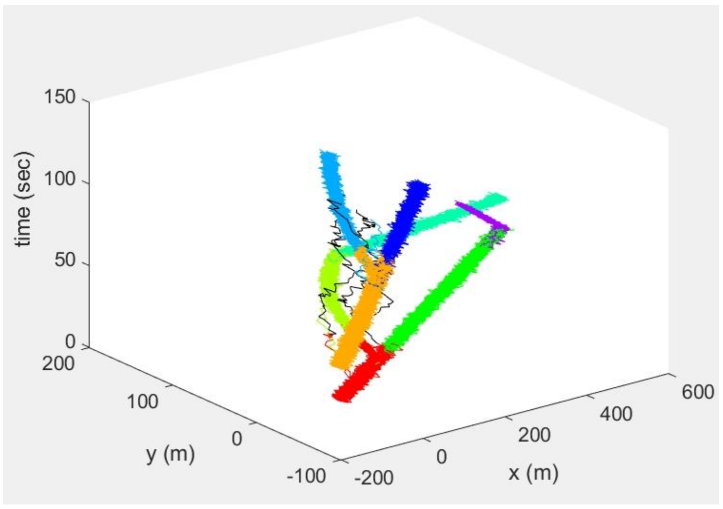


(c)

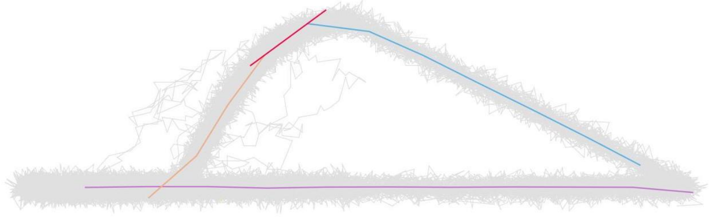
Figure 3.4: Visualization of the clusters' representatives provided by: S^{2}T-Clustering in (a) 2D and (b) 3D, (c) TRACLUS, when applied to a subset of SMOD consisting of 2 patterns.



(a)



(b)



(c)

Figure 3.5: Visualization of the clusters' representatives provided by: S^2T-Clustering in (a) 2D and (b) 3D, (c) TRACLUS, when applied to the entire SMOD consisting of 8 patterns.

of the dataset we provide a 3D illustration in Figure 3.5(b). According to this experiment, S^2T-Clustering effectively discovers all eight clusters (as well as the noisy subtrajectories, depicted in black color in Figure 3.5(b)), thus S^2T-Clustering is not affected by the trajectories' shape, yielding an effective and robust approach for the discovery of linear and non-linear patterns. On the contrary, TRACLUS fails to identify the hidden ground truth in this SMOD due to the fact that it ignores the time dimension. Interestingly, TRACLUS discovers almost the same sets of representatives when applied to either a subset of or the entire SMOD; see Figure 3.4(c) vs. Figure 3.5(c).

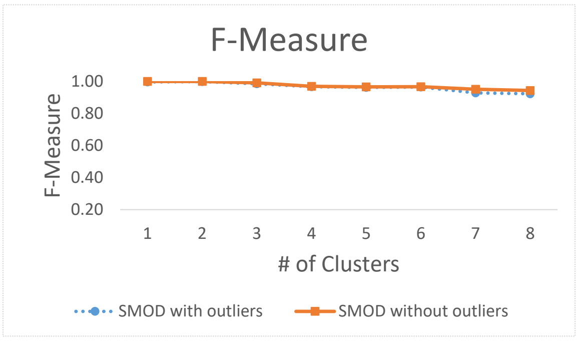
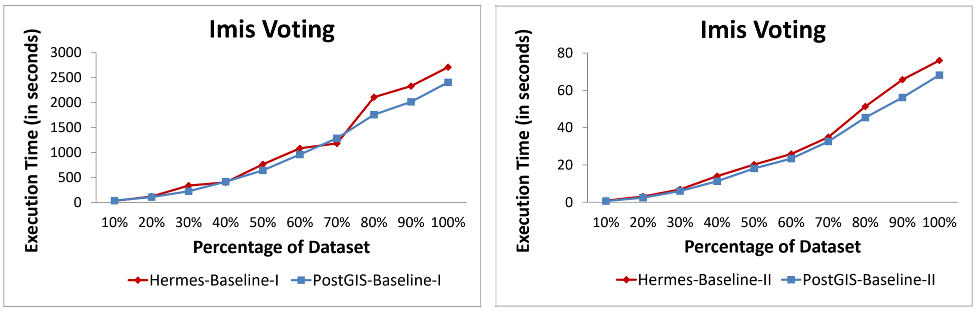


Figure 3.6: Quality of S^2T-Clustering w.r.t. number of clusters.

In order to evaluate the accuracy of our proposal in a quantified way, we further employed F-Measure in SMOD. In detail, we built 8 datasets, with the first consisting of the subtrajectories of the first cluster only, the second consisting of the subtrajectories of the first and the second cluster only, and so on, until the eighth dataset, which consisted of the subtrajectories of all eight clusters; all eight datasets appeared in two variations: including or not the set of outliers. For each dataset, we applied S^2T-Clustering and calculated F-Measure; Figure 3.6 illustrates this quality criterion by increasing the number of clusters. It is evident that S^2T-Clustering turns out to be very robust, achieving always precision and recall values over 92.3%, while the outliers are always detected correctly.

3.5.3 Efficiency and Scalability

In order to study the efficiency and scalability of our proposal we followed two competing approaches: Hermes@PostgreSQL [1], implemented according to the discussion in Section 3.4, vs. PostGIS extension of PostgreSQL that simulated the two baseline solutions presented in Section 3.4.1. We have noticed that the implementation of the 3D-R-tree in PostGIS suffers from rounding errors because it uses 32-bit IEEE floating-point numbers to store the coordinates [2]. In our experiments we observed that the MBB of a trajectory or a segment was always enlarged due to this rounding, thus making the overlap query in PostGIS return more segments than our implementation. Since this made the comparison between the two systems unfair, we simulated PostGIS inside Hermes, in other words, also the baseline solutions were simulated inside Hermes (thus, making all solutions run under the same framework). In the charts that follow, we denote the implementation of Baseline-I and Baseline-II solutions implemented both in Hermes and in PostGIS as Hermes | PostGIS-Baseline-I | II, i.e. four different implementations. In particular, Figure 3.7 illustrates the execution time of the voting step for the IMIS$_1$ dataset when varying the dataset size (i.e. the number of trajectories). Obviously, the two implementations present similar performance, with the PostGIS implementation performing slightly better mainly due to the fact that the size of index node entries in PostGIS (which uses 32-bit numbers for storing the temporal dimension) is slightly less than that of Hermes (which uses 64-bit numbers).



(a) (b)

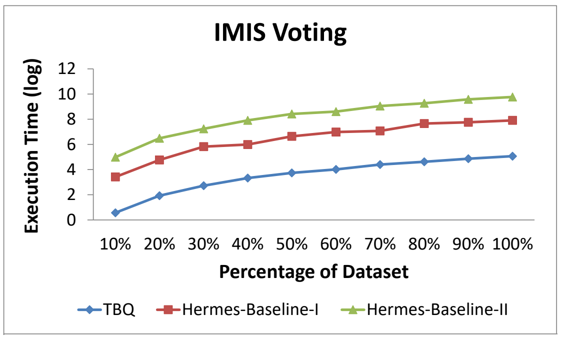
Figure 3.7: Comparing the performance of baseline solutions: (a) Baseline-I; (b) Baseline-II.

We repeated the same experiment with the GeoLife dataset and the results lead to similar conclusions, thus they are excluded due to space limitations. Based on the above results, in the remainder of the experimental study, the scalability study is conducted using the Hermes implementation of the algorithms. As illustrated in Figure 3.8(b), all three approaches (Baseline-I, Baseline-II and TBQ, presented in Section 3.4.1) perform similarly on the IMIS$_1$ dataset as far as it concerns the segmentation, sampling and clustering steps of the algorithm (please note that y-axis is at log scale). The crucial difference is at the expensive voting step, where TBQ significantly outperforms the two baseline solutions by almost two orders of magnitude;

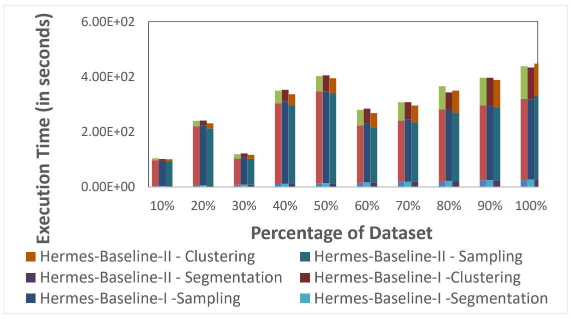
this is illustrated in Figure 3.8(a) whereas in Figure 3.8(c) we present the accumulated processing time. Due to the fact that the overall performance is dominated by the performance of the voting step, we further studied this step over the GeoLife dataset. As it can be observed in Figure 3.8(d), the behavior of the voting step of S^2T-Clustering over GeoLife is slightly different from that over IMIS_1. TBQ still outperforms both Baseline-I and Baseline-II solutions by several orders of magnitude, but in the case of GeoLife, Baseline-II outperforms Baseline-I. This can be explained by the fact that GeoLife consists of trajectories with significantly larger number of segments than IMIS_1 (recall the statistics in Table 3.2). This fact leads Baseline-I to perform considerably more lookups in the index.

3.6 Summary

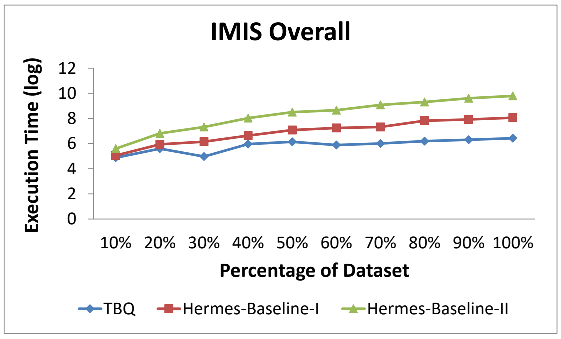
In this chapter, we discussed the problem of subtrajectory clustering and outlier detection in trajectory databases, aiming to take both space and time information into consideration. In particular, we proposed S^2T-Clustering that is novel not only because it solves the problem more effectively than the state-of-the-art (namely, TRACLUS), but also for an additional, quite important reason: our proposal is designed in-DBMS, i.e., it performs as a query operator in a real MOD engine over an extensible DBMS (namely, PostgreSQL in our current implementation). Having such functionality in their hands, data scientists are able to perform cluster analysis via simple SQL in real DBMS, where concurrency and recovery issues are taken into consideration. Moreover, our algorithm is boosted by an efficient index-based Trajectory Buffer Query (TBQ) that speeds up the overall process, resulting in a scalable solution, outperforming the state-of-the-art in-DBMS solutions supported by PostGIS by several orders of magnitude.



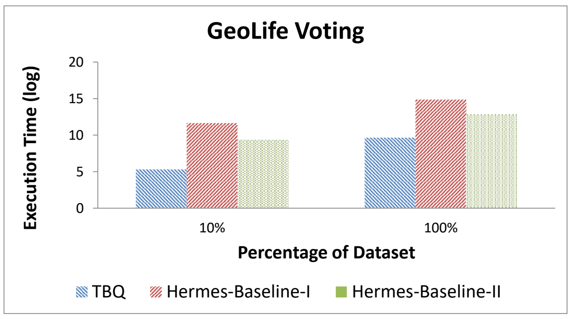
(a)



(b)



(c)



(d)

Figure 3.8: Step-by-step execution time of S^2T-Clustering: (a) voting over IMIS$_1$; (b) segmentation/sampling/clustering over IMIS$_1$; (c) overall over IMIS$_1$; (d) voting over GeoLife.

4 Temporal-constrained Subtrajectory Cluster Analysis

Finding a solution to the above described subtrajectory clustering problem is challenging; An even more challenging problem, than that of subtrajectory clustering, is how one can support incremental and progressive cluster analysis in the context of dynamic applications, where (i) new trajectories arrive at frequent rates, and (ii) the analysis is performed over different portions of the dataset, which might be repeated several times per analysis task. Towards this direction, in this chapter, we study the temporal-constrained subtrajectory cluster analysis problem, where the aim is to discover clusters of subtrajectories given an ad-hoc, user-specified temporal constraint within the dataset's lifetime. The problem is challenging because: (a) the time window is not known in advance, instead it is specified at query time, and (b) the MOD is continuously updated with new trajectories. To address this problem, we propose an incremental and scalable solution to the problem, which is built upon a novel indexing structure, called Representative Trajectory Tree (*ReTraTree*). *ReTraTree* acts as an effective spatio-temporal partitioning technique; partitions in *ReTraTree* correspond to groupings of subtrajectories, which are incrementally maintained and assigned to representative (sub-)trajectories. Due to the proposed organization of subtrajectories, the problem under study can be efficiently solved as simply as executing a query operator on *ReTraTree*, while insertion of new trajectories is supported. Our extensive experimental study performed on real and synthetic datasets shows that our approach outperforms a state-of-the-art in-DBMS solution supported by PostgreSQL by orders of magnitude. The original content of this chapter appears in [69].

4.1 Introduction

Nowadays, huge volumes of location data are available due to the rapid growth of positioning devices (GPS-enabled smartphones, on-board navigation systems in vehicles, vessels and planes, smart chips for animals, etc.). This explosion of data already contributes in what is called the Big Data era, raising new challenges for the mobility data management and exploration field [35, 72].

Efficient and scalable trajectory cluster analysis is one of these challenges [109, 102]. The research so far has focused on adapting well-known solutions that are effective for legacy data types to trajectory datasets. Thus, a typical approach is to transform trajectories to multi-dimensional (usually, point) data, in order for well-known clustering algorithms to be applicable. For instance, CenTR-I-FCM [67] builds upon a Fuzzy C-Means variant. Another approach is to focus on effective and efficient trajectory similarity search, which is the basic building block of every clustering approach. Once one has defined an effective similarity metric, she can adapt well-known algorithms to tackle the problem. For instance, TOPTICS [56] adapts OPTICS [8] to enable whole-trajectory clustering (i.e. clustering the entire trajectories), and TRACLUS [49] exploits on DBSCAN [27] to support subtrajectory clustering.

Subtrajectory clustering is a typical cluster analysis problem in Moving Object Databases (MOD). Recall Figure 3.1, i.e. a dataset consisting of four trajectories, $T_1, \ldots, T_4$ (please note that the time dimension has been ignored for visualization reasons). Upon this dataset, the goal of subtrajectory cluster analysis is to identify two clusters (coloured red and blue, respectively) and five outliers (coloured black).

Finding a solution to the above described subtrajectory clustering problem is challenging; what is even more challenging, is how one can support incremental and progressive cluster analysis in the context of dynamic applications, where (i) new trajectories arrive at frequent rates, and (ii) the analysis is performed over different portions of the dataset, and this might be repeated several times per analysis task.

As motivational example, consider the Location-based Services (LBS) scenario where LBS users transmit their trajectories to a central LBS server, e.g., when their trip is completed. From the server side, a MOD system is responsible for organizing user traces, aiming to support extensive (usually incremental and explorative) querying and mining processes. Since users (the data producers) transmit their location information in batch mode and asynchronously, the underlying data management framework should be able to handle this kind of information transmission. In other words, as we are especially interested in cluster analysis, the data server should be able to cluster users' trajectories in an incremental fashion. Clearly, the above techniques fail to meet such a specification.

Coming back to the example of Figure 3.1, two main challenges need to be confronted: (i) given the addition of a new trajectory in the existing set of four trajectories, how can cluster analysis be performed over the updated data without applying the (quite expensive) clustering process from scratch, and (ii) how could we organize these trajectories so as to retrieve clusters valid in an ad-hoc temporal period of interest, without re-applying the clustering for the user-defined temporal period? In this chapter, we address the challenge of efficient and effective *temporal-constrained subtrajectory cluster analysis*, by proposing an incremental and progressive solution to the problem. To this end, we propose a novel indexing scheme for large MODs, which is designed upon optimally selected samples of subtrajectories, called Representative Trajectories, hence the term *ReTraTree.* Each subtrajectory of this type acts as the representative of a group (cluster) of subtrajectories. Thus, *ReTraTree* may be considered as a data structure that organizes (sub-)trajectories in a hierarchical fashion, while having small, but in any case adaptable, memory footprint. Based on its design, *ReTraTree* is able to incrementally partition and cluster trajectories as they are inserted in the MOD. Interestingly, the actual clustering process for the user-defined temporal period of interest, called *Query-based Trajectory Clustering* (*QuT-Clustering*), is performed as simply as a query execution upon the *ReTraTree.*

The contributions of our work are summarized below:

- we introduce the temporal-constrained subtrajectory cluster analysis problem, which is a key problem for supporting progressive clustering analysis;
- we design *ReTraTree*, an efficient indexing scheme for large dynamic MODs, which is based on representative trajectories found in the dataset;
- as a solution to the problem of study, we devise *QuT-Clustering*, a subtrajectory clustering algorithm running as simply as a query operator upon *ReTraTree*;

- we facilitate incremental trajectory cluster analysis by exploiting the incremental maintenance of *ReTraTree* along with the query-based clustering approach of *QuT-Clustering*;
- we perform an extensive experimental study upon real and synthetic datasets, which demonstrates that our in-DBMS implementation outperforms a state-of-the-art PostgreSQL extension by several orders of magnitude.

The rest of the chapter is organized as follows: Section 4.2 formally defines the problem of temporally-constrained subtrajectory cluster analysis. Section 4.3 presents the *ReTraTree* structure and its maintenance algorithms while Section 4.4 puts *ReTraTree* in action, in other words it provides the QuT-Clustering algorithm, also providing a complexity analysis of the entire framework. Section 4.5 presents our experimental study. Section 4.6 concludes the chapter and outlines future research directions.

4.2 Problem Setting

In this section, we provide the necessary definitions and terminology. Table 4.1 summarizes the definitions of the symbols used in the chapter.

Definition 4.1. *(Voting between segments of two trajectories): Given two segments e and e' belonging to trajectories T and T', respectively, the voting function $V(e,e')$ that calculates the voting e receives by e' is given by Equation 4.1:*

$$V(e,e') = e^{-\frac{d^2(e,e')}{2\cdot\sigma^2}} \tag{4.1}$$

where the control parameter $\sigma > 0$ shows how fast the function (“voting influence”) decreases with distance.

Since Euclidean distance $D(t)$ is symmetric, distance $d(e,e')$ is symmetric as well. As such, it holds that $V(e,e') = V(e',e)$; it also holds that $0 \leq V(e,e') \leq 1$. If the two segments are almost identical, i.e. distance $d(e,e')$ is close to zero, the voting function gets value close to 1. On the other hand, high values of distance $d(e,e')$ result in voting close to zero.

We can generalize the above discussion to define the representativeness of a

Table 4.1: Table of Symbols used in Chapter 4

Symbol	**Definition**
D	A dataset, $D = T_1, \ldots, T_N$, of N trajectories
T	A trajectory of D, whose length is $\|T\|$ (in terms of number of points composing it)
$x_{t.s}(x_{t.e})$	Starting (ending) timestamp of the time-varying object x, e.g., $T_{t.s}$ ($T_{t.e}$) is the minimum (maximum) timestamp of trajectory T
l	Lifespan of D, namely the temporal period $[min(T_{t.s}), max(T'_{t.e}))$, $\forall T$, $T' \in D$
p_i	i-th (3D) point of trajectory T, $p_i = (x_i, y_i, t_i)$
e_i	i-th (3D) line segment of T, $e_i = (p_i, p_{i+1})$
l_i	Lifespan of line segment e_i, namely the temporal period $[t_i, t_{i+1})$
S	Set of subtrajectories partitioning trajectory T
S_i	i-th subtrajectory of trajectory T
$V(e, e')$	Voting function between two segments e and e' belonging to trajectories T and T', respectively
$V_T^{T'}$	Voting descriptor of trajectory T with respect to T'
V_D^{T}	Voting descriptor of trajectory D with respect to trajectory dataset T
V_T^{D}	Voting descriptor of trajectory T with respect to trajectory dataset D
$V_D^{D'}$	Voting descriptor of trajectory dataset D with respect to trajectory dataset D'
R	Sample of representative subtrajectories $R = R_1, \ldots, R_M$
C	Clustering of subtrajectories in M clusters, $C = \{C_{R_1}, \ldots, C_{R_M}\}$, $C_i \subset P$, $C_{R_i} \cap C_{R_j} = \emptyset$, $i \neq j$, with subtrajectory R_i representing cluster C_{R_i} of subtrajectories
M	Cardinality of C (and R)
Out	Set of outlier subtrajectories
W	The user-defined time window ($W \in l$) for which we want to discover the subtrajectory clusters

trajectory with respect to another trajectory. Notice that the definition that follows is applicable to subtrajectories as well (since a subtrajectory is itself a trajectory, essentially a set of consecutive segments).

Definition 4.2. *(Voting descriptor and average voting of a trajectory with respect to another trajectory): Given a trajectory T of length $|T|$ and another trajectory T', the voting descriptor $V_T^{T'}$ of T with respect to T' is a vector*

$$V_T^{T'} : (V(e_1, *), \ldots, V(e_{|T|-1}, *)) \tag{4.2}$$

of dimensionality $|T| - 1$ where wildcard '' corresponds to the segment of T' that minimizes distance $d(e_i, \cdot)$, $i = 1, \ldots, |T| - 1$. By $avg(V_T^{T'})$ we denote the average of the values of the vector $V_T^{T'}$ of trajectory T with respect to trajectory T'.*

Obviously, the voting descriptor is not symmetric, i.e. $V_T^{T'} \neq V_{T'}^{T}$.

Definition 4.3. *(Voting descriptor of a trajectory with respect to a trajectory dataset): Given a trajectory dataset D and a trajectory T of cardinality $|T|$, $T \notin D$, the voting descriptor V_T^D of T with respect to D is a vector*

$$V_T^D : (\sum_{T' \in D} V(e_1, *), \ldots, \sum_{T' \in D} V(e_{|T|-1}, *)) \tag{4.3}$$

of dimensionality $|T| - 1$ where wildcard '' corresponds to the segment of each T' in D that minimizes distance $d(e_i, \cdot)$, $i = 1, \ldots, |T| - 1$.*

Recall that (i) the vote a segment can receive by another segment is a value ranging from 0 to 1, according to Equation 4.1, and (ii) only one segment from each trajectory votes for a given segment of another trajectory, i.e. its nearest. This implies that the total voting - the sum of votes - received by a given segment is a value ranging from 0 (if all members of D vote 0) to N (if all members of D vote 1). To exemplify the above, back to the example of Figure 3.1, voting descriptor $V_{T_1}^{\{T_2, T_3, T_4\}}$ presents in general higher values than voting descriptor $V_{T_4}^{\{T_1, T_2, T_3\}}$ since T_1 is more centrally located than T_4 in the dataset.

Definition 4.4. *(Voting of a trajectory dataset with respect to a trajectory): Given a trajectory dataset D of cardinality N and a trajectory T of cardinality $|T|$, $T \notin D$, voting V_D^T of D with respect to T is a value*

$$V_D^T = \sum_{T' \in D} avg(V_{T'}^T) \tag{4.4}$$

that accumulates the average voting of all trajectories $T' \in D$ with respect to T.

Definition 4.5. *(Voting of a trajectory dataset with respect to another trajectory dataset): Given a trajectory dataset D of cardinality N and another (reference) trajectory dataset D' of cardinality N', $D \cap D' = \emptyset$ voting $V_D^{D'}$ of D with respect to D' is a value calculated as follows:*

$$V_D^{D'} = \sum_{T \in D'} avg(V_D^T) \tag{4.5}$$

Now, we define the *temporally-constrained subtrajectory clustering* problem that we address in this chapter. Let W represent a time window within the lifespan of D, i.e. $W \in l$. Further, let D_W denote the set of subtrajectories partitioning the trajectories in D, which are temporal-constrained within W. Formally:

Problem 4.1. *(Temporal-constrained subtrajectory clustering): Given (i) a trajectory database $D = T_1, \ldots, T_N$ of lifespan l, consisting of N trajectories of moving objects, and (ii) a time window W ($W \in l$), the temporal-constrained subtrajectory clustering problem is to find: (a) a set $C = \{C_{R_1}, \ldots, C_{R_M}\}$ of M clusters of subtrajectories, $C_{R_i} \in D_W$, $i = 1, \ldots, M$, around respective subtrajectories $R = \{R_1, \ldots, R_M\}$, $R_i \in C_{R_i}$, $i = 1, \ldots,$, called representative subtrajectories, and (b) a set Out of outlier subtrajectories, $Out \in D_W$, so that voting $V_{D_W - R}^R$ of dataset $D_W - R$ with respect to R is maximized:*

$$(R, C, Out) = argmax(V_{D_W - R}^R) \tag{4.6}$$

The above problem is quite challenging, for a number of reasons. First, the segmentation (or partitioning) of trajectories found in D in subtrajectories cannot be predefined nor is the result of a third-party trajectory segmentation algorithm, such as [14, 49, 53]. Instead, it is problem-driven: it is the clustering algorithm that solves the above problem that is responsible to find the best segmentation of trajectories into subtrajectories. Practically, it is the clustering algorithm that is responsible to detect the red and blue parts of trajectories in Figure 3.1, given that the analyst requires a clustering providing as time window W the whole lifespan of the dataset. Second, the optimization of the above scenario is a hard problem, since the solution space is huge. Third, one has to define the technique for selecting the set of the most representative subtrajectories, whose cardinality M is unknown. Fourth, as already discussed, in a real MOD setting, the solution should support incremental updates. Put differently, data updates should be accommodated as soon as they come and update the existing clusters at low cost, instead of performing a new clustering process from scratch. Finally and most importantly, since clustering is applied over different portions of the dataset, and this might be repeated several times per analysis task, the solution to the problem should be repeatable for all the different time windows W that are of interest during explorative analysis. This comprises a novel feature and a major contribution of our work, since existing solutions for subtrajectory clustering are not able to support progressive clustering analysis taking into account temporal constraints as filters.

4.3 The ReTraTree Indexing Scheme

We start this section with an overview of the *ReTraTree* indexing scheme (Section 4.3.1) and we continue with the algorithms that are necessary for its maintenance (Sections 4.3.2 – 4.3.4).

4.3.1 ReTraTree Overview

ReTraTree consists of four levels: the two upper levels operate on the temporal dimension while the 3rd level is built upon the spatiotemporal characteristics of the trajectories. The idea is to hierarchically partition the time domain by first segmenting trajectories into subtrajectories according to fixed equi-sized disjoint temporal periods, called chunks (1st level partitioning). Then, each chunk is organized into sub-chunks, which form a partitioning of subtrajectories within each chunk (2nd level partitioning). Notice that sub-chunks may overlap in time, i.e. they are not temporally disjoint.

Example 4.1. *Figure 4.1 illustrates six trajectories, $T_1, \ldots, T_6$ spanning in two days (called Day 1 and Day 2). The dataset is split into two chunks at day-level, with mauve (green) colored subtrajectories corresponding to the evolution of moving objects on Day 1 (Day 2, respectively). Furthermore, the chunk corresponding to Day 1 is subdivided to two sub-chunks, corresponding*

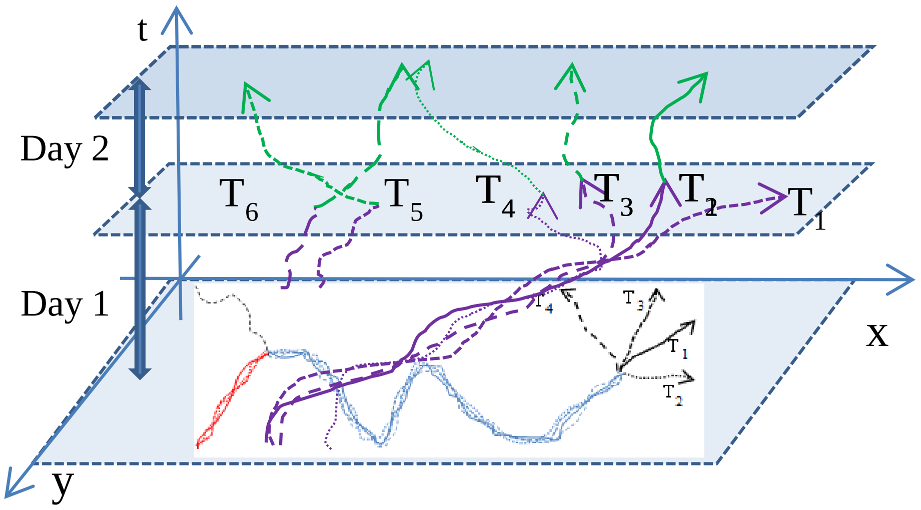


Figure 4.1: Six trajectories, spanning in 2 days, split into daily chunks.

to $< T_1, T_2, T_3, T_4 >$ *and* $< T_5, T_6 >$*, respectively. Although not illustrated in the figure, the first sub-chunk is valid during [20:00, 0:00) of Day 1 while the second sub-chunk is valid during [22:00, 0:00) of Day 1, thus they are overlapping in time. Especially for the first sub-chunk, we also illustrate the projection of the four trajectories on the spatial domain, which corresponds to Figure 3.1(b).*

Next, the subtrajectories of each sub-chunk are clustered on the spatiotemporal domain with a sampling-based algorithm. In the previous example, this step results in the formation of two clusters of subtrajectories (in red and blue) and five outlier subtrajectories (in black), see Figure 3.1(b). Thus, *ReTraTree* maintains only the representatives at the 3rd level of the structure, while the actual clustered data are archived at the 4th level.

Figure 4.2 (and the paragraphs that follow) present *ReTraTree* in detail. Note that the top-three levels of the *ReTraTree* reside in main memory and only the 4th level is disk-resident.

1st level (chunks). The root of the *ReTraTree* consists of p entries, $p \geq 1$, corresponding to chunks sorted by time (in the example of Figure 4.1, at daily level). Note that for each chunk H_i , $i = 1, \ldots, p$, there is no need to maintain the actual temporal periods in the index nodes since they correspond to fixed equal-length splitting intervals. Each entry H_i maintains only a pointer to the respective set of sub-chunks $H_{i,n}$, $n \geq 1$, under this chunk. The set of all

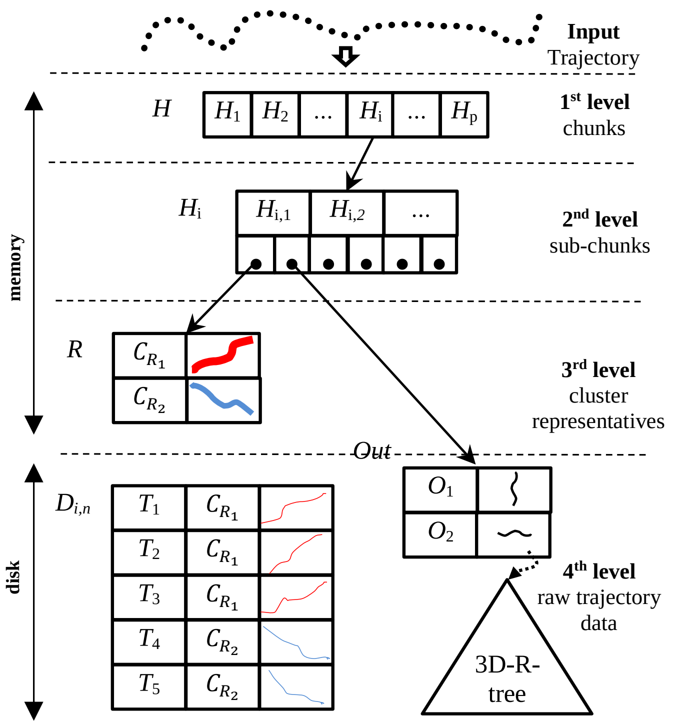


Figure 4.2: Overview of the *ReTraTree* indexing scheme.

chunks forms the 1st level of the structure.

2nd level (sub-chunks). For each chunk, there is a set of sub-chunks, actually a sequence of triples $< H_{i,n}.per, H_{i,n}.R, H_{i,n}.Out >$, $n \geq 1$, where per is a temporal period $[pert.s, pert.e)$ when the sub-chunk is valid (in the example of Figure 4.1, [20:00, 0:00) and [22:00, 0:00), respectively for the two sub-chunks of Day 1), while R (Out) are pointers to the set of representative (outlier, respectively) subtrajectories belonging to sub-chunk $H_{i,n}$. The sequence of triplets is ordered by $< pert.s, pert.e >$. The set of all sets of sub-chunks forms the 2nd level of the structure.

3rd level (cluster representatives). For each sub-chunk, the entries of set R consist of pairs $< R_j, C_{R_j} >$, $j \geq 0$, where each entry includes the representative subtrajectory R_j and a pointer C_{R_j} to the subset of subtrajectories belonging to that sub-chunk and forming a cluster around R_j. Note that $j = 0$ implies that there may exist sub-chunks with zero clusters (i.e. including outliers only). The set of all sets of cluster representatives (along with the pointers to actual data) forms the 3rd level of the structure.

4th level (raw trajectory data and outliers). The sets of actual subtrajectories that compose clusters C_{R_j} are stored at the 4th level of the structure. For each sub-chunk $H_{i,n}$, there corresponds a set $D_{i,n}$ consisting of triples <subtrajectory-id, C_{R_j}, subtrajectory-3D-polyline> that keep the information about which subtrajectory belongs to which cluster. On the other hand, set Out contains the outlier subtrajectories of that sub-chunk. The outlier subtrajectories are appropriately indexed in a 3D-R-tree structure [94]. The clustering process of subtrajectories belonging to a sub-chunk, during which we detect sets S and Out, is a key process for *ReTraTree* and is described in detail in Section 4.3.3.

How *ReTraTree* handles a new trajectory is discussed in the subsections that follow.

4.3.2 Hierarchical Temporal Partitioning

Given a trajectory database D of lifespan l (whose duration is denoted as $|l|$), a new trajectory T, and a fixed partitioning granularity p, applicable at the *ReTraTree* 1st level, T is partitioned into a number of subtrajectories S_i, $i \geq 1$, where the subtrajectory S_i is the restriction of T inside a temporal period p_i,

$$p_i = [\tfrac{|l| \cdot (i-1)}{p}, \tfrac{|l| \cdot (i)}{p})\ ,\ 1 \leq i \leq p$$

where $|l|/p$ is the length of each time interval (i.e. the duration of the lifespan of each chunk) and timestamps $D_{t.s} + |l| * (i-1)/p$, $2 \leq i \leq p$ are called *splitting timestamps*. As such, every trajectory in the dataset is partitioned into subtrajectories using the same (pre-defined, according to granularity p) splitting timestamps. This chunking process is applied incrementally, whenever a batch of new recordings from a moving object arrives. In case of a new trajectory with temporal information that exceeds the last existing chunk, a new chunk is created and the set of chunks CK is extended.

At the 2nd level, each chunk is subdivided into (possibly, overlapping) sub-chunks. Specifically, a chunk is split into sub-chunks by grouping the subtrajectories contained in the chunk, according to the following definition.

Definition 4.6. *(Grouping of subtrajectories in the same sub-chunk): Given a temporal tolerance parameter τ and two subtrajectories $S \in T$ and $S' \in T'$ belonging to the same chunk, these subtrajectories can be grouped together in* *the same sub-chunk if their starting (ending) timepoints differ at most $\tau/2$, respectively. Formally, it should hold that:*

$$|S_{t.s} - S'_{t.s}| \leq \tau/2 \land |S_{t.e} - S'_{t.e}| \leq \tau/2 \tag{4.7}$$

Note that the above definition is not deterministic as there might be a subtrajectory $S'' \in T''$ that also satisfies this condition. We handle this case by grouping the subtrajectories when this condition is satisfied for the first time. Thus, we do not define and we do not search for a kind of "best-matching" sub-chunk. The reasons for this choice is that we are in favor of a very efficient insertion process, while we do not care about an optimal matching as this issue will be handled when the analyst asks for a clustering analysis. Regarding tolerance parameter τ, it is a user-defined parameter and can be exploited to impose an either stricter or looser notion of grouping. It also implies that e.g., when τ is set to 10 minutes, a subtrajectory of less than 20 minutes duration cannot be grouped together with a subtrajectory of more than 30 minutes duration.

4.3.3 Sampling-based Subtrajectory Clustering

As already mentioned in Section 4.2, maximizing Equation 4.6 is a hard problem. In order to tackle it, we adopt a methodology for the optimal segmentation and selection of a sample of subtrajectories from a trajectory dataset. Thus, in Algorithm 4.1, we outline the *<u>S</u>ampling-based <u>S</u>ub-<u>T</u>rajectory <u>C</u>lustering* (S^{2}T-Clustering) algorithm, a two-step process that relies on a subtrajectory sampling method, proposed in [62]. Briefly, S^{2}T-Clustering relies on the output of the aforementioned sampling method (1st step), which is a set of subtrajectories in the MOD that can be considered as representatives of the entire dataset. These samples serve as the seeds of the clusters, around which clusters are formed based on a greedy clustering algorithm (2nd step).

Algorithm 4.1 S^{2}T-Clustering

```
1: Input: MOD $D = \{T_1, T_2, \ldots, T_N\}$, $\epsilon$, $\delta$
2: Output: Sampling set $R$, Clustering $C$, Outlier set $Out$.
3: $(R, S) \leftarrow Sampling(D, \epsilon)$
4: $(C, Out) \leftarrow GreedyClustering(R, S, \delta)$
5: return $(R, C, Out)$
```

The first step of S^{2}T-Clustering algorithm (line 3) invokes the Sampling method, which aims to solve an optimization problem, namely to maximize the number of subtrajectories represented in a sampling set. In a few words, Sampling calculates the voting descriptor V_T^D of all trajectories T in D with respect to D, as described in Definition 4.3. Then, based on this signal, each trajectory is partitioned into subtrajectories having homogeneous representativeness (i.e. the representativeness of all segments in a subtrajectory does not deviate over a user-defined threshold), irrespectively of their shape complexity. According to [62], a trajectory should have at least w points in order for the segmentation to take place. Thus, w is an application-based parameter of Sampling that acts as a lower bound of the length of a trajectory under segmentation. Subsequently, Sampling selects a sampling set $R = \{R_1, \dots, R_M\}$ of subtrajectories, which are hereafter considered as the representatives of D. Note that the number M of subtrajectories is not user-defined; instead, it is dynamically calculated by the method itself. This is achieved by tuning Sampling with a parameter ϵ ($\epsilon > 0$ and $\epsilon \to 0$), the role of which is to terminate the internal iterative optimization process when the optimization formula is lower than a given threshold (i.e. the ϵ parameter). Back to the example of Figure 3.1, the above voting-and-segmentation phase would result in segmenting trajectory T_1 into three subtrajectories (coloured red, blue, and black, respectively, in Figure 3.1) according to its representativeness; similar for the rest trajectories of the MOD.) Then, Sampling would intuitively select two subtrajectories as representatives, one from the blue subtrajectories, and one from the red subtrajectories.

At its second step (line 4), S^{2}T-Clustering uses sampling set R in order to cluster the subtrajectories of the dataset according to the following idea: each subtrajectory in the sampling set is considered to be a cluster representative. More specifically, clustering is performed by taking into account sampling set $R = \{R_1, \dots, R_M\}$ and vector of votes (i.e. representativeness) $V_{S_i}^{R_j}$ (actually we use the average voting $avg(V_{S_i}^{R_j})$) between subtrajectories of the original MOD $S_i \in D - R$ with respect to the representative subtrajectories $R_j \in R$. Recall that $V_{S_i}^{R_j}$ (Definition 4.2) consists of $|S_i|$ elements, where each one represents the voting that the segments of S_i receive from the segments of R_j. To this end, in order for the S^{2}T-Clustering algorithm to maximize Equation 4.7 for the special case where the time window W corresponds to the lifespan l of D, the cluster C_{R_j} of a representative subtrajectory of the sampling dataset $R_j \in R$, i.e. the set of subtrajectories that are assigned to cluster C_{R_j}, is provided by:

$$\begin{aligned} C_{R_j} = \{S_i \in D - R : avg(V_{S_i}^{R_j}) \geq avg(V_{S_i}^{R_v}), \\ \forall R_v \in R \wedge avg(V_{S_i}^{R_j}) \geq \delta\} \end{aligned} \tag{4.8}$$

On the other hand, set *Out* of outliers consists of subtrajectories that have been assigned to no cluster:

$$Out = \{S_i \in D - R - C_{R_j}, \forall R_j \in R\} \tag{4.9}$$

The algorithm outlined in Algorithm 4.1 simply iterates through all the representative subtrajectories $R_j \in R$ of the sampling dataset R and applies the constraints of Equation 4.8. Parameter δ is a positive real number between 0 and 1 that acts as a lower bound threshold of similarity between subtrajectories and representatives. As such, it controls the size of the clusters C and the outlier set *Out*.

4.3.4 ReTraTree Maintenance

S^{2}T-Clustering does not support arbitrary time windows nor dynamic data. The additional challenge that we have to address is to efficiently support such a clustering for arbitrary time windows and dynamic data. To achieve this, we need to efficiently support insertions of new trajectories in the *ReTraTree*.

The incremental maintenance of the *ReTraTree*, whenever a batch of recordings of a moving object (i.e. a trajectory T) arrives, is supported by the *ReTraTree-Insert* algorithm outlined in Algorithm 4.2. We have already described how our method incrementally performs the first phase of partitioning in the time dimension (line 2). The *update_chunks* function returns the set of chunks H and the respective set of subtrajectories S that correspond to the input trajectory T, i.e. the subtrajectories S_i that intersect temporally with chunk H_i. Then, the algorithm assigns each subtrajectory S_i to an appropriate sub-chunk (lines 3-5). This is actually checked by the *find_subchunk* function which, instead of applying Definition 4.6 between S_i and the other subtrajectories in the sub-chunk, simply tests whether the following inequality holds: $|S_{i,t.s} - H_{i,n,t.s}| \leq \tau/2 \wedge |S_{i,t.e} - H_{i,n,t.e}| \leq \tau/2$. To gain this efficiency, the implicit assumption is that the temporal borders of each sub-chunk are left unchanged since its initialization with its first subtrajectory. If there is not a matching sub-chunk with respect to time (line 6), a new sub-chunk is created, which is initialized with an empty representative set R, and an outliers set Out including the unmatched subtrajectory (line 18). If there is an appropriate sub-chunk for the subtrajectory under processing (line 6), the algorithm tries to greedily assign it to the best existing cluster (lines 7- 14). If this attempt fails (line 15), the algorithm invokes *ReTraTree-Handle-Outlier* algorithm (outlined in Algorithm 4.2).

Algorithm 4.2 ReTraTree-Insert

```
1: **Input:** *ReTraTree* root, trajectory $T$, $\tau$, $\epsilon$, $\delta$
2: $(H, S) \leftarrow update\_chunks(root, T)$
3: **for** each pair $(H_i, S_i) \in (H, S)$ **do**
4:    clustered = false
5:    $H_{i,n} = find\_subchunk(S_i, H_i)$
6:    **if** $H_{i,n} \neq \emptyset$ **then**
7:       $max\_v_i = -1$
8:       **for** each $R_j \in H_{i,n.R}$ **do**
9:          **if** $(non\_common\_lifespan(S_i, R_j) < \tau)$ **then**
10:            $v = avg(V_{S_i}^{R_j})$
11:         **if** $(v \leq \delta \wedge v > max\_v_i)$ **then**
12:            assign $S_i$ to $C_{R_j}$
13:            $max\_v_i = v$
14:            clustered = true
15:         **if** (clustered = false) **then**
16:            $ReTraTree - Handle - Outlier(root, H_{i,n}, S_i, \epsilon, \delta)$
17:    **else**
18:       $update\_chunk(S_i, H_i)$
19: **return**
```

In particular, Algorithm 4.3 adds the subtrajectory into the outliers' set of the sub-chunk, which acts as a temporary relation upon which S^{2}T-Clustering is applied, whenever the size of the relation exceeds a threshold α (e.g., α Mb that may correspond to a percentage of the dataset) with respect to its size, at the time of the previous invocation of the algorithm (line 3). Then, a new set of representative subtrajectories will extend the existing set of representatives, only if it is δ-different from them (line 5). For each of the resulting new outliers, we re-insert the subtrajectory from the top of the *ReTraTree* structure. This implies that we recursively apply *ReTraTree*-Insert for that subtrajectory in order to search for other sub-chunks wherein it could be clustered or to form a new sub-chunk. This recursion is continued until an outlier is either clustered or partitioned to smaller pieces, due to successive applications of S^{2}T-Clustering. In case the size of an outlier becomes smaller than w, we archive it in the relation containing the raw data. Before applying a clustering analysis task and if the tree has been updated since the insertion of this specific trajectory, we give a last chance to these small outliers to be clustered by re-dropping them from the top of the structure. In other words, for a (sub-)trajectory T_k, if its length $|T_k| < w$ and T_k has not been assigned to a cluster, then, since it cannot be further segmented (and thus become again candidate to be clustered in a different sub-chunk); it cannot also be clustered before new trajectories update the tree.

Algorithm 4.3 ReTraTree-Handle-Outlier

```
1: Input: ReTraTree root, sub-chunk $H_{i,n}$, outlier $S_i$, $\tau$, $\epsilon$, $\delta$
2: $H_{i,n}.Out \leftarrow H_{i,n}.Out \cup S_i$
3: if $|H_{i,n}.Out| > \alpha$ then
4:     $(R, C, Out) \leftarrow S^2T - Clustering(H_{i,n}.Out, \epsilon, \delta)$
5:     $H_{i,n}.R \leftarrow H_{i,n}.R \cup \{R' \subset R \mid \text{NOT } \delta - join(H_{i,n}.R, R)\}$
6:     for each outlier $O$ in $Out$ do
7:         if $|O| < w$ then
8:             archive $O$
9:         else
10:            $ReTraTree - insert(root, O, \tau, \epsilon, \delta)$
11: return
```

4.4 *ReTraTree* in Action

ReTraTree maintains clustered subtrajectories at its leaves. However, given a temporal period, it is not enough to retrieve the clusters (i.e. the subtrajectories "following" the representatives) that overlap this period. The reason is that the subtrajectory clustering of overlapping sub-chunks may form representatives that: (a) are almost identical (as such, a 'merge' operation should take place in order to report only one cluster as the union of the two (or more) clusters built around the similar representatives), and/or (b) can be continued by others (as such, an 'append' operation should take place to identify the longest clusters, i.e. representatives).

In other words, an algorithm is required that takes *ReTraTree* as input and searches within it in order to identify the longest patterns with respect to the user requirements (e.g., discover all valid clusters during a specific period of time). This is made feasible through appropriate 'merge' and 'append' operations applied to the query results. To the best of our knowledge, such a query-based clustering approach is novel in the mobility data management and mining literature.

4.4.1 QuT-Clustering

Given two representatives R_i and R_j if (a) the two representatives have the same lifespan with respect to threshold τ and (b) the two representatives are also similar with respect to similarity threshold δ (this means that they origin from different sub-chunks), then this implies a *'merge'* operation. On the other hand, if (a) R_i ends close to the timepoint when the R_j starts with respect to threshold t, (b) the Euclidean distance of the last point of R_i is close (with respect to a distance threshold d) to the first point of R_j, and (c) a sufficient number of the same moving objects are represented by both representatives (with respect to a percentage threshold γ), this implies an *'append'* operation. Figure 4.3 illustrates representatives of a chunk consisting of two chunks. A *'merge'* operation occurs between R_1 and R_2, whereas R_5 and R_6 will both be maintained in the final outcome although they have similar lifespans. An *'append'* operation occurs between R_3 and R_4.

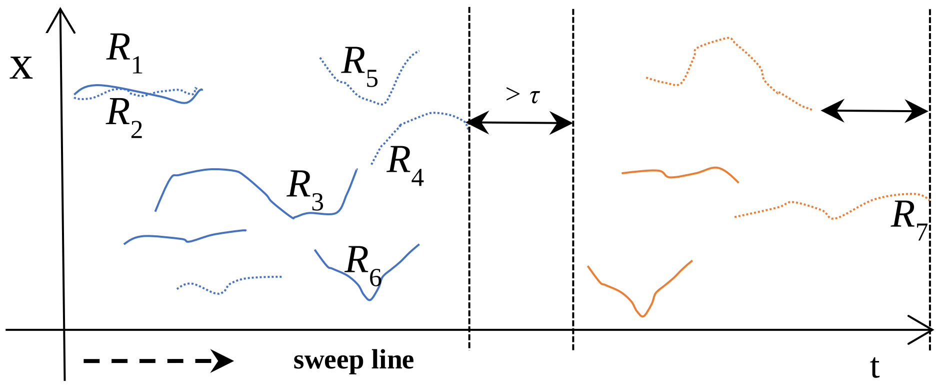


Figure 4.3: Representatives of a chunk with two sub-chunks (dashed vs. continuous polylines) organized in a temporal priority queue of two groups (blue vs. red polylines).

Algorithm QuT-Clustering provided in Algorithm 4.4 proposes such a solution on top of *ReTraTree*. The user gives as parameters the period of interest W, and the algorithm traverses the tree and returns clusters valid in this period.

More specifically, the algorithm initially finds the chunks and then the sub-chunks that overlap the given period (lines 3-5). These sub-chunks are organized in a priority queue (line 6), which orders groups of representatives

Algorithm 4.4 QuT-Clustering

```
1: Input: $ReTraTree$ root, temporal period $W$
2: Output: Clusters $C$ valid inside $W$
3: $H \leftarrow \{H_i, \mid overlap(root.H_i, W)\}$
4: for $H_i \in H$ do
5:     $H_{i,n} \leftarrow \{H_{i,n}, \mid overlap(W, H_{i,n}.per)\}$
6:     $TEQ\_PQ \leftarrow bulk\_push\_TEQ(TEQ\_PQ, H_{i,n}, \tau)$
7:     while $TEQ\_PQ \neq \emptyset$ do
8:         $R \leftarrow temporal\_interleaving(R \cup TEQ\_PQ.pop())$
9:         for $R_j \in R$ do
10:            $R_{overlap} \leftarrow temporal\_overlap(R_j, R, t)$
11:            for $R_k \in R_{overlap}$ do
12:                if $(non\_common\_lifespan(R_j, R_k) < \tau)$ then
13:                    if $(avg(V_{R_j}^{R_k}) \geq \delta)$ then
14:                        $merge(R_j, R_k)$
15:                else if $(|R_{j,t.e} - R_{k,t.s}| < t)$ then
16:                    if $(euclidean\_dist(p(R_{j,t.e}), p(R_{k,t.s})) < d)$ AND
                       $(common\_IDs(R_j, R_k) > \gamma)$ then
17:                        $append(R_j, R_k)$
18:                else
19:                    continue
20:        $R_{clustered} \leftarrow \{R_j \in R, \mid |R_{j,t.e} - H_{i,t.e}| > \tau\}$
21:        $R \leftarrow R - R_{clustered}$
22:        $C \leftarrow C \cup R_{clustered}$
23: return $C$
```

of the sub-chunks. Each group contains temporally successive representatives that are at most in temporal distance τ from each other. To exemplify this ordered grouping of sub-chunks, Figure 4.3 shows the representative subtrajectories (excluding outliers) of a single chunk, which consists of two sub-chunks, distinguished as dashed vs. continuous polylines. Note that for simplicity, y-dimension is omitted and specific borders of sub-chunks are not depicted, while the representatives form two groups, colored blue and red, respectively. Subsequently, the algorithm pops each group one-by-one and sorts all representatives with respect to time dimension, by interleaving the already sorted (from the step that constructs the priority queue) representatives coming from different sub-chunks (line 8). This is done by including representatives left from a previous round of the algorithm. Then, the algorithm sweeps the temporally interleaved representatives along the time dimension (line 9) and, for each of them, identifies the subset of its subsequent representatives in time that their lifespan overlap with the lifespan of the currently investigated representative, after the extension of the latter towards the future by t timepoints. For each pair of representatives R_j and R_k, the algorithm checks whether a merge operation (lines 12-14) or an append operation (lines 15-17) is necessary. In any other case (line 19) the algorithm simply continues with the next representative, and maintains both representatives intact. After each sweep, the algorithm maintains in the next round only those representatives that end at most τ seconds before the border of the current chunk (e.g., R_7, in Figure 4.3), as candidates for merging with subsequent representatives (lines 20-22). The rest of the representatives are part of the final outcome of the algorithm.

Regarding the technical details, a *'merge'* operation practically maintains (in the working set of representatives R) one of the two representatives (e.g., the first) in the remaining process. The other representative is appropriately flagged so as to be able to retrieve the raw data that correspond to this cluster, if needed. For the *'append'* operation, we need to retrieve the identifiers of the subtrajectories (not the subtrajectories themselves) that correspond to the clusters implied by the representatives and apply a set intersection operation. This is facilitated by traditional indexing structures, such as by indexing the pair of representative id (i.e. cluster identifier) and subtrajectory id of the raw data relation at the 4th level of *ReTraTree*. Practically, an *'append'* procedure replaces from the working set of representatives S the two representatives with one of those subtrajectories that exist in both clusters. Note that the chosen subtrajectory is selected randomly and it is the one used in the remaining process. Using another non-random choice at this step would be possible but not desired, as it would imply retrieval of the actual subtrajectories. Finally, note that for simplicity reasons, we use the same threshold τ to compute the equivalence classes, as well as for considering whether two representatives refer to the same temporal period. In practice, these two easily configured parameters may be different, depending on the analysis scenarios pursued by the user. Similarly, threshold t corresponds to a small duration value, for instance, $t = 0$ in order to be as strict as possible.

4.4.2 Architectural Aspects

The architecture of our framework is illustrated in Figure 4.4. The core of the framework is the *ReTraTree* structure that is fed by either new incoming trajectories or data that have been processed in a previous round and could not be clustered. In both cases the *ReTraTree-Insert* algorithm handles the insertion. The trajectories are partitioned according to the in-memory part of the structure and stored on disk-based partitions. The trajectories assigned to an existing representative trajectory) are archived on disk in clustered partitions. Instead, trajectories that were not clustered are organized on disk in an (intermediate) outlier partition. When the size of the partitions exceeds a threshold, the *S*^{2}*T-Clustering* algorithm applies the *Voting* process upon which the *Segmentation* of the trajectories takes place. The resulting subtrajectories and their voting descriptors form the input of the *Sampling* module that selects new (i.e. non-existing) representatives that are back-propagated to the in-memory part of the *ReTraTree*. The new representative trajectories and the raw subtrajectories form the input of the *GreedyClustering* module. If a subtrajectory is clustered around a new representative, it is archived on disk. Otherwise it is an outlier and is re-inserted to *ReTraTree*, as it may now be accommodated in the index. This is due to its segmentation during the operation of *S*^{2}*T-Clustering*, or due to the creation of new matching sub-chunks or representatives in the index. Finally, the analyst uses the *QuT-Clustering* algorithm to perform interactive clustering analysis by providing different time windows W as input.

4.4.3 Complexity Analysis

Concluding the discussion about our proposal, we provide a complexity analysis of (i) loading the *ReTraTree* structure and (ii) performing *QuT-Clustering*, according to the algorithms proposed so far. The assumption we make throughout our analysis is that the distribution of trajectories during the dataset's lifetime is uniform; in other words, selecting two random timepoints, t_i and t_j, the number of trajectories being 'alive' at t_i and t_j, respectively, remains more or less the same. In real world datasets, we do not expect to find perfect compliance to this, but we believe that this is a realistic assumption.

Lemma 4.1. *Under the uniformity assumption, the loading cost of the* ReTraTree *is:*

$$O(p \cdot (\overline{T_k} \cdot N + \overline{H} \cdot \overline{R}^2) \cdot log(\overline{T_k} \cdot N / \overline{H} \cdot \overline{R}))$$

where $\overline{H}$ is the average number of sub-chunks per chunk, $\overline{R}$ is the average number of representative subtrajectories per sub-chunk and ($\overline{T_k}$ denotes the average number of trajectory points in a database consisting of N trajectories.

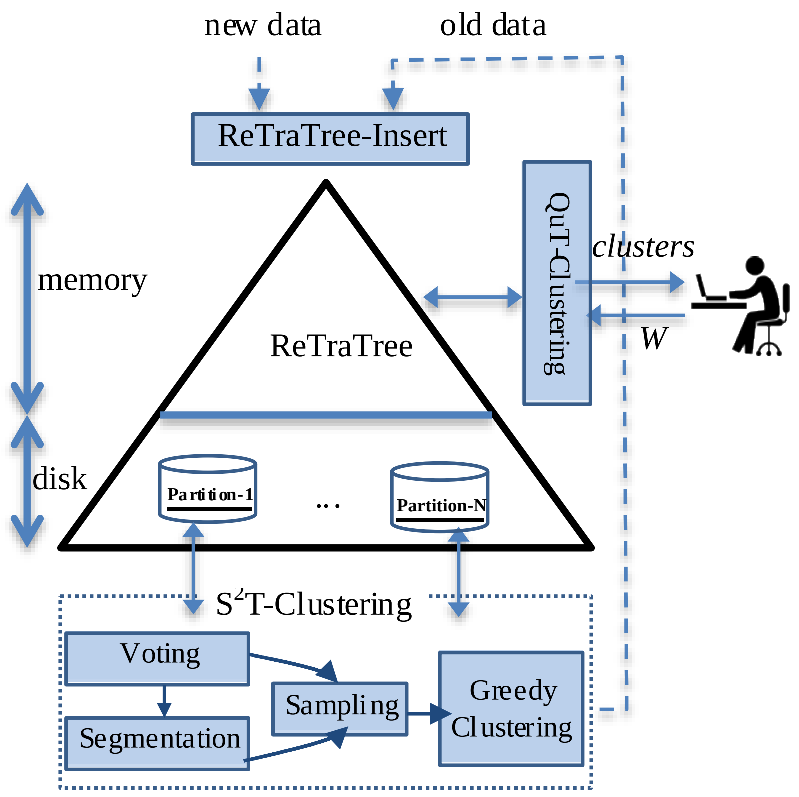


Figure 4.4: Architectural aspects of ReTraTree.

Proof. Considering that $\overline{H}$ is the average number of sub-chunks per chunk and $\overline{R}$ denotes the average number of representative subtrajectories per sub-chunk, *ReTraTree* can be considered as p balanced trees of $h = 2$ (excluding the root and the 3D-Rtrees found at the 1st and the 4th level of the structure, respectively) with the upper bound for the maximum number of leaves per tree being upper bounded by $\overline{H} \cdot \overline{R}$. Given the above, each sub-chunk has an average size of $\overline{T_k} \cdot N/\overline{H} \cdot \overline{R}$ segments. Setting threshold α of each sub-chunk to this value, *ReTraTree* will invoke the S^2 *T-Clustering* algorithm $O(p \cdot \overline{H} \cdot \overline{R})$ times [**result 1**].

Regarding the cost of S^2 *T-Clustering* algorithm, it is composed by the costs of its two components, namely *Sampling* and *Greedy-Clustering* (see Algorithm 4.1). As it has been shown in [62], the most computationally intensive part of the *Sampling* method is the voting process with $O(\overline{T_k} \cdot N \cdot log(\overline{T_k} \cdot N))$ cost for each trajectory in a database consisting of N trajectories indexed by a 3D-Rtree structure. Note that in our case, we maintain a forest of such trees, where each of them corresponds to the segments of the subtrajectories that belong to the dynamically changing set of outliers of a sub-chunk. Therefore, in our case the number N of trajectories corresponds to the number of subtrajectories that have been assigned to this set. Regarding *Greedy-Clustering*, as the voting vectors are pre-calculated during the *Sampling* step, its cost is dominated by the size of the representatives set $\overline{R}$. More specifically, the cost is $O(\overline{R} \cdot log(\overline{T_k} \cdot N))$, i.e. the cost of performing $\overline{R}$ trajectory-based range queries in the database [**result 2**].

Since the size of the outliers of a sub-chunk set is estimated to be $\overline{T_k} \cdot N/\overline{H} \cdot \overline{R}$, the cost of the S^2 *T-Clustering* algorithm in a sub-chunk is: $O(\overline{T_k} \cdot N/\overline{H} \cdot \overline{R} \cdot log(\overline{T_k} \cdot N/\overline{H} \cdot \overline{R}) + \overline{R} \cdot log(\overline{T_k} \cdot N/\overline{H} \cdot \overline{R}))$ [**result 3**].

By combining results 1-3 above (i.e. multiply the $p \cdot \overline{H} \cdot \overline{R}$ number of leaves with the cost of the S^2 *T-Clustering* algorithm of a single sub-chunk), we have proven Lemma 4.1.

□

From a different point of view, the cost per trajectory insertion can be split in four parts: (i) the cost of chunking and sub-chunking the original trajectory to subtrajectories, (ii) for each subtrajectory, the cost of finding the matching representative, (iii) the cost of invoking the S^2 *T-Clustering* algorithm, which is only in case that the subtrajectory overflows threshold α of the sub-chunk, and (iv) the cost of checking whether the new representatives extracted by S^2 *T-Clustering* can be inserted into the already identified representatives. Regarding the cost of each part, that of (i) is trivial, while that of (ii) and (iv) is $O(R)$ in both cases, since it implies a scan on the set of representatives, which however is small ($R \ll N$). Obviously, the cost per trajectory insertion is dominated by the S^2 *T-Clustering* algorithm.

Lemma 4.2. *Under the uniformity assumption, the cost of the* QuT-Clustering *algorithm is:*

$$O((\overline{H} \cdot \overline{R})^2)$$

where $\overline{H}$ is the average number of sub-chunks per chunk and $\overline{R}$ is the average number of representative subtrajectories per sub-chunk.

Proof. As already shown, under uniformity assumption, each chunk maintains $O(\overline{H} \cdot \overline{R})$ representatives. Thus, invoking *QuT-Clustering* will eventually scan a number of $O(\lceil (|W|)/p \rceil \cdot \overline{H} \cdot \overline{R})$ representatives, where $\lceil (|W|)/p \rceil$ is the number of the involved chunks. However, at any time, the algorithm maintains a priority queue of $O(\overline{H} \cdot \overline{R})$ representatives (worst case scenario). Note that sorting this priority queue costs $O(\overline{H} \cdot \overline{R})$ only, since the sets of representatives of the corresponding sub-chunks are already sorted, thus a merge-sort performs the required temporal interleaving. Given this, as the representatives reside in memory and there is no special organization at this level, for each of these representatives the algorithm will scan all the other representatives in the worst case, thus leading to $O((\overline{H} \cdot \overline{R})^2)$.

□

Interestingly, the cost of the *QuT-Clustering* algorithm is independent to the size of the database, thus it is a highly efficient solution for *progressive temporally-constrained subtrajectory clustering analysis*. This is validated in the experimental study that follows.

4.5 Experimental Study

In this section, we present our experimental study. *ReTraTree* and its algorithms were implemented in-DBMS in Hermes [1] MOD engine over PostgreSQL, by using the GiST extensibility interface provided by PostgreSQL. More specifically, the top three levels of *ReTraTree* that reside in memory were implemented as temporary tables, while the 4th level was stored in traditional tables, upon which the 3D-Rtrees were built. Although our proposal is generic, we chose to put extra effort to implement it on a real-world MOD management system rather than an ad-hoc implementation, because of the initially placed goal to support progressive clustering analysis. We argue that this is an important step towards bridging the MOD management and mobility mining domains, as state-of-the-art frameworks [35] could make use of the efficiency and the advantage of our proposal to execute clustering analysis tasks via simple SQL. This way, our approach becomes practical and useful in real-world application scenarios, where concurrency and recovery issues are taken into consideration. All the experiments were conducted on an Intel Xeon X5675 Processor 3.06GHz with 48GB Memory running on Debian Release 7.0 (wheezy) 64-bit. We used PostgreSQL 9.4 Server with the default configuration for the memory parameters (shared_buffers, temp_buffers, work_mem, etc.). The outline of our experimental study is as follows: in Section 4.5.1, we discuss the setting of various parameters. In Section 4.5.2 we present baseline solutions with which we compare our proposals. In Section 5.3 we describe the datasets that we used in this study. In Section 4.5.4, we apply a qualitative analysis to verify that our proposal operates as expected by using datasets with ground truth. In Section 4.5.5 we provide a sensitivity analysis with respect to various parameters. In Section 4.5.6 we continue the qualitative evaluation of our approach in real datasets with general-purpose clustering validation metrics. In Section 4.5.7, we evaluate the maintenance of *ReTraTree* in terms of loading performance and size. In Section 4.5.8, we measure the I/O performance of *ReTraTree* with respect to the *QuT-Clustering* algorithm, the performance of which is assessed in Section 4.5.9.

4.5.1 Parameter Settings

Regarding parameter settings, as our approach makes use of the sampling methodology of [62], we followed the best practices presented in that work. More specifically, the value of parameter σ was set to 0.1% of the dataset diameter, while that of ϵ was set to 10^{-3}. We would like to note that we made several experiments by modifying the values of these parameters and the differences in the results were negligible, thus in a way we re-validated our earlier experience in the current setting.

As far as it concerns the parameters that affect the construction of *ReTraTree*, their effect is rather straightforward. Here we report our findings, which have been experimentally validated. More specifically, the more we increase p, the more chunks we create and hence the more the partitions (i.e. relations in our implementation). As the number of these partitions increases, the size and the construction time of *ReTraTree* decreases as the structure holds the same amount of data, but in smaller relations (i.e. smaller indexes). Moreover, as by increasing p we have a smaller structure size, the runtime of *QuT-Clustering* will be smaller. Regarding the τ parameter, the smaller it is, the more the number of sub-chunks and hence the more relations; thus, we fall at the previous case. In addition, the smaller the similarity threshold δ, the more the subtrajectories that are assigned to already existing clusters. This implies that fewer subtrajectories will end up to the outliers' set and hence the S^2T*-Clustering* algorithm runs fewer times. This means that the lower the δ the lower the construction time of the *ReTraTree*. Finally, regarding the value of α that is the threshold of the size of the outliers' set above which the *S*^{2}*T-Clustering* algorithm is applied, the more we increase α, the fewer times the *S*^{2}*T-Clustering* will run and consequently the smaller the construction time of *ReTraTree*. In our experiments we fixed threshold α to 5% of the dataset size.

In the subsequent sections we report on the effect of the important parameter of the time window W, while in Section 4.5.5 we particularly study the effect on both the efficiency and the quality of *QuT-Clustering* when varying the values of the remaining parameters, whose effect is not trivial to foresee without experimentation.

4.5.2 Baseline Solution

To the best of our knowledge the *ReTraTree* structure and the corresponding *QuT-Clustering* algorithm is a novel solution to the temporally-constrained subtrajectory cluster analysis problem and there is no comparable technique. Furthermore, as already mentioned, the *S*^{2}*T-Clustering* algorithm has some unique characteristics that make it appropriate as part of our solution. The most important characteristic is that it provides a greedy solution to the problem for the degenerated case where the time window W is equal to the entire lifespan of the dataset. This is a key observation that we exploit in our approach by organizing our data in sub-chunks consisting of subtrajectories having the same lifespan and applying *S*^{2}*T-Clustering* to them. In Section 4.5.4 we demonstrate that the state-of-the-art TRACLUS algorithm [49] that is utilized also by the TCMM framework [53] cannot identify the clusters in datasets including ground truth. Moreover, in [62] it is shown that an efficient solution for the sampling process that the *S*^{2}*T-Clustering* algorithm utilizes, it requires a 3D-Rtree index.

Given the above, in this empirical study we set the following comparable pairs: (i) we compare the *ReTraTree* structure with the 3D-Rtree structure. A secondary but important reason for this choice is that 3D-Rtree is the prevailing structure that state-of-the-art spatial DBMS vendors have chosen to support in their products (e.g., PostGIS, Oracle Spatial); (ii) we compare the *S*^{2}*T-Clustering* algorithm with *QuT-Clustering* algorithm for the degenerated case where the time window W is equal to the entire lifespan of the dataset. Of course, our approach is applicable in any user-defined time window W. Thus, in this case the comparable pair is on the one hand the *QuT-Clustering* and on the other hand again the *S*^{2}*T-Clustering* algorithm after having restricted the dataset to the selected time window W. This implies that an analyst should first apply a temporal range query to restrict the dataset inside W, then build a 3D-Rtree on the restricted dataset and afterwards run the S^2T*-Clustering* algorithm. This is the best choice to perform a progressive clustering analysis without the *ReTraTree* and this is how the analysts work currently.

4.5.3 Datasets

In this study we used two real datasets, $IMIS_2$ and GeoLife, and one synthetic, called SMOD, that were presented in Section 1.5. Table 4.2 presents their statistics.

Table 4.2: Dataset Statistics

Statistic	**SMOD**	**GeoLife**	**$IMIS_2$**
# Trajectories	400	18668	5110
# Segments	35273	24159325	443657
Dataset Duration (hh:mm:ss)	0:02:00	1932 days 22:59:48	6 days 19:59:53
Avg. Sampling Rate (hh:mm:ss)	0:00:01	0:00:08	0:18:02
Avg. Segment Length (m)	8	72	1545
Avg. Segment Speed (m/s)	7.83	5.01	7.03
Avg. Trajectory Speed (m/s)	2.86	3.91	4.52
Avg. # Points per Trajectory	89	1295	88
Avg. Trajectory Duration (hh:mm:ss)	0:01:28	2:43:15	11:33:45
Avg. Trajectory Length (m)	691	93046	134,148

4.5.4 Quality of Clustering Analysis in Synthetic Datasets Including Ground Truth

To the best of our knowledge there is no real trajectory dataset that provides ground truth that can be utilized for validating clustering techniques. Thus, our premise is to evaluate our approach qualitatively by using a synthetic dataset. The description of SMOD implies that the possible ending times of a moving object are $t \approx 20$, $t \approx 50$, $t \approx 80$ or $t \approx 100$. Based on this fact and by setting the chunk size equal to the duration of the dataset (i.e. 100 sec) we infer that the *ReTraTree* construction process should create 4 sub-chunks. We also infer the lifespan l of each sub-chunk. The invocation of the *ReTraTree-Insert* that builds these sub-chunks, concludes to apply the *S^2T-Clustering* algorithm in each of these sub-chunks, which in its turn results in discovering representatives (i.e. clusters) in each of them. This ground truth is illustrated in Table 4.3.

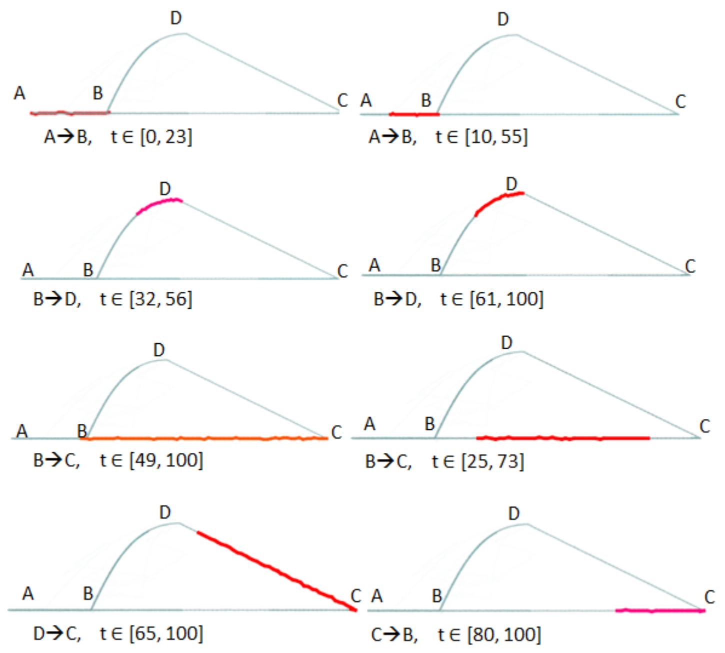
Table 4.3: The ground truth hidden in SMOD

Sub-chunk	Path	Time periods (clusters)
H1,1 l = [0, 100]	$A \rightarrow B$	[0, 20], [0, 50]
	$B \rightarrow C$	[20, 80], [50, 100]
	$B \rightarrow D$	[20, 52], [50, 100]
	$C \rightarrow B$	[80, 100]
	$D \rightarrow C$	[52, 100]
H1,2 l = [0, 80]	$A \rightarrow B$	[0, 20], [20, 80]
	$B \rightarrow C$	[20, 80]
H1,3 l = [0, 50]	$A \rightarrow B$	[0, 20] [0, 50]
	$B \rightarrow D$	[20, 52]
H1,4 l = [0, 20]	$A \rightarrow B$	[0, 20]

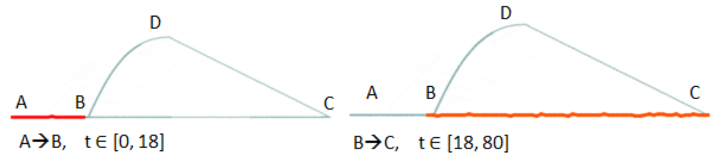
For instance, sub-chunk $H_{1,1}$ with lifespan [0, 100] (i.e. objects that move through out the dataset's lifespan) includes eight representatives, for each of which we note its lifespan. For example, in $H_{1,1}$ there are two subtrajectory clusters on the path $A \rightarrow B$, with lifespans [0, 20], [0, 50], respectively.

We have loaded the SMOD dataset to the *ReTraTree*. We set the temporal tolerance parameter to $\tau = 2$ (i.e. we impose 1 second difference in the starting/ending timepoints). The resulting *ReTraTree* discovered indeed four sub-chunks with lifespans: [0, 100], [0, 81], [0, 54] and [0, 20]. By incrementally applying *S^2T-Clustering* in each of them, we resulted in the discovery of the representatives. Figure 4.5 illustrates the representatives of the four sub-chunks. By combining each row in Table 4.3 with Figure 4.5(a)-(d), we conclude that *ReTraTree* discovers the correct representatives, with their lifespans only slightly deviating from ground truth.

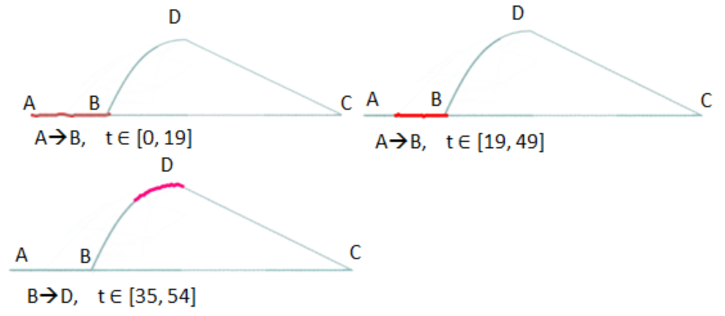
We now investigate how the *QuT-Clustering* algorithm would operate by setting the temporal period W e.g., to the whole lifespan of the dataset. We



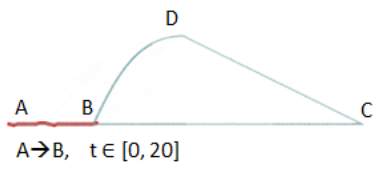
(a)



(b)



(c)



(d)

Figure 4.5: The representatives of the four sub-chunks.

used the values 5 sec, 10 m and 50% for (t, τ), d and γ respectively. After all *'append'* and *'merge'* operations take place, the resulting representatives are depicted in Fig.12, which is almost identical to the expected ground truth.

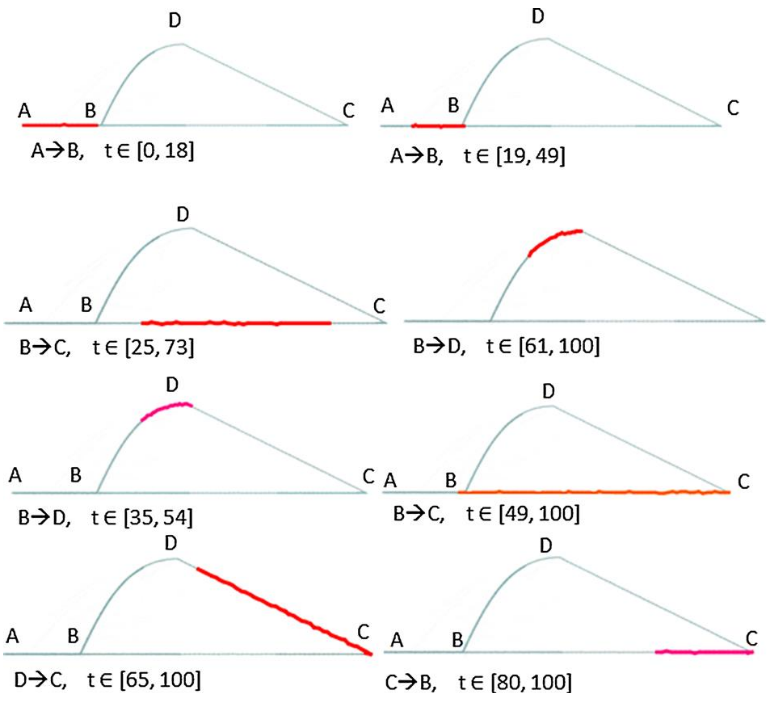


Figure 4.6: *QuT-Clustering* results with W=[0, 100].

In order to measure the stability of our method to noise effects, we have added more Gaussian white noise with Signal to Noise Ratio (SNR) level SNR = 30 db. The initial SMOD with additive noise of SNR = 50 db and the new SMOD with SNR = 30 db projected in 2-D spatial and 3-D spatiotemporal space is illustrated in Figure 4.7. A small number of objects (i.e. outliers, four in our experiment) randomly move in space other than the roads that the other objects reside. These are also depicted in Fig. 13. In addition, the speed of outliers is updated randomly. Furthermore, for the sake of simplicity we assume that the chunk size is the whole lifespan of the dataset. According to this, the ground truth is restricted to the eight different paths that are valid for sub-chunk $H_{1,1}$.

Given the above, and in order to demonstrate the benefits of S^2*T-Clustering* we compare with TRACLUS [49], the state-of-the-art subtrajectory clustering technique. Again we assume that the chunk size is the whole lifespan of the dataset, hence the ground truth restricts to the eight different paths that are valid for sub-chunk $H_{1,1}$. In Figure 4.8(a) and (b), we present the results of the

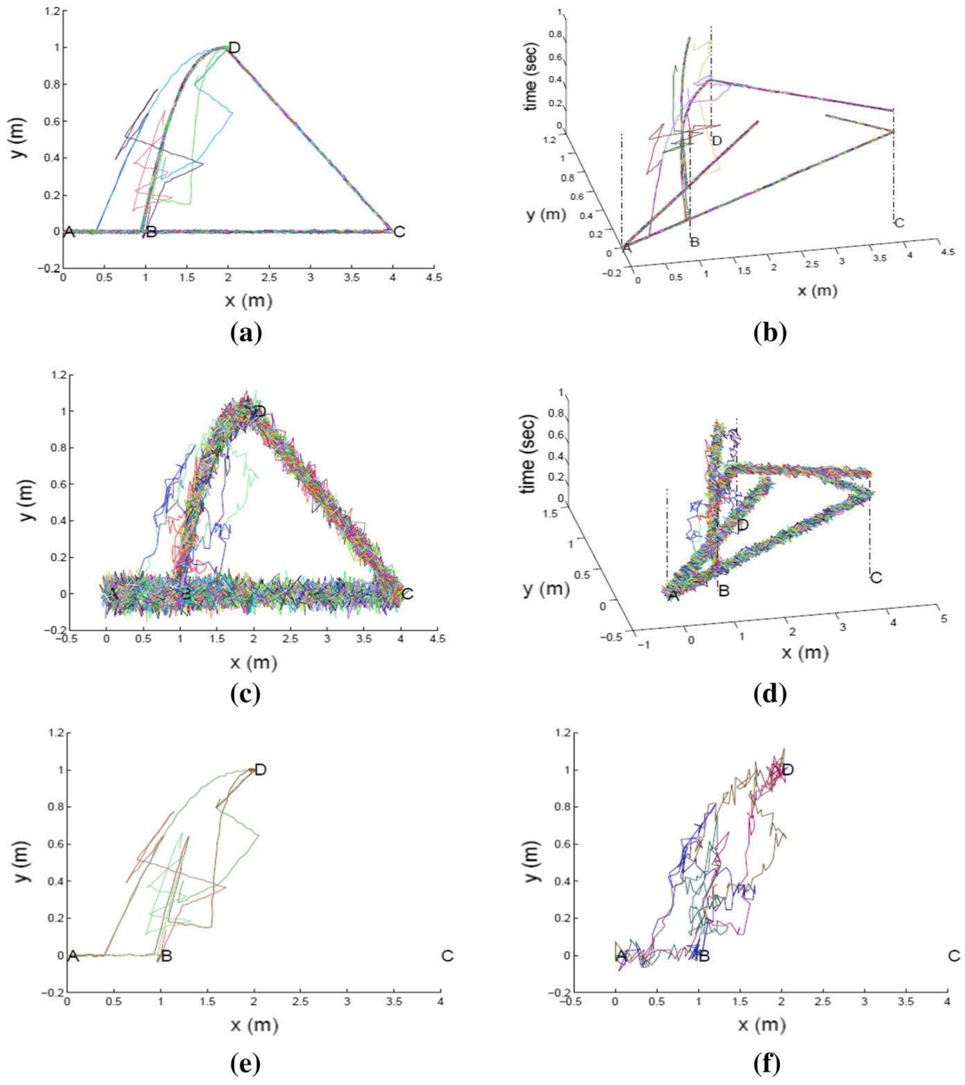
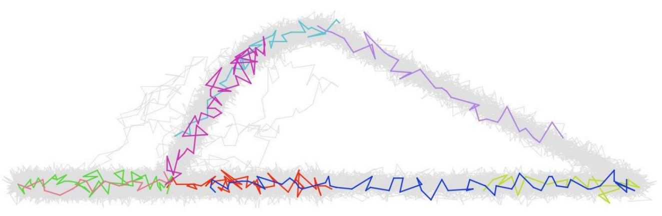
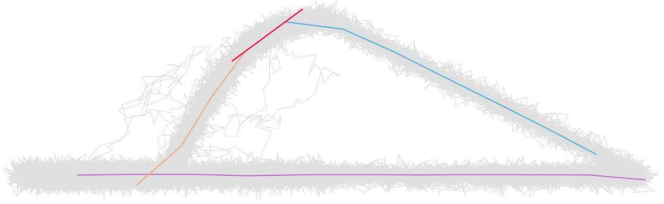


Figure 4.7: The trajectories of the SMOD with additive noise of SNR = 50 db projected in (a) 2-D spatial space ignoring time dimension and (b) spatiotemporal 3-D space. The trajectories of the SMOD with additive noise of SNR = 30 db projected in (c) 2-D spatial space and (d) spatiotemporal 3-D space. (e) The four outliers of the SMOD with additive noise of SNR = 50 db projected in 2-D spatial space ignoring time dimension. (f) The four outliers of our synthetic MOD with additive noise of SNR = 30 db projected in 2-D spatial space.



(a)



(b)

Figure 4.8: The representative trajectories (i.e. clusters) discovered by (a) S^2 *T-Clustering* (b) TRACLUS.

S^2 *T-Clustering* and TRACLUS, respectively. Specifically, in Figure 4.8(a) we depict the selected subtrajectories by S^2 *T-Clustering* to serve as the pivots (i.e. representatives) for grouping other subtrajectories around them, while in Figure 4.8(b) we depict the synthesized representatives extracted (with RTG algorithm [49]) after the TRACLUS's grouping phase. Based on this experiment, it turns out that S^2 *T-Clustering* effectively discovers all eight clusters (as well as the noisy subtrajectories), thus S^2 *T-Clustering* is not affected by the trajectories' shape, yielding an effective and robust approach for the discovery of linear and non-linear patterns. On the contrary, TRACLUS fails to identify the hidden ground truth in this SMOD (i.e. it discovers only four out of the eight clusters) due to the fact that it ignores the time dimension. Interestingly, note that TRACLUS discovers more or less linear patterns, ignoring the temporal information of the trajectories, as mentioned in [49].

In order to evaluate the accuracy of our proposal in a quantified way, we further employed F-Measure in SMOD. In detail, we built 8 datasets, with the first consisting of the subtrajectories of the first cluster of sub-chunk $H_{1,1}$ only, the second consisting of the subtrajectories of the first and the second cluster only, and so on, until the eighth dataset, which consisted of the subtrajectories of all eight clusters. All eight datasets appeared in two variations: including or not the set of outliers. For each dataset, we

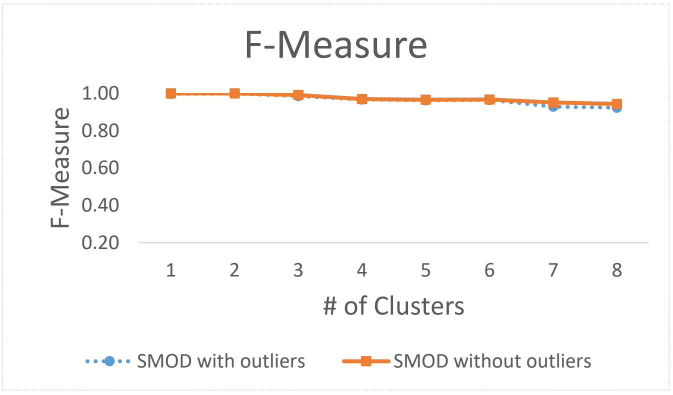


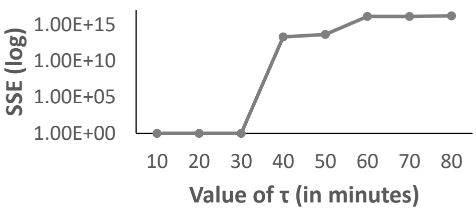
Figure 4.9: Quality of S^2T-*Clustering* w.r.t. number of clusters.

applied S^2T-*Clustering* and calculated F-Measure; Figure 4.9 illustrates this quality criterion by increasing the number of clusters. It is evident that S^2T-*Clustering* turns out to be very robust, achieving always precision and recall values over 92.3%, while the outliers are always detected correctly.

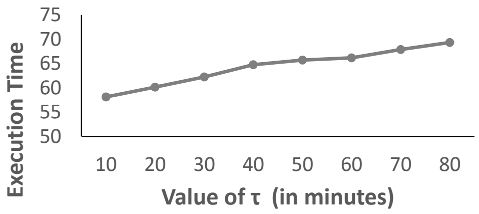
4.5.5 Sensitivity Analysis with Respect to Various Parameters

In this section we first study the effect on the quality of the clustering result when varying the values of the parameters of the *QuT-Clustering* algorithm. Recall that *QuT-Clustering* does not change the clusters of the trajectories organized in *ReTraTree*. It returns modified representatives which are valid inside the given time window W, by merging or appending the initial representatives. Thus, the goal of the experiment is to measure the difference between the representatives resulted by the *QuT-Clustering* and the initial representatives. Intuitively, having this difference for different values of various parameters gives us a good hint about the sensitivity of *QuT-Clustering* with respect to the various parameters. To measure the difference, we employ the SSE metric between the initial representatives and their counterparts returned by *QuT-Clustering*. Obviously, if a representative is returned as-is, it contributes 0 to SSE. Apart from this set of experiments (one for each parameter), we further measure the execution time of *QuT-Clustering*, so as to study the effect of the parameters in the efficiency of the algorithm, in contradiction with its quality.

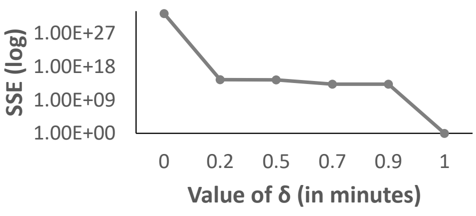
The results of these experiments on IMIS$_2$ dataset are depicted in Figure 4.10. More specifically, as depicted in Figure 4.10(a) as τ increases the quality drops



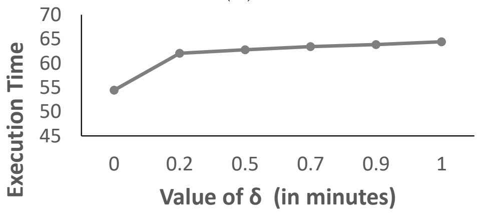
(a)



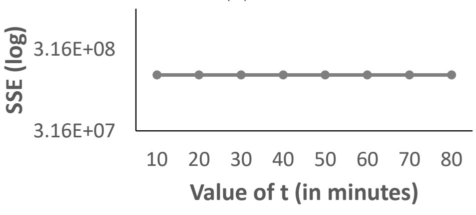
(b)



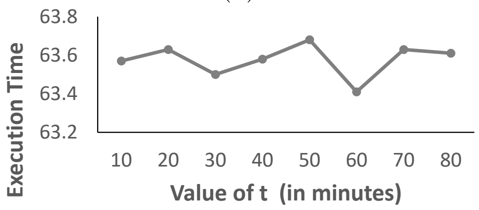
(c)



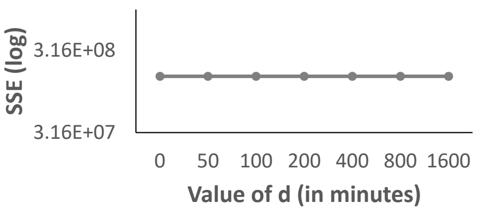
(d)



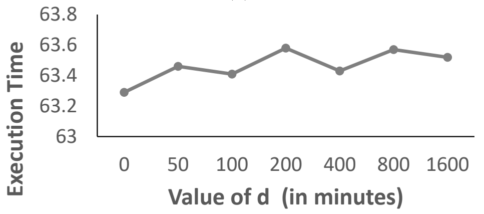
(e)



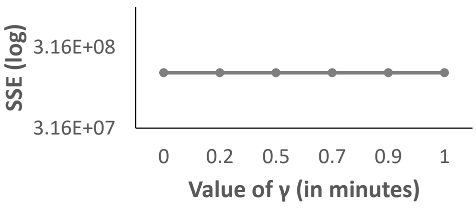
(f)



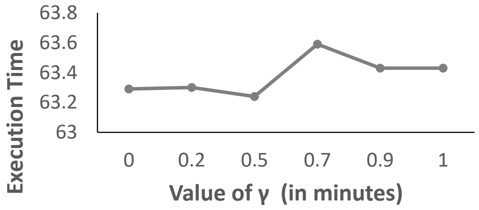
(g)



(h)



(i)



(j)

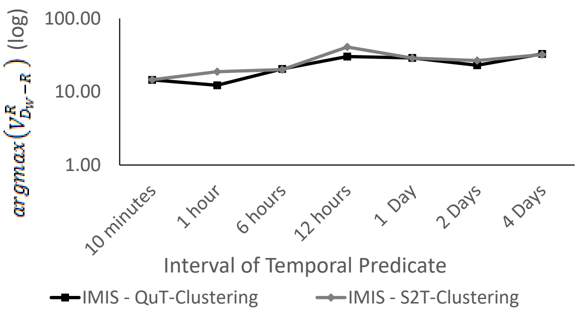
Figure 4.10: (a)-(c)-(e)-(g)-(i) Sum of Square Errors, (b)-(d)-(f)-(h)-(j) Execution time, when varying the parameters of *QuT-Clustering*.

due to the fact that we have more merges, hence the resulted representatives are more different than the original. The increasing number of merges results in gradual increase of the execution time (Figure 4.10(b)). Figure 4.10(c) shows that as δ increases the quality increases as fewer merges take place. Someone would expect that this would decrease execution time, however this is not the case as the costly operation in the merge phase is the calculation of the similarity between the two representatives, which is something that has already taken place. What we actually observe is a slight increase in the execution time, which occurs because *QuT-Clustering* ends up processing more representatives. (Figure 4.10(e), (g), (i) depict that quality is not affected by the different values of t, d and γ, respectively. The same conclusion stands also for the execution time illustrated in (Figure 4.10(f), (h), (j), respectively. This is because an append (raised when satisfying thresholds on these parameters) does not change the representatives themselves, i.e. an append simply returns two representatives as one.

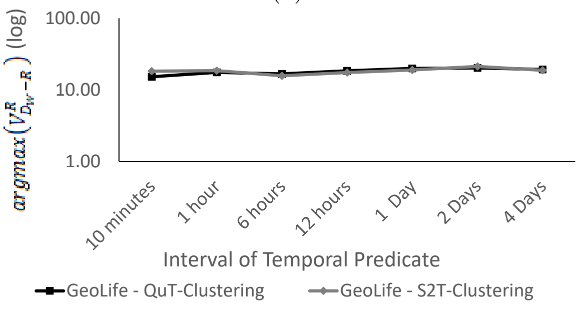
Given the above stable behavior of *QuT-Clustering* with respect to its parameters, in the rest of the experimental study the values of δ, d, γ, t and τ were set to the following intermediate values 0.7, 1km, 0.7, 30min and 30min, respectively.

4.5.6 Quality of Clustering Analysis in Real Datasets

In Section 4.5.4 we used a dataset including ground truth. In this section we use real datasets and general-purpose clustering validation metrics. Specifically, we evaluate the quality of the clustering through the $V^R_{D_W-R}$ measure introduced in Equation 4.6. Note that this measure stands as an alternative to the Sum of Square Errors (SSE) and QMeasure used in the evaluation of TRACLUS [49], as it accumulates the (normalized) distances from the cluster centroids. More specifically, we use the IMIS$_2$ and GeoLife real datasets and we compute $V^R_{D_W-R}$ for *S^2T-Clustering* and *QuT-Clustering* in different subsets of the two datasets. These subsets have been produced by selecting gradually coarser slices in the time domain. The time window W is set to lifespan of the subsets. The rationale of the experiment is that the $V^R_{D_W-R}$ of *QuT-Clustering* should be as close as possible to *S^2T-Clustering*, as the latter is a good solution of the problem for the degenerated case where the time window is equal to the lifespan of the dataset. Figure 4.11 confirms that *QuT-Clustering* is able to identify clusters as well as *S^2T-Clustering* does. Put differently, *QuT-Clustering* results in representatives (after all the



(a)



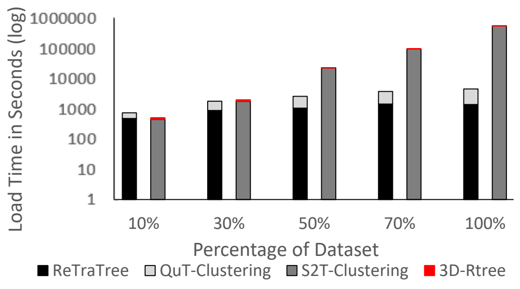
(b)

Figure 4.11: $V^R_{D_W-R}$ of *QuT-Clustering* and S^2*T-Clustering* against batches of varying lifespan (setting W to their whole lifespan): (a) IMIS$_2$, (b) GeoLife.

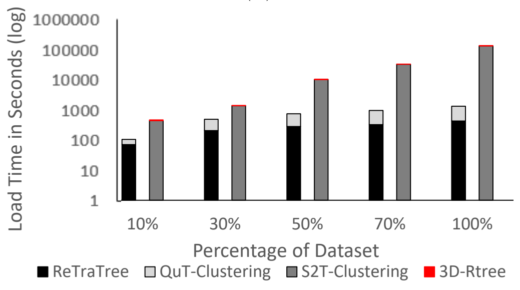
merge and append operations) which are very similar to those resulting from S^2*T-Clustering*, however as we will show in the subsequent sections, *QuT-Clustering* achieves this result with orders of magnitude better performance than S^2*T-Clustering*.

4.5.7 *ReTraTree* Maintenance

In this section, we evaluate three different aspects of the *ReTraTree* structure, namely the efficiency of (i) loading and (ii) appending data, as well as (iii) the size of the structure. More specifically, the Load operation, measures the required time to load increasing volumes of data from scratch, which correspond to partitions of the MOD that are produced by selecting randomly a percentage of the total number of trajectories. Figure 4.12 depicts the construction (loading) time to build, on the one hand, the *ReTraTree* and, on the other hand, the 3D-Rtree indices, i.e. the two alternatives to solve the problem at hand. Moreover, in order to correlate the required construction time of the indices with query time, we also add the execution time of the *QuT-Clustering* and the S^2*T-Clustering* algorithms, setting as time window



(a)



(b)

Figure 4.12: Construction time of *ReTraTree* vs. 3DR-Tree (and execution time of *QuT-Clustering* and S^2 *T-Clustering*) against datasets with increasing size: (a) IMIS$_2$, (b) GeoLife.

W equal to the whole lifespan of each dataset. Note the log-scale on y-axis.

From these results, we can make the following observations. First, the increase in loading time for *ReTraTree* is sublinear with respect to the dataset size, which is a positive testimony about its scalability. Second, when the total cost is considered (indexing and querying), it is clear that for large datasets our approach outperforms the competitor by two orders of magnitude. This is due to the fact that querying the *ReTraTree* is much more efficient than the 3D-Rtree, as the latter quickly becomes expensive, even for moderate dataset sizes. Put differently, *ReTraTree* harvests the increased construction cost in terms of fast query processing, thus boosting the performance of spatiotemporal clustering.

On the other hand, the *Append* operation measures the required time to append an existing *ReTraTree* with new batches of data, which correspond to new temporal periods – to perform this experiment we have split the datasets in 7 batches of equal duration (however, skewed size). Note that the first append operation loads data to an empty *ReTraTree*. Figure 4.13 illustrates the time for each batch of data to be appended in the two index structures.

Moreover, in the same figure we present the size of each batch. We observe that there is a very high correlation between batch size and batch execution time, perhaps with the exception of the first batch. This demonstrates that there exist no additional overheads as more batches are appended, thus the cost of *Append* mainly depends on the size of the appended batch. The fact that the first batch's execution time is disproportionate to its size has to do with the initialization cost of the *ReTraTree*.

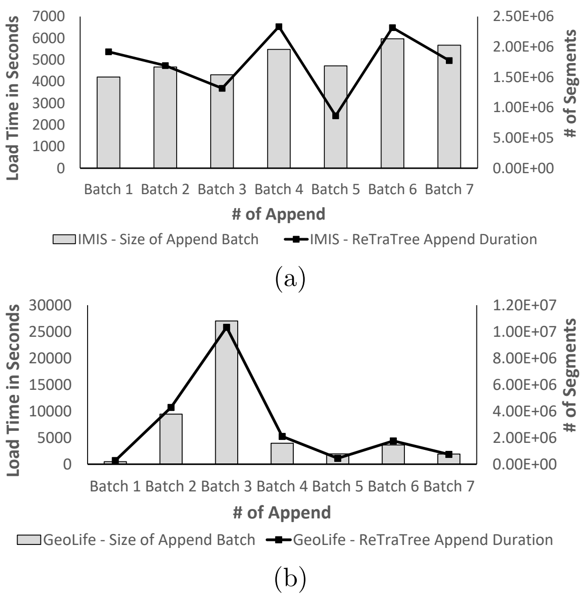
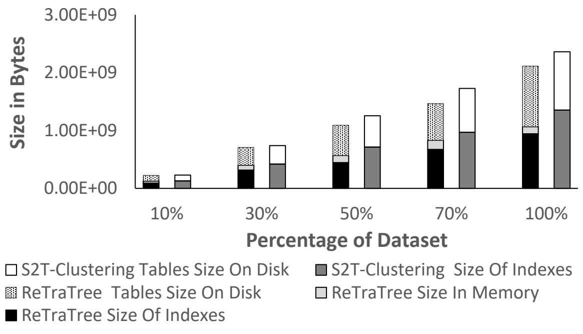


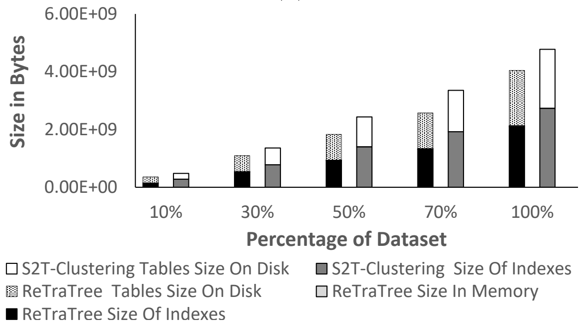
Figure 4.13: Append of *ReTraTree*: (a) IMIS_2, (b) GeoLife.

Next, we measure the size occupied by the structure. Figure 4.14 depicts the size of the *ReTraTree* structure, both on disk and in memory, and compares it with the size occupied by the indices required for the S^2*T-Clustering* algorithm. As we have an in-DBMS implementation, the size of the indices is augmented with the required B-trees on the primary keys of the database tables.

For clarity, we also present the size of original tables, namely a single table for the S^2*T-Clustering* case and multiple tables for *ReTraTree*. As expected, we observe that the first three levels of *ReTraTree* have a small in-memory footprint, while, notably, our approach has a smaller size on disk in contrast to 3D-Rtree. This is due to that the *ReTraTree*'s partitioning scheme leads to more compact 3D-Rtrees (i.e. less dead space).



(a)



(b)

Figure 4.14: Space requirements: (a) IMIS$_2$, (b) GeoLife.

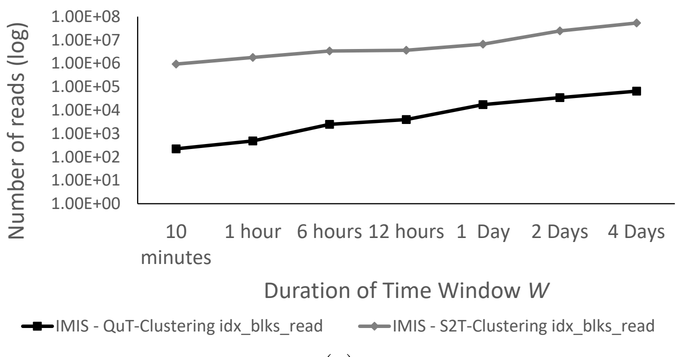
4.5.8 I/O Performance

Now, we evaluate the I/O performance of both *QuT-Clustering* and *S^{2}T-Clustering* with respect to the number of index blocks read from disk (*idx_blk_read*) and the ratio of the index page hits (i.e. blocks read from cache) with respect to to all blocks (*idx_hit_ratio*).

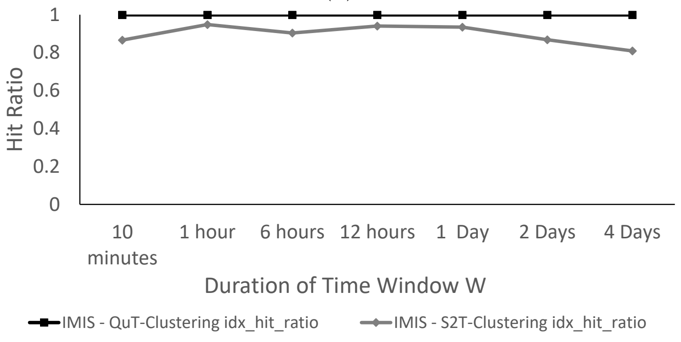
Figure 4.15(a) depicts the number of index blocks read from disk while increasing the duration of the time window W whereas Figure 4.15(b) illustrates the hit ratio that clearly shows the advantageous use of the index in our case. The results are for IMIS$_2$ dataset; we observed similar behaviour in GeoLife. Clearly, *QuT-Clustering* needs to access orders of magnitude fewer blocks to perform the clustering task, when compared to *S^{2}T-Clustering*. Moreover, this behaviour is consistent also for increased time windows.

4.5.9 Efficiency of QuT-clustering versus S^{2}T-Clustering

As a final experiment, we measure the efficiency of performing the clustering task. The goal is to evaluate the retrieval of all the valid maximal clusters



(a)



(b)

Figure 4.15: *QuT-Clustering* vs. *S^2T-Clustering* (IMIS$_2$ only): (a) blocks read from disk, (b) hit ratio.

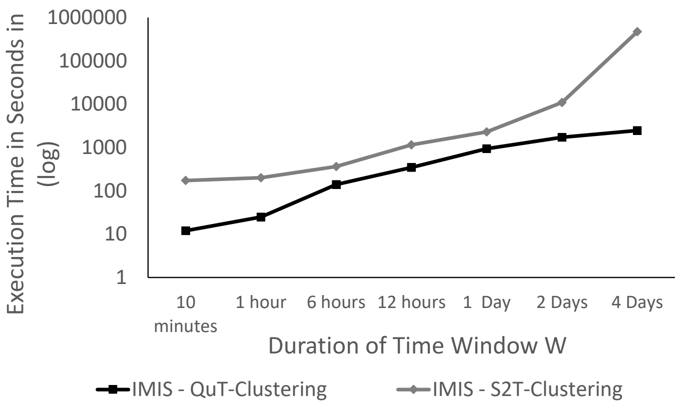


Figure 4.16: Execution time of *QuT-Clustering* vs. *S^2T-Clustering* by varying the datasets' lifespan (IMIS$_2$).

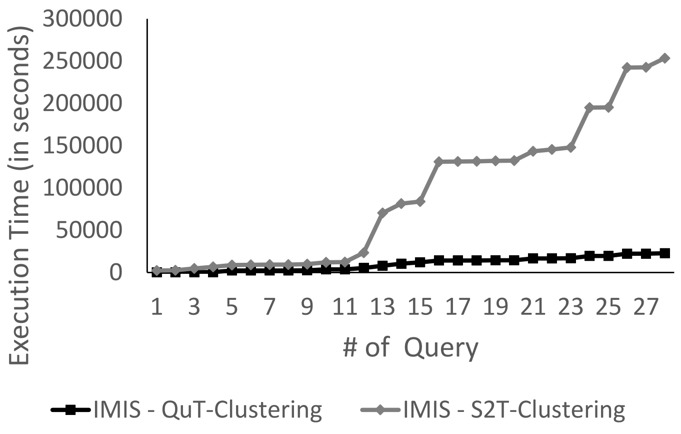


Figure 4.17: Accumulated execution time of *QuT-Clustering* vs. *S^2T-Clustering* w.r.t. a bundle of queries of random lifespan (IMIS$_2$).

for varying time windows W, which is critical for progressive clustering analysis. We compare *QuT-Clustering*, with an approach that first extracts the relevant records using a temporal range query, then creates a 3D-Rtree index on the extracted values, and then applies *S^2T-Clustering*. Figure 4.16 depicts the execution time of both approaches (using a log scale on the y-axis) by varying the duration of the time windows W. Again, it is clear that for large datasets our approach outperforms the competitor by two orders of magnitude.

Moreover, we created a bundle of queries with random lifespan (i.e. time window W) and we executed them with random sequence. Figure 4.17 depicts the accumulated execution time, i.e. time depicted for query $i + 1$ also includes time required for query i. This experiment clearly shows a major benefit of *ReTraTree*. More specifically, *S^2T-Clustering* presents an excessive cost in performing multiple clustering tasks (with different time windows W), while in the case of *ReTraTree* this cost simply disappears. In *ReTraTree*, the overhead of performing a new clustering is negligible, as depicted by the almost straight line in the chart. Both results are for IMIS$_2$ dataset; we observed similar behaviour in GeoLife (results omitted as they present no added value).

4.6 Summary

In this chapter, we introduced the *temporally-constrained subtrajectory cluster analysis* problem. To address it, we proposed *ReTraTree*, an indexing scheme which organizes trajectories by using an effective spatio-temporal partitioning technique. Partitions in *ReTraTree* correspond to groupings of subtrajectories, which are incrementally maintained and represented via a hierarchical organization of a small (thus, light-weight in-memory) set of 'representative' subtrajectories. Given this, the problem in hand can be efficiently solved as a query operator on *ReTraTree*, coined *QuT-Clustering*. Our approach further contributes to the mobility data management and mining domain for the additional reason that it has been designed and implemented in a MOD engine. Such functionality enables the application users to perform progressive cluster analysis via simple SQL in real extensible DBMS. Our extensive experimental study showed that our approach outperforms the state-of-the-art in-DBMS solution supported by PostgreSQL by several orders of magnitude.

5 Time-Aware Subtrajectory Clustering in Hermes@PostgreSQL

In this chapter, we present an efficient in-DBMS framework for progressive time-aware subtrajectory cluster analysis. In particular, we address two variants of the problem: (a) spatiotemporal subtrajectory clustering and (b) index-based time-aware clustering at querying environment. Our approach for (a) relies on a two-phase process: a voting-and-segmentation phase followed by a sampling-and-clustering phase. Regarding (b), we organize data into partitions that correspond to groups of subtrajectories, which are incrementally maintained in a hierarchical structure. Both approaches have been implemented in Hermes@PostgreSQL, a real MOD engine built on top of PostgreSQL, enabling users to perform progressive cluster analysis via simple SQL. The framework is also extended with a Visual Analytics (VA) tool to facilitate real world analysis. The original content of this chapter appears in [90].

5.1 Introduction

Knowledge discovery in mobility data [72] exposes patterns of moving objects useful in several fields, such as transportation, climatology, zoology, mobile social networks. Mobility (mostly GPS-based) data capturing results in trajectories of moving objects stored in Moving Object Databases (MOD), call for novel methods aiming at effective comprehension and analysis of mobility. In the literature of trajectory-based data mining [109], one can identify several types of mining models used to describe various collective behavioral patterns. Focusing on trajectory clustering, the majority of related work proposes a variety of distance functions, utilized by well-known clustering algorithms to identify collective behavior among entire trajectories. In a parallel line of research most related to the current approach, researchers aim to discover local patterns in MOD, i.e. co-movement patterns that are alive only for a portion of moving objects' lifespan, such as moving clusters, flocks, convoys, swarms, platoons and other patterns [109]. Other techniques, such as TRACLUS [49], simplify and partition the given trajectories and then apply density-based clustering, focusing on the spatial and ignoring the temporal dimension.

Exploration of clustering results is often supported by interactive Visual Analytics (VA) tools, as in the examples illustrated in Figure 5.1. The dataset employed for this example is a real MOD consisting of aircrafts approaching airports of the London metropolitan area. However, it is straightforward to employ datasets from other domains, such as maritime or urban traffic movement. For instance, the cluster affiliation of trajectory segments is represented on a map display by color coding. The user can interactively select which clusters to show or hide, in order to be able to examine selected clusters in detail. The existence times of the clusters and the changes of their cardinality over time can be explored using a time histogram, in which bars are divided into segments painted in the same colors as the cluster members in the map. The 3D shapes of the cluster members can be seen in a 3D display. In general, VA systems provide a number of visual and interactive techniques designed to support mobility data exploration and analysis [6].

Towards the goal of interactive mobility data exploration and analysis, our motivation in this work is to demonstrate how a MOD engine built on top of extensible DBMS can efficiently incorporate two subtrajectory clustering algorithms proposed recently, namely Sampling-based subtrajectory Clustering (*S^2T-Clustering*) [70] and Query-based Trajectory Clustering (*QuT-Clustering*) [69]. Interestingly, both algorithms operate on the entire spatiotemporal domain, by overpassing on the one hand some limitations of the state-of-the-art TRACLUS framework, while on the other hand eliminating hard-to-tune parameters as those introduced in the co-movement patterns approaches. Most importantly, with our approach we demonstrate the feasibility of progressive time-aware analytics, in terms that we allow a data analyst to select different time periods to perform his/her analysis, without being obliged to apply from scratch costly preprocessing or iterative clustering procedures.

The practical contribution of this work is that we present a framework that fulfils two significant specifications: (a) implements efficient and scalable solutions for subtrajectory clustering that (b) operate on a real-world DBMS

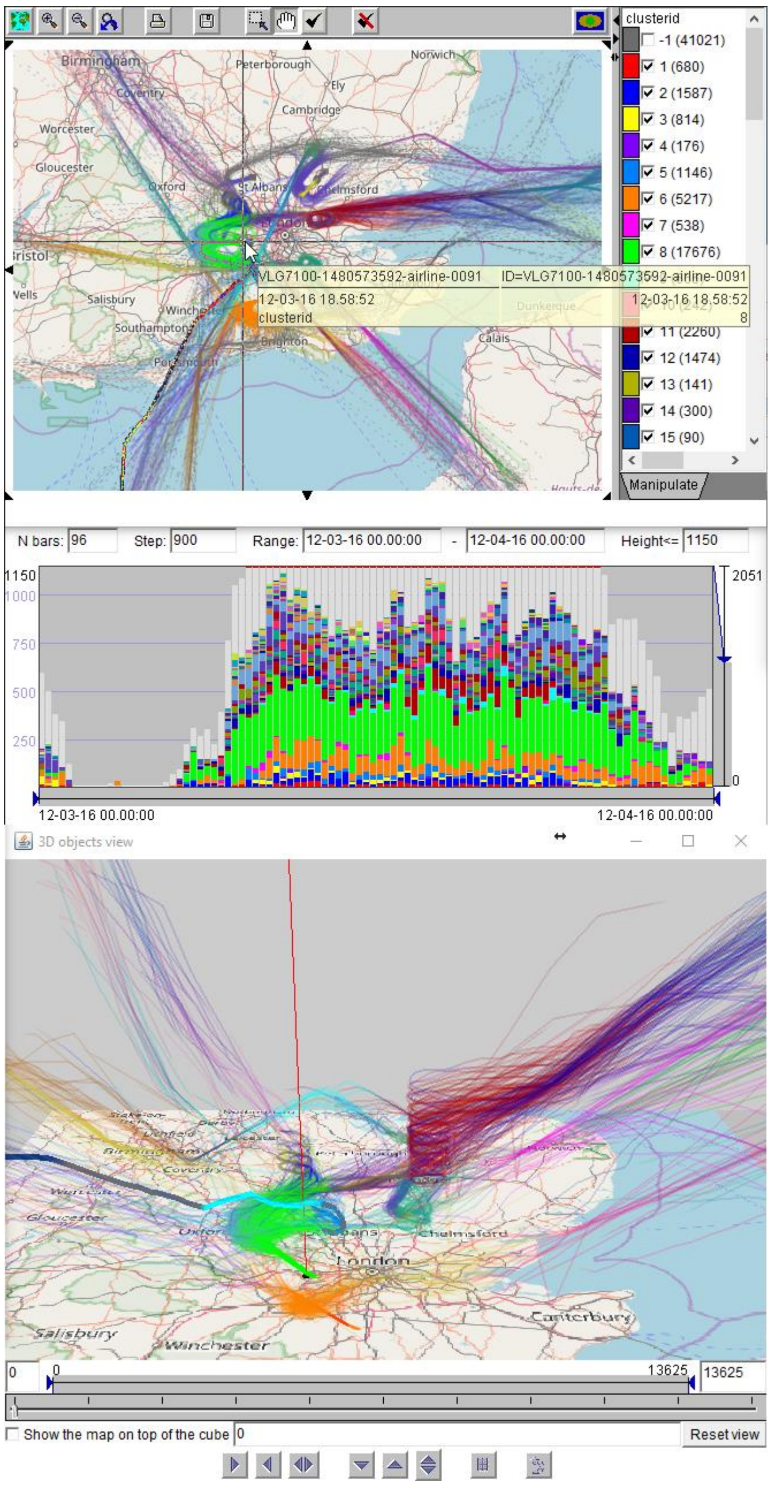


Figure 5.1: Interactive visual exploration of clustering results: map display of clusters (top); evolution of cardinality of clusters over time (middle); 3D shapes of cluster members (bottom).

rather than being ad hoc implementations. The in-DBMS implementation of our methods is performed in Hermes@PostgreSQL [1], our open source Moving Object Database (MOD) engine built on top of PostgreSQL, by using GiST [40] extensibility interface provided by PostgreSQL. To our knowledge, it is the first time in the literature that GiST is used to index trajectory-based mobility data for the above purposes. More specifically, GiST is utilized in order to build a 3D-RTree index tailored for trajectories. Therefore, we argue that this is a step towards bridging the gap between MOD management and mobility data mining, as state-of-art frameworks [6, 52, 99] could make use of the efficiency and the advantage of our approach to execute in-DBMS (sub-)trajectory clustering via simple SQL queries.

5.2 Major Modules and System Architecture

In this section, we present the details of the two major modules of our approach, namely *S^2T-Clustering* and *QuT-Clustering*. Then, we provide the overall architecture of our system.

5.2.1 S^2T-Clustering

In general, the objective of subtrajectory clustering is to partition trajectories into subtrajectories and then form groups of similar ones, while at the same time separate those (called outliers) that fit into no group. *S^2T-Clustering* [70] is a state-of-the-art algorithm consists of two phases: during the first phase, a Neighborhood-aware Trajectory Segmentation (NaTS) method is applied over trajectories, thus splitting them in subtrajectories; during the second phase, Sampling, Clustering and Outlier (SaCO) detection steps are performed in order to provide the final result. NaTS relies on a voting and segmentation process that detects homogenized subtrajectories in the MOD with respect to how many other objects move in their neighborhood, while SaCO selects the most representative ones to serve as the seeds of the clusters, around which the clusters are formed (also, the outliers are isolated). In more detail, during the adopted **voting** process each 3D trajectory segment of a given trajectory is voted by other trajectories with respect to their mutual distance. The voting received by each segment is a value ranging from 0 to N (N being the cardinality of the MOD) that has the physical meaning of how many trajectories co-move with that trajectory for a certain period of time. After the voting process takes place, the trajectory **segmentation** process follows. The goal of this step is to partition each trajectory into subtrajectories having homogeneous representativeness, irrespectively of their shape complexity. However, the goal of subtrajectory clustering is to partition the entire dataset into groups (clusters) and to detect the outliers among the subtrajectories identified by the trajectory segmentation step. Therefore, in our proposal, we first select the appropriate **sampling** set S and then, we tackle the problem of clustering according to the following idea: each subtrajectory in the sampling set is considered to be a cluster representative. So, the sampling set should contain highly voted trajectories of the MOD which, at the same time, would cover the 3D space occupied by the entire dataset as much as possible. Then, the clustering is done building the clusters "around" those representatives.

5.2.2 QuT-Clustering

In summary, *QuT-Clustering* [69] relies upon a hierarchical structure, called *ReTraTree* (for Representative Trajectory Tree) that effectively indexes a MOD for subtrajectory clustering purposes. *ReTraTree* consists of four levels: the first two levels operate on the temporal dimension, the third level builds clusters upon the spatio-temporal characteristics of the trajectories, and the fourth level is the actual data storage along with the corresponding indexes (3D-RTree) for effective retrieval. Given a MOD indexed according to *ReTraTree* structure and a temporal period W of interest, *QuT-Clustering* efficiently retrieves the subset of the MOD, actually the clusters and outliers at subtrajectory level, that temporally intersect W. The structure of the *QuT-Clustering* query in SQL follows:

$$\textbf{SELECT}\ QUT(D, W_i, W_e, \tau, \delta, t, d, \gamma);$$

where D is the dataset name, W_i and W_e are the initial and the ending time of the temporal period W, and τ, δ, t, d, γ correspond to the respective parameters of the *QuT-Clustering* algorithm [69].

5.2.3 System Architecture

The architecture of our framework is illustrated in Figure 5.2. As expected, the core of the framework is the *ReTraTree*. The trajectories composing the MOD are partitioned according to the in-memory part of the structure and stored on disk-based partitions. The trajectories assigned to an existing representative trajectory are archived on disk in dedicated R-tree indexed partitions (called ‘pg3D-Rtree-k’ in Figure 5.2). On the other hand, outlier trajectories are organized on disk in a separate partition. When the size of a partition exceeds a predefined threshold, S^2*T-Clustering* takes action: it applies Voting among trajectories, with the voting results indicating the Segmentation of trajectories into subtrajectories that should take place. The resulting subtrajectories along with their voting descriptors feed the Sampling module that selects new representatives, which are then back-propagated to the in-memory part of *ReTraTree*. The new representative trajectories as well as the raw subtrajectories form the input of the *GreedyClustering* module: if a subtrajectory is clustered around a new representative, it is archived in its appropriate partition on disk; otherwise, it is considered outlier and is re-inserted to *ReTraTree*, as it may now be accommodated in the index. Note that our implementation is completely independent from PostGIS. This implies that the underlying R-tree index, coined pg3D-Rtree in Figure 5.2, has also been implemented from scratch on top of GiST.

Having this functionality in hand, the data analyst is able to perform interactive clustering analysis, by providing different values of W as input, through either the SQL interface of Hermes@PostgreSQL [1] or the incorporated V-Analytics tool [6].

5.3 Demonstration of Results

Throughout the demonstration, users will be able to test the system using a real dataset of moving objects. The users will have the chance to catch a glimpse “under the hood”, experiment and visualize the results of some state of the art clustering techniques, such as [70] and [69]. More specifically, the demonstration captures the following phases - scenarios:

Preparatory phase (background knowledge): Initially, the user has the opportunity to comprehend the internals of our implementation and API, which exploits on the extensibility interface given by PostgreSQL. We show off the data types and operands resulting in Hermes@PostgreSQL MOD engine. In addition, we demonstrate how the user can use our SQL API to run all legacy operands, and even more interestingly, focus on the two subtrajectory clustering approaches, allowing orders of magnitude speedup in comparison to corresponding PostgreSQL functions [70].

In action phase – scenario 1: Having gained the necessary background

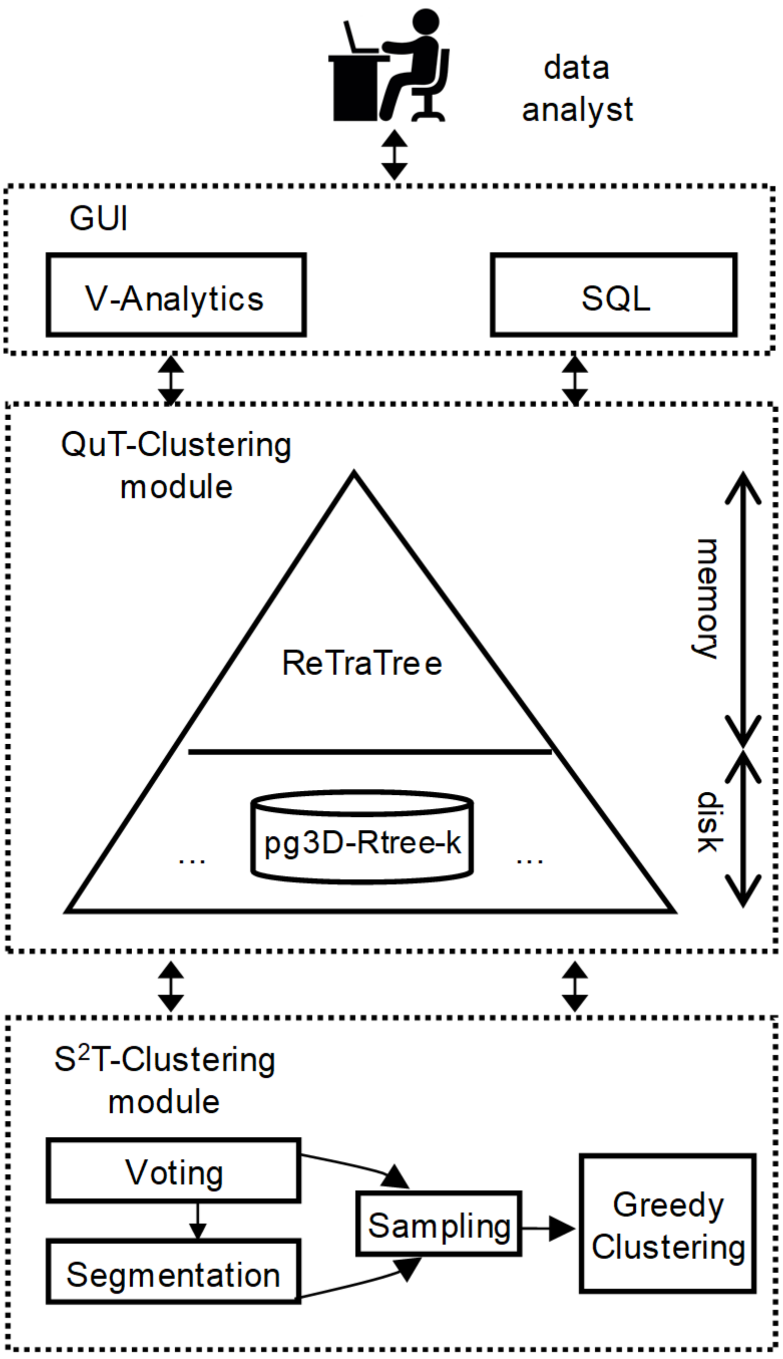


Figure 5.2: Architecture of the time-aware subtrajectory clustering module implemented in Hermes@PostgreSQL.

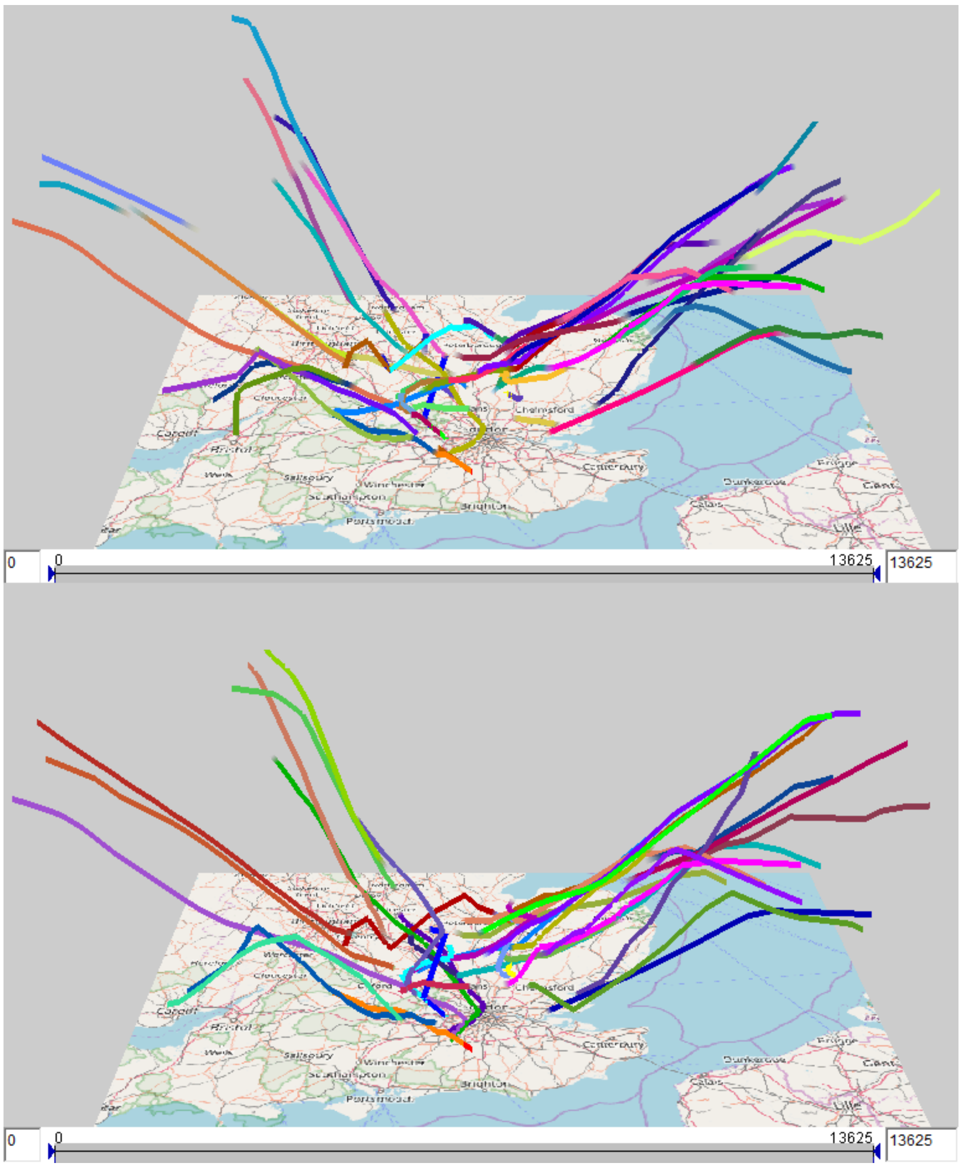


Figure 5.3: Time-aware subtrajectory clustering in action: cluster representatives from two different runs of S^2T-*Clustering* are visually compared by means of a 3D display.

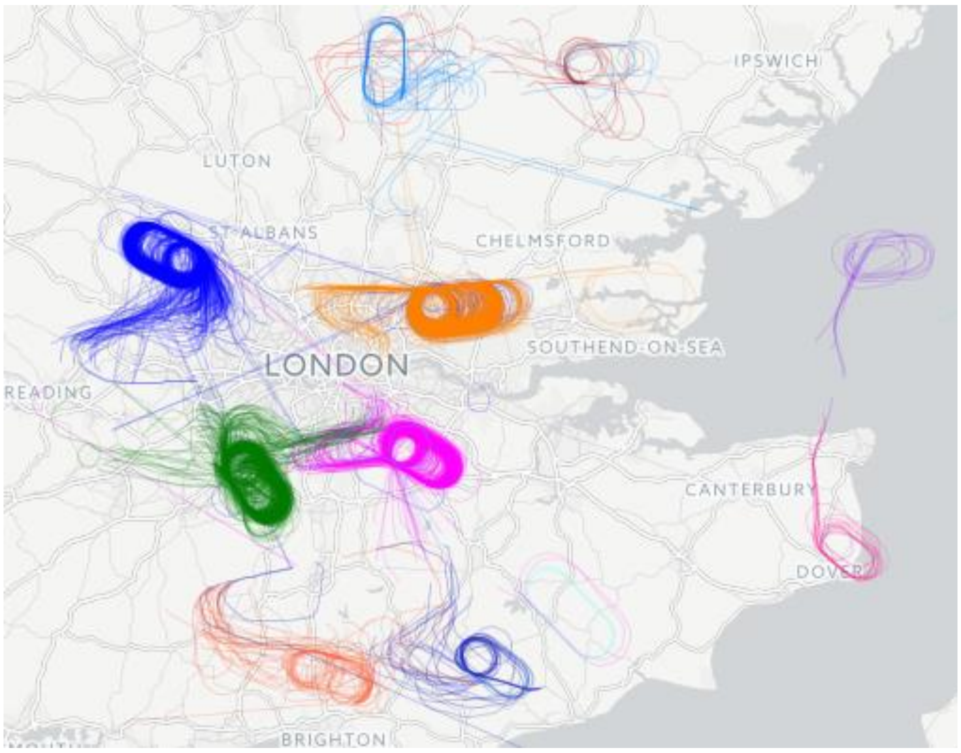


Figure 5.4: Time-aware subtrajectory clustering in action (cont.): holding patterns performed by aircrafts are discovered and visualized.

knowledge, the user experiences a progressive clustering scenario based on the S^{2}T-Clustering algorithm [70] as well as related methods, such as T-OPTICS [56], TRACLUS [49] and Convoys [44], and VA methods that aim at the analysis of the discovered patterns. For instance, Figure 5.3 presents an example of results of two runs of *S^{2}T-Clustering*, for comparison purposes. The cluster representatives from the two runs are viewed in a 3D display. The user can either put both sets of results in the same 3D display or create two 3D displays, each showing one set of results. In the first case, the user can interactively switch on and off the visibility of each set of results. The same can be done with a map display. Moreover, the user experiences in discovering and visualizing other interesting patterns, such as the holding patterns typically performed by aircrafts as they approach to their destination, in our case London airports (as it is illustrated in Figure 5.4).

In action phase – scenario 2: In turn, we present a progressive clustering scenario, this time focusing on the temporal dimension and highlighting the *QuT-Clustering* functionality. The goal of this scenario is twofold: first, we demonstrate via the SQL API the efficiency speedup of performing the clustering task for varying time periods W. We compare *QuT-Clustering* with the alternative approach that consists of (i) extracting the relevant records using a temporal range query, (ii) creating an R-tree index on the result of the query, and (iii) applying clustering (S^2T*-Clustering*, in our case). Second, we follow a similar approach, but this time we use the V-Analytics component of our framework in order to comprehend the evolvement of the subtrajectory patterns with increasing time periods W. In detail, by setting small value of W we focus on the landing phase of aircrafts and visualize the discovered clusters (recall, e.g., Figure 5.3); then, we increase the value of W to the past in order to realize the evolution of patterns as aircrafts pass from the cruising to the landing phase.

For deeper comprehension of both progressive analysis scenarios (S^2T*-Clustering* and *QuT-Clustering*) to be demonstrated, two related videos are available at Hermes@PostgreSQL demo web page[1].

5.4 Summary

In this chapter, we presented an efficient in-DBMS framework that facilitates progressive time-aware subtrajectory cluster analysis. In more detail, we tackled two variations of the problem: (a) spatiotemporal subtrajectory clustering and (b) on demand index-based time-aware clustering. The framework is also extended with a Visual Analytics (VA) tool to facilitate real world analysis. Having such functionality in their hands, data scientists are able to perform time-aware cluster analysis via simple SQL in real DBMS, where concurrency and recovery issues are taken into consideration.

[1]www.datastories.org/hermes/demo

Distributed Algorithms and Techniques

Part III

6 Distributed Subtrajectory Join on Massive Datasets

As already mentioned, performing advanced knowledge discovery operations, such as subtrajectory clustering (e.g., [70, 49, 3]), over immense volumes of data in a centralized way is far from straightforward and calls for parallel and distributed algorithms that address the scalability requirements. In more detail, the bottleneck of these approaches is that their computation raises efficiency issues due to the fact that all of them are actually based on a trajectory join query. Joining trajectory datasets is a significant operation in mobility data analytics and the cornerstone of various methods that aim to identify different kinds of mobility patterns (group behavior, etc.). In the era of Big Data, the production of mobility data has become massive and, consequently, performing such an operation in a centralized way is not feasible.

In this chapter, we address the problem of *Distributed Subtrajectory Join* processing by utilizing the MapReduce programming model. We propose three solutions: (i) a well-designed basic solution, coined $DTJb$, (ii) a solution that uses a preprocessing step that repartitions the data, labeled $DTJr$, and (iii) a solution that, additionally, employs an indexing scheme, named $DTJi$. In our experimental study, we utilize a 56GB dataset of real trajectories from the maritime domain, which, to the best of our knowledge, is the largest real dataset used for experimentation in the literature of trajectory data management. Our extensive experimental study is performed over a solid and realistic cluster setup, comprised of 49 nodes. The results show that $DTJi$ performs up to 16× faster compared with $DTJb$ and 10× faster than $DTJr$. An earlier version of the content of this chapter appears in [89].

6.1 Introduction

During the recent years, the proliferation of GPS enabled devices has led to the production of enormous amounts of mobility data. This "explosion" of data generation has posed new challenges in the world of mobility data management. One of these challenges is the so-called trajectory join problem, which aims to find all pairs of "similar" (i.e. nearby in space-time) trajectories in a dataset [11, 17, 22, 88]. An even more interesting and challenging problem is the subtrajectory join query [9], where, for each pair of trajectories, we want to retrieve all the "portions" of trajectories that are "similar". However, the subtrajectory join is a processing-intensive operation. Centralized algorithms do not scale with the size of today's trajectory data, thus parallel and distributed algorithms are necessary in order to provide efficient processing of subtrajectory join, an issue largely overlooked in the related research.

Several modern applications that manage trajectory data could benefit from such an operation. For instance, in the urban traffic domain, carpooling is becoming increasingly popular. More concretely, consider a mobile application which tries to match users that can share a ride based on their past movements. Here, given a set of trajectories we want to find all the pairs of users that can share a ride for a portion of their everyday routes without significantly deviating (spatially and temporally) from their daily routine (i.e. retrieve all pairs of maximal subtrajectories that move close in space and time). Another interesting scenario concerns the identification of suspicious movement by a governmental security agency. For instance, given a set of trajectories that depict the movement of suspicious individuals, we would like to retrieve all the pairs of moving objects that move "close" to each other for more than a threshold (moving together for small periods of time could be considered as coincidental) as candidates for illegal activity. Moreover, such a query is in fact the building block for a number of operations than aim to identify mobility patterns, such as co-movement patterns (e.g., flocks [37], convoys [44], swarms [51]). An even more challenging problem is that of subtrajectory clustering [70, 3]. An interesting application scenario of subtrajectory clustering is network discovery, where given a set of trajectories (e.g from the maritime or the aviation domain) we want to identify the underlying network of movement by grouping subtrajectories that move "close" to each other and use cluster representatives/medoids as network edges. One of the main goals of subtrajectory clustering is to segment trajectories to subtrajectories. Finally, trajectory segmentation techniques [62, 70], can directly benefit from the subtrajectory join query since their input, for each

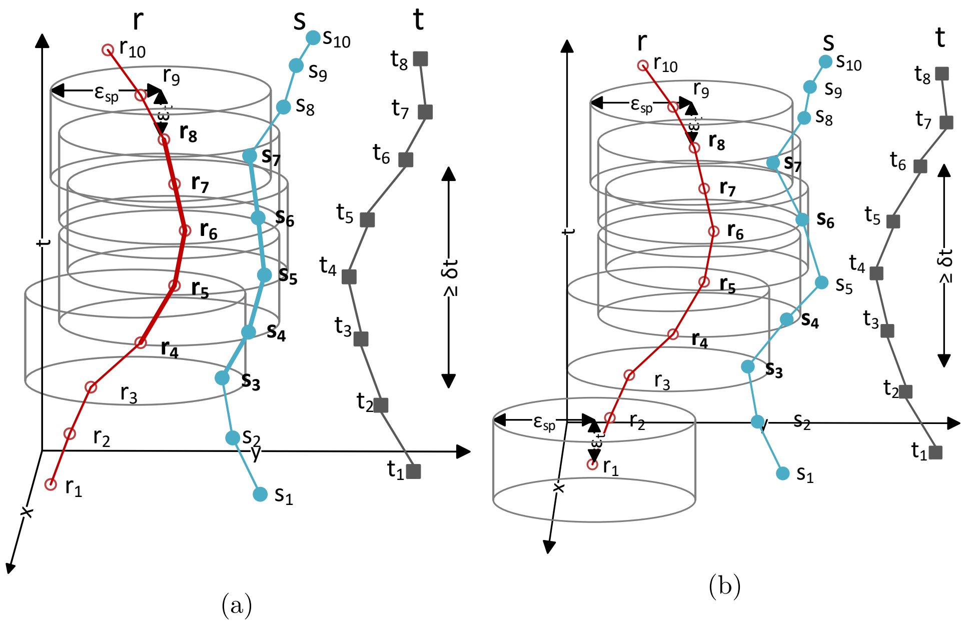


Figure 6.1: (a) A pair of maximally "matching" subtrajectories and (b) a breaking point r_1 and a non-joining point s_5 w.r.t. r.

trajectory, is the number of objects that were located close to it at any given time. However, the bottleneck in all these applications is the underlying processing cost of the join operation, which calls for parallel and distributed solutions that scale beyond the limitations of a single machine.

Inspired by the above application scenarios, the problem that we address in this chapter is as follows: given two sets of trajectories (or a single set and its mirror in the case of self-join), identify all pairs of maximal "portions" of trajectories (or else, subtrajectories) that move close in time and space with respect to a spatial threshold ϵ_{sp} and a temporal tolerance ϵ_t, for at least some time duration δt. To illustrate this informal definition, as depicted in Figure 6.1(a), given two trajectories r and s, the pair of their maximal matching "portions" is $(\{r_4, r_5, r_6, r_7, r_8\}, \{s_3, s_4, s_5, s_6, s_7\})$. Each point of a trajectory defines a spatiotemporal 'neighborhood' area around it, a cylinder of radius ϵ_{sp} and height ϵ_t. In order for a pair of subtrajectories to be considered "matching", each point of a subtrajectory must have at least one point of the other subtrajectory in its "neighborhood", thus making the result symmetrical. A pair of matching subtrajectories is maximal if there exists no superset of either subtrajectories that can replace them and the pair still qualifies as a "matching" pair.

There have been some efforts to tackle variations of this problem in a centralized way [9, 11, 17]. However, these solutions discover pairs of entire trajectories and cannot identify matching sub-trajectories. In [10], all pairs of "matching" (with respect to a spatial threshold) subtrajectories of exactly δt duration are retrieved in contrast with the problem addressed in this chapter, where the goal is to identify maximally "matching" subtrajectories, which is vital for exploiting the output in subsequent steps, e.g., the mining operations mentioned above. Moreover, applying these centralized solutions to a parallel and distributed environment is not straightforward and is often impossible if radical changes to the methods/algorithms do not take place, since there are several non-trivial issues that arise. For instance, how to partition the data in such a way so that each partition can be processed independently and be of even size.

In a recent effort, in [81] the authors try to tackle the problem of trajectory similarity join in spatial networks in parallel. The solution proposed in [81] handles each trajectory separately and all the data have to be replicated for each trajectory and, consequently, to each node. Due to this fact, such a solution cannot scale to terabytes of data, thus making it inapplicable to Big Data. Furthermore, such an approach assumes that the underlying network is known in advance, hence it cannot support datasets of moving objects that move freely in space (e.g., from the maritime or the aviation domain). As a result, a scenario where the goal is to identify the underlying network cannot be supported. Finally, the output of [81] is pairs of trajectories and not subtrajectories, which is significantly different than the problem addressed in this chapter. More recently, in [82] the authors try to tackle the problem of trajectory similarity join. Specifically, given two sets of trajectories, a similarity function (e.g., DTW) and a similarity threshold, they aim to identify all pairs of trajectories that exceed this similarity threshold. Again, the problem addressed in [82] is to retrieve pairs of trajectories in contrast with the problem that we try to tackle in this chapter, which is to retrieve all pairs of "maximally matching" subtrajectories. In another line of research, the authors in [29] tackle the problem of k-nn trajectory join in a distributed manner by employing the MapReduce programming model. In more detail, given two sets of trajectories R and M, an integer k and a time interval $[t_s, t_e]$, the goal is to return the k nearest neighbors from R for each object in M during this time interval. In order to achieve this, a five step procedure (five MR jobs) is adopted, where the data are preprocessed, subtrajectories are extracted, the time dependent upper bound is computed, candidates are found and the trajectories are joined. The intuition behind [29] is to find a distance upper bound d for each trajectory of M, that includes at least k trajectories from R and then perform a plane sweep distance join based on d. This approach, address an entirely different problem than the one presented in this chapter, because we retrieve all pairs of subtrajectories that moved "close enough" in space and time for at least some time duration.

It is straightforward to claim that an integral part of any algorithm that tries to address the subtrajectory join query is to identify all pairs of points that move "close enough" in time and space with respect to a spatial threshold ϵ_{sp} and a temporal tolerance ϵ_t, e.g., r_4 and s_3 in Figure 6.1(a). In that sense, another line of research that is closely related to our problem is that of MapReduce-based spatial [105, 5, 26] and multidimensional joins [86, 54, 32], where the goal is to identify such points. A generic solution which could form the basis for any MapReduce-based spatial (or spatiotemporal) join algorithm is presented in [105], where the input data are partitioned into small, disjoint tiles at *Map* stage and get joined at the *Reduce* stage by performing a plane sweep algorithm along with a duplication avoidance technique. However, all of the above approaches try to solve a problem that is significantly different from ours since our problem is not to join spatial or multidimensional objects but identify all pairs of "maximally matching" subtrajectories.

In this chapter, we provide efficient solutions for the *Distributed Subtrajectory Join* processing problem, as it is formally defined in Section 6.2. To the best of our knowledge, this problem has not been addressed in the literature yet. Our main contributions are the following:

- We formally define the problem of *Distributed Subtrajectory Join* processing, investigate its main properties, and discuss its main challenges.
- We present a well-designed algorithm, called *DTJb*, that solves the problem of *Distributed Subtrajectory Join* processing by employing two MapReduce phases.
- We propose an improvement of *DTJb*, termed *DTJr*, which is equipped with a repartitioning mechanism that achieves load balancing and collocation of temporally adjacent data.
- To boost the performance of query processing even further, we introduce *DTJi*, which extends *DTJr* by exploiting an indexing scheme that speeds up the computation of the join.
- We compare with an appropriately modified state of the art MapReduce spatial join algorithm and show that our solution performs several times

better.

- We study the performance of the proposed algorithms by using, to the best of our knowledge, the largest real trajectory dataset (56GB) used before in the relevant literature, thus demonstrating the scalability of our algorithms.

The rest of the chapter is organized as follows. In Section 6.2 we introduce the problem. In Section 6.3, we present *DTJb*. Subsequently, in Section 6.4 we propose *DTJr* that utilizes a preprocessing step. In Section 6.5, we introduce *DTJi* that boosts the performance of the join processing. In Section 6.6, we provide our experimental study. Finally, we conclude the chapter in Section 6.7.

6.2 Problem Statement

Given a set R of trajectories moving in the xy-plane, a trajectory $r \in R$ is a sequence of timestamped locations $\{r_1, \dots, r_N\}$. Each $r_i = (x_i, y_i, t_i)$ represents the i-th sampled point, $i \in 1, \dots, N$ of trajectory r, where N denotes the length of r (i.e. the number of points it consists of). The pair (x_i, y_i) and t_i denote the 2D location in the xy-plane and the time coordinate of point r_i respectively. A subtrajectory $r_{i,j}$ is a subsequence $\{r_i, \dots, r_j\}$ of r which represents the movement of the object between t_i and t_j where $i < j$.

Given a pair (r, s) of trajectories (the same holds for subtrajectories) with $r \in R$ and $s \in S$, the *common lifespan* $w_{r,s}$ is defined as the time interval $[max(r_1.t, s_1.t), min(r_N.t, s_M.t)]$, where r_1 (s_1) is the first sample of r (s, respectively) and r_N (s_M) is the last sample of r (s, respectively). The duration of the common lifespan $w_{r,s}$ is $\Delta w_{r,s} = min(r_N.t, s_M.t)$ - $max(r_1.t, s_1.t)$

Further, let $DistS(r_i, s_j)$ denote the spatial distance between two points r_i, s_j, which is defined as the Euclidean distance in this chapter, even though other distance functions are also applicable. Also, let $DistT(r_i, s_j)$ denote the temporal distance, defined as $|r_i.t - s_j.t|$. Table 6.1 summarizes the notations used throughout this chapter.

Definition 6.1. ***(Matching subtrajectories)*** *Given a spatial threshold ϵ_{sp}, a temporal tolerance ϵ_t and a time duration δt, a "match" between a pair of subtrajectories (r', s') occurs iff $\Delta w_{r',s'} \geq \delta t - 2\epsilon_t$, and $\forall r'_i \in r'$ there exists at least one $s'_j \in s'$ so that $DistS(r'_i, s'_j) \leq \epsilon_{sp}$ and $DistT(r'_i, s'_j) \leq \epsilon_t$, and $\forall s'_j$ there exists at least one r'_i so that $DistS(s'_j, r'_i) \leq \epsilon_{sp}$ and $DistT(s'_j, r'_i) \leq \epsilon_t$.*

Table 6.1: Table of Symbols used in Chapter 6

Notation	**Description**
R (or S)	A set of trajectories
r (or s)	A trajectory $\in R$ (S, respectively)
r_i (or s_j)	A point $\in r$ (s, respectively)
$DistS(r_i, s_j)$	The spatial distance between r_i and s_j
$DistT(r_i, s_j)$	The temporal distance between r_i and s_j
ϵ_t	Temporal tolerance
ϵ_{sp}	Spatial threshold
δt	Minimum duration of a "match"
$w_{r,s}$	Common lifespan of r and s
$\Delta w_{r,s}$	Duration of $w_{r,s}$
JP	The set of pairs of joining points
NJP	The set of pairs of non-joining points
BP	The set of breaking points
$sNJP$	The subset of pairs of non-joining points
$part_i$	The i-th temporal partition
t_s^{part}	The starting time of a partition
t_e^{part}	The ending time of a partition
$expPart_i$	The i-th temporal partition expanded by ϵ_t

Definition 6.2. ***(Maximally matching subtrajectories)*** *Given a pair of "matching" subtrajectories (r', s') which belong to trajectories r, s respectively, this pair is considered a "maximal match" iff $\nexists$ superset r'' of r' or s'' of s' where the pair (r'', s') or (r', s'') or (r'', s'') would be "matching".*

At this point, we should clarify that two trajectories may have more than one "maximal matches" (i.e. pairs of subtrajectories). Having provided the above background definitions, we can define the subtrajectory join query between two sets of trajectories.

Definition 6.3. ***(Subtrajectory join)*** *Given two sets of trajectories R and S, a spatial threshold ϵ_{sp}, a temporal tolerance ϵ_t and a time duration δt, the subtrajectory join query searches for all pairs (r', s'), $r' \in r \in R$ and $s' \in s \in S$, which are "maximally matching" subtrajectories.*

6.2.1 A Closer Look at the Subtrajectory Join Problem

An integral part of any algorithm addressing the subtrajectory join query, as defined in Definition 6.3 above, is to identify all *pairs of joining points*.

Definition 6.4. ***(Joining points)*** *A pair of points (r_i,s_j), where $r_i \in r$ and $s_j \in s$, is a pair of joining points iff they satisfy the following property:*

$$DistS(r_i, s_j) \leq \epsilon_{sp} \text{ and } DistT(r_i, s_j) \leq \epsilon_t.$$

In fact, the set of *joining points* is the outcome of the inner join $R \bowtie S$, where the evaluated join predicates are the ones mentioned above. However, as it will be explained next, these pairs of points do not suffice to return the correct query result.

A naive algorithm $A \in \mathcal{A}$ would require the Cartesian product $R \times S$ to produce the correct result. We claim that $R \times S$ can be represented by two sets of pairs of points, the set of *joining points* (JP) and the set of *non-joining points* (NJP). Formally, $R \times S = JP \cup NJP$.

The definitions of these sets follow, and the discussion is aided by Figure 6.1(b), which is a variation of Figure 6.1(a) in order to emphasize the distinction between JPs and $NJPs$.

The set NJP consists of the pairs of points that do not "match", coined *non-joining points*, since some of them might indicate the start or the end of "maximally matching" subtrajectories.

Definition 6.5. ***(Non-joining points)*** *A pair of points (r_i,s_j), where $r_i \in r \in R$ and $s_j \in s \in S$, are non-joining points iff r_i is not a joining point with s_j:*

$$DistS(r_i, s_j) > \epsilon_{sp} \lor DistT(r_i, s_j) > \epsilon_t.$$

This case is illustrated in Figure 6.1(b), where (r_5, s_5) is a pair of non-joining points or put differently r_5 is a non-joining point w.r.t. s_5 and vice versa.

A special case of non-joining points, called *breaking points* (BP), contains all points $r_i \in r \forall r \in R$ that are non-joining points w.r.t. any other point in S. The reason why we call such points as *breaking points* is that they essentially define the starting or ending of subtrajectories that could potentially belong to the answer set.

Definition 6.6. ***(Breaking points)*** *A point $r_i \in r \in R$ is a breaking point iff it is not a joining point with any other point $s_j \in S$:*

$$\nexists s_j \in S\text{: } DistS(r_i, s_j) \leq \epsilon_{sp} \land DistT(r_i, s_j) \leq \epsilon_t.$$

As it will be shown later, the lack of information about BPs can make an algorithm $A \in \mathcal{A}$ to falsely identify a pair of subtrajectories as "matching".

The set of BP along with the set of JP is actually the outcome of the full outer join of R and S. Figure 6.1(b) depicts the case where r_1 is a breaking point of r (r_2, r_3 and r_{10} are also breaking points), since it does not "match" with any other point of any trajectory. Obviously, breaking points are never reported as part of the answer set and the portion of r that could possibly contribute to the result is subtrajectory $r_{4,9}$. By differentiating *breaking points* from *non-joining points*, we reduce the amount of information that needs to be kept, i.e. instead of keeping multiple pairs of *non-joining points* we only keep one *breaking point.*

In the section that follows, we investigate the theoretical properties of an efficient algorithm in class $\mathcal{A}$.

6.2.2 Properties of Subtrajectory Join

In this section, we provide the theoretical properties for designing efficient algorithms for the subtrajectory join problem. The properties shown below essentially determine which pairs of points from the sets BP and NJP are necessary for a correct algorithm in class $\mathcal{A}$.

Lemma 6.1. *The set of breaking points is necessary in order to produce the correct result set for the Subtrajectory Join problem.*

Proof. It suffices to construct an instance of the problem where an algorithm A that operates solely on *joining points* and is unaware of *breaking points* would produce erroneous results, thus $A \notin \mathcal{A}$. Let $r' = r_{1,n}$ and $s' = s_{1,(n+1)}$ denote two subtrajectories, such that there exist n pairs of *joining points* (r_i,s_j) and $\Delta w_{r',s'} = \delta t$. However, let us assume that there exists a point $s_k \in s'$ which is a *breaking point.* Based on the problem definition (Definition 6.1), if algorithm A was unaware of *breaking points*, it would falsely identify r' and s' as "matching" subtrajectories. $\square$

This result indicates that *breaking points* cannot be ignored by an algorithm, without compromising the correctness of the result. The remaining question is whether all *non-joining points* are also necessary. In the following, we define a subset of *non-joining points* points $sNJP \subseteq NJP$, and show that this subset is actually necessary.

Definition 6.7. ***(Necessary subset** $sNJP$ **of non-joining points)*** *A pair of non-joining points* (r_i, s_j)*, where* $r_i \in r \in R$ *and* $s_j \in s \in S$*, belongs to*

$sNJP$, iff (a) $\nexists$ a point $s_p \in s$, with $p \neq j$, such that s_p is a joining point w.r.t. r_i, (b) $\nexists$ a point $s_q \in s$, with $q \neq j$, such that $DistT(r_i, s_q) \leq DistT(r_i, s_j)$ and (c) at least one of the adjacent points of r_i, r_{i-1} or r_{i+1}, is a joining point w.r.t. some point $s_t \in s$, with $t \neq j$.

Actually, condition (a) ensures that r_i is a non-joining point w.r.t. every point of s, (b) guarantees that s_j is the temporally closest point of trajectory s to r_i and (c) that at least one of the adjacent point of r_i is a joining point w.r.t. some point of trajectory s. Returning to the example of Figure 6.1(b), (s_5, r_5) qualifies to participate in $sNJP$, since (a) s_5 does not "match" with any point in r, (b) r_5 is the temporally closest point of r to s_5 and (c) at least one of adjacent point of s_5 (both s_4 and s_6 in this specific example) "match" with some point of trajectory r (r_5 and r_6, respectively). Again, failure to identify pairs of points such as (s_5, r_5) would result in erroneously identifying larger maximally "matching" subtrajectories. In a few words, $sNJP$ consists of all the pairs of non-joining points that signify the "beginning" or the "end" of candidate matching subtrajectories. In order for these candidates to qualify as matching subtrajectories, we need to further verify whether their common lifespan is larger or equal than $\delta t - 2\epsilon_t$, as depicted in Definition. 6.1.

Lemma 6.2. *The set $sNJP$ of pairs of non-joining points is necessary in order to produce the correct result set for the Subtrajectory Join problem.*

Proof. The proof is similar to the proof of Lemma 6.1, only using a *non-joining point* instead of a *breaking point* in the constructed instance of the problem. $\square$

In summary, our main finding is that a typical join algorithm that identifies only the set of JP is not enough in order to address the subtrajectory join problem. Additionally to the set of JP, an algorithm needs to identify both the set of BP and the subset $sNJP$ during the join processing, in order to ensure correctness. It is obvious, from Definition. 6.6 and Definition. 6.7, that $BP \cap sNJP = \varnothing$.

6.2.3 Distributed Subtrajectory Join

Given two sets R and S of trajectories, the typical approach for parallel join processing consists of two main phases: (a) *data repartitioning*, in order to create pairs of partitions $R_i \subset R$ and $S_j \subset S$, such that part of the join can be processed using only R_i and S_j, and (b) *join processing*, where a join algorithm is performed on partitions R_i and S_j.

Problem 6.1. *(***Distributed Subtrajectory Join***) Given two distributed sets of trajectories, $R = \cup R_i$ and $S = \cup S_j$, compute the subtrajectory join (Definition 6.3) in a parallel manner.*

In this setting, the main challenges are the following: (a) ensure that the created partitions are sufficient to produce parts of the total join without additional data, (b) generate even-sized partitions in order to balance the load fairly to multiple nodes, (c) handle the problem of potential duplicate existence in the join results, which may arise due to the way partitions are created, and (d) process the actual join on the partitions in an efficient way. The first challenge sets the foundations for *parallel processing*, as it identifies pairs of partitions that can be processed together, without any additional data, and produce a subset of the final join result. The second challenge is about *load balancing* and determines the efficiency of parallel processing, which is not straightforward, since processing uneven work units in parallel may lead to sub-optimal performance (as the slowest task will determine the query execution time). The third challenge, labeled *duplicate avoidance*, is about avoiding to generate duplicate results which typically occurs in parallel join processing. Finally, the fourth challenge, labeled *efficient join*, refers to the efficiency of the (centralized) algorithm used to join two partitions.

Clearly, solving the above problem is quite challenging in a distributed setting, as multiple challenges need to be addressed at the same time. In the following sections, we present a well designed solution to the *Distributed Subtrajectory Join* problem, named *DTJb* along with two improved versions, coined *DTJr* and *DTJi*, following the popular MapReduce paradigm. In more detail, as depicted in Table 6.2, *DTJb* consists of two MapReduce jobs and provides a duplicate avoidance mechanism. On the other hand, *DTJr*, consists of one MapReduce job provides a duplicate avoidance and a load balancing mechanism. Both *DTJb* and *DTJr* have a $O(n^2)$ time complexity, with *DTJr* being more efficient than *DTJb* due to the fact that *DTJr* consists of 1 job and provides a load balancing mechanism. Finally, *DTJi* extends *DTJr* with an indexing mechanism which improves the time complexity (O(nlogn)) and further boosts the performance by approximately an order of magnitude, compared to *DTJr*.

Table 6.2: Comparison between the proposed solutions

Solution	**# of MR Jobs**	**Duplicate Avoidance**	**Load Balancing**	**Indexing**	**Join Complexity**
DTJb	2	✓	×	×	O(n^2)
DTJr	1	✓	✓	×	O(n^2)
DTJi	1	✓	✓	✓	O(nlogn)

6.3 The Basic Subtrajectory Join Algorithm

6.3.1 Preliminaries

One of the prevalent technologies for dealing with Big Data and offline analytics, is the MapReduce programming paradigm [20] and its open-source implementation Hadoop [85]. A lot of efforts have been made as far as it concerns join processing through this technology and a survey on limitations of MapReduce/Hadoop, also related to join processing, is conducted in [25]. In more detail, Hadoop is a distributed system created in order to process large volumes of data which are usually stored in the Hadoop Distributed File System (*HDFS*). When running a MapReduce (MR) job, each *Mapper* processes (in parallel) an input split, which is a logical representation of data. An input split typically consists of a block of data (the default block size is 128MB) but it can be adjusted according to the users' needs by implementing a custom *FileInputFormat* along with the corresponding *FileSplitter* and *RecordReader*. Subsequently, for each record of the split the "map" function is applied. The output of the *Map* phase is sorted and grouped by the "key" and written to the local disk. Successively, the data is partitioned to *Reducers* based on a partitioning strategy (also known as *shuffling*), and each *Reducer* receives a partition (group) of data and applies the "reduce" function to the specific group. Finally, the output of the *Reduce* phase is written to *HDFS*.

In recent years, Spark [103] has received much attention and it has demonstrated to be more efficient than MapReduce, and its open source implementation Hadoop. In [83], it is shown that Spark outperforms Hadoop in the majority of operations, such as word count, k-means and page-rank. However, the only case where Hadoop presents better performance than Spark, as presented in [83] , is the case of sort. The reason for this behavior is that, in case the intermediate result between the Map and Reduce phase is very large and the shuffle selectivity is high (i.e., the ratio of the map output size to the job input size), Hadoop, unlike Spark, can overlap the shuffle stage with the map stage, which effectively hides the network overhead. Actually, the intermediate result when performing the proposed subtrajectory join operation can be several times larger than the original dataset, depending on the values of e_{sp} and e_t, which motivates us for using Hadoop over Spark.

6.3.2 The DTJb Algorithm

Our first algorithm, named *DTJb*, consists of three phases: (a) the *Partitioning phase*, where input data is read and partitioned, (b) the *Join phase*, where the sets JP, BP and $sNJP$ are identified in each partition, and (c) the *Refine phase*, where these sets are grouped by trajectory and sorted by time in order to identify all the pairs of “maximally matching” subtrajectories[1].

Partitioning Phase

The first challenge is how to partition the input data in order to satisfy the requirement for *parallel processing.* Partitioning the data into N disjoint temporal partitions $R = \cup_{i=1}^{N} part_i$, where R is the set of trajectories, cannot guarantee the correctness during parallel processing, due to the temporal tolerance parameter ϵ_t. Hence, we define a partitioning where each $part_i$ is expanded by ϵ_t, thus expanded partitions can be processed independently in parallel. Let $expPart_i$ denote such an expanded partition. Processing each $expPart_i$ individually guarantees correctness, but at the cost of having duplicates due to the point replication in temporally overlapping partitions. To address this *duplication avoidance* challenge, we supplement each point with a flag *partFlag* that indicates whether this point belongs to the original partition (i.e. not expanded by ϵ_t) or not.

Lemma 6.3. *An expanded partition $expPart_i$ is sufficient in order to produce the sets of JP and BP for $part_i$*

[1]For the sake of simplicity, from now on, we are going to consider the case of self-join. The transition to the problem of joining two relations is straightforward.

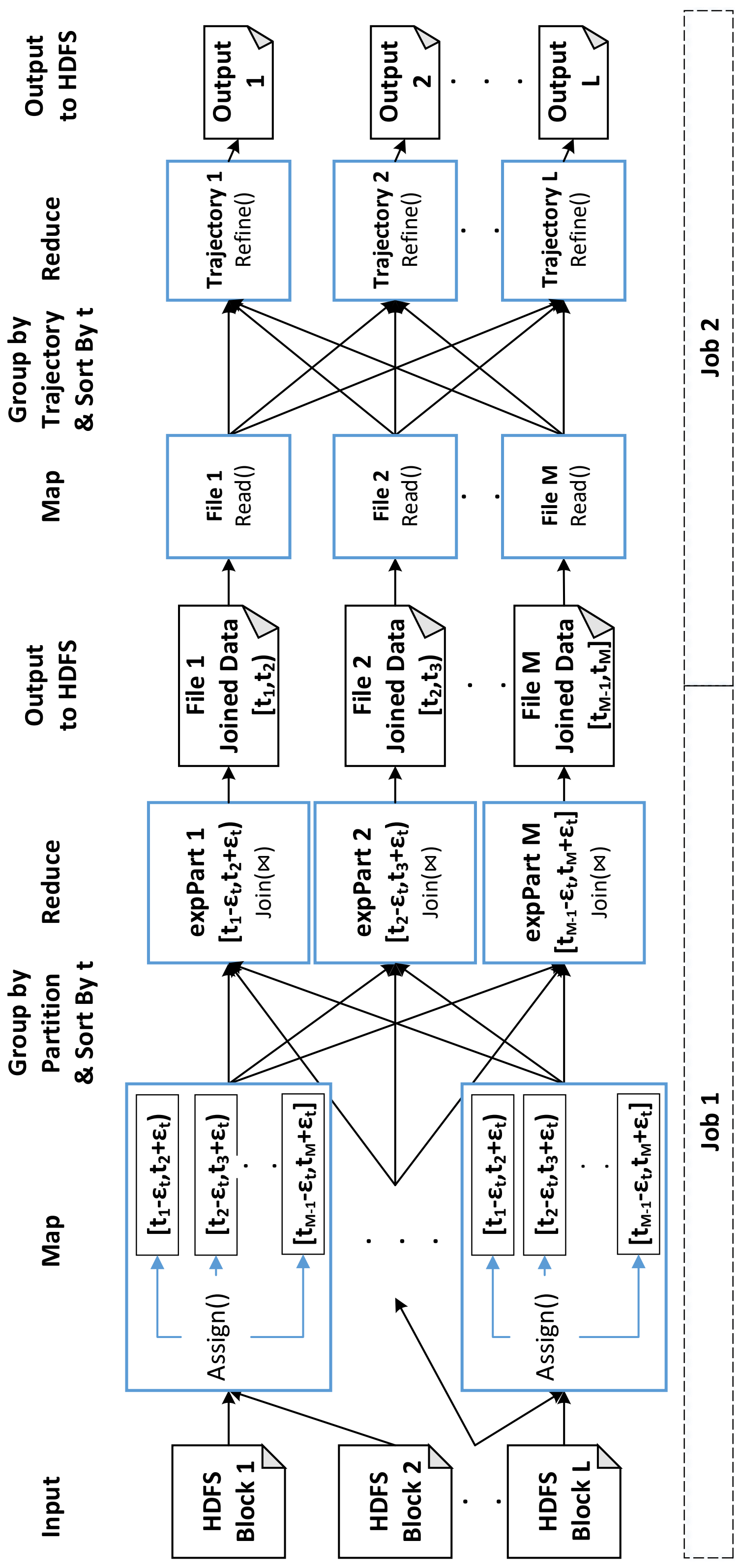

Input
Map
Group by Partition & Sort By t
Reduce
Output to HDFS
Map
Group by Trajectory & Sort By t
Reduce
Output to HDFS
HDFS Block 1
HDFS Block 2
HDFS Block L
Assign()
$[t_1-\varepsilon_t,t_2+\varepsilon_t]$
$[t_2-\varepsilon_t,t_3+\varepsilon_t]$
$[t_{M-1}-\varepsilon_t,t_M+\varepsilon_t]$
expPart 1 $[t_1-\varepsilon_t,t_2+\varepsilon_t)$ Join(⋈)
expPart 2 $[t_2-\varepsilon_t,t_3+\varepsilon_t)$ Join(⋈)
expPart M $[t_{M-1}-\varepsilon_t,t_M+\varepsilon_t]$ Join(⋈)
File 1 Joined Data $[t_1,t_2)$
File 2 Joined Data $[t_2,t_3)$
File M Joined Data $[t_{M-1},t_M]$
File 1 Read()
File 2 Read()
File M Read()
Trajectory 1 Refine()
Trajectory 2 Refine()
Trajectory L Refine()
Output 1
Output 2
Output L
Job 1
Job 2

Figure 6.2: The DTJb algorithm in MapReduce.

Proof. **Joining points**: By contradiction. Let us assume that an expanded partition $expPart_i$ is not sufficient to produce the set of JP in $part_i$. Then, there must exist a pair of joining points $r_j \in r$ and $s_k \in s$, such that r_j belongs to partition $part_i$, whereas s_k does not belong to $expPart_i$. Based on the definition of expanded partitions, it follows that $DistT(r_j, s_k) > \epsilon_t$. Thus, r_j and s_k cannot be joining points, which is a contradiction. **Breaking points**: By contradiction. Let us assume that an expanded partition $expPart_i$ is not sufficient to produce the set of BP for trajectories in $part_i$. Then, there must exist a point r_* (that belongs to $part_i$) of trajectory r, such that r_* is a breaking point. To identify if r_* is a breaking point, we need to examine whether there exists a point $s_j \in s$ of any other trajectory s with $DistT(r_*, s_j) \leq \epsilon_t$ (Definition 6.6). However, based on the definition of expanded partitions, such a point s_j must belong to $expPart_i$, which contradicts with the assumption that $expPart_i$ is not sufficient. $\square$

Unfortunately, an expanded partition $expPart_i$ is not sufficient in order to produce the set of $sNJP$ since, according to Definition 6.7, for each pair (r_j, s_k) that belongs to NJP we need to examine r_{j-1} and r_{j+1}, which may span to other partitions. However, the set of $sNJP$ can be identified at the *Refine* phase, where all the pairs concerning a trajectory are grouped together.

In more detail, we choose to partition the data into uniform temporal partitions, where for each pair of partitions $(part_i, part_j)$, with $i \neq j$ and $i, j \in [1, N]$, it holds that $DistT(t_e^{part_i}, t_s^{part_i}) = DistT(t_e^{part_j}, t_s^{part_j})$. Typically, the duration of a partition is larger than the maximum interval between two consecutive points of any trajectory. As illustrated in Figure 6.2, in the *Map* phase we access each data point and assign it to the expanded partition with which it intersects, essentially applying a temporal range partitioning. Then, the data is grouped by expanded partition, sorted by time and fed to the *Reduce* phase, where the *Join* procedure takes place.

Join Phase

Figure 6.2 shows that each *Reducer* task takes as input an expanded partition and performs the *Join* operation. At this point, the duplication avoidance technique is applied, by employing the aforementioned flag and emitting only pairs where at least one point belongs to the original partition. The input of this phase is a set of tuples of the form $\langle t, x, y, trajID, partFlag \rangle$. The output of this MR job is a set of (a) JP, (b) BP and (c) candidate $sNJP$.

In more detail, we apply a plane sweep technique in order to perform the *Join*, by sweeping the temporal dimension. We choose to employ such a technique due to the fact that is much more efficient than a nested-loop join approach, since our data already arrive sorted by the temporal dimension, as illustrated in Figure 6.2. A typical plane sweep algorithm would emit only the set of JP, which is not enough in our case. For this reason, we devised and implemented a modified plane sweep technique, named *TRJPlaneSweep*, depicted in Figure 6.3, which also reports the sets of BP and candidate $sNJP$.

Algorithm 6.1 Join($expPart, \epsilon_{sp}, \epsilon_t$)

```
1: **Input:** An $expPart$, $\epsilon_{sp}$, $\epsilon_t$
2: **Output:** All pairs of $JP$, $BP$ and candidate $sNJP$
3: for each $point\ i \in expPart$ do
4:     $D[i] \leftarrow point$
5:     TRJPlaneSweep($D[], \epsilon_{sp}, \epsilon_t$)
6: TreatLastTrPoints()
7: for each $point\ j \in BP[]$ do
8:     output(($BP[j], null$), True)
```

Algorithm 6.1 presents how the *Join* processing is performed. Each accessed point is inserted to an array D, which contains points sorted in increasing time. After point insertion, (Algorithm 6.2 is invoked for the currently accessed point (say $D[i]$) if $D[i]$ belongs to the original partition. *TRJPlaneSweep* examines the previously accessed points for the previous ϵ_t window (line 4). The role of this function is threefold. First, it identifies *joining points* with $D[i]$, e.g., point $D[j]$, and emits them in the form $((D[i], D[j]), True)$ (lines 5-10). Depending on the outcome of the duplicate avoidance technique, pairs $((D[j], D[i]), True)$ are also output. Second, it discovers points that belong to the candidate $sNJP$ set by examining whether the previous trajectory point (*getPrevTrPoint*)) of $D[j]$ (and $D[i]$), say $D[k]$, is a NJP (*FindMatch*) with each point $\in D[i].trajID$ ($D[j].trajID$, respectively) (lines 11-17). In case such points are identified, they are output with a different flag $((D[i], D[k]), False)$ to differentiate them from JP. Third, it discovers the points that belong to BP. In more detail, in lines 18–19, a *breaking point* $D[i]$ is added to the breaking points set BP and in lines 7 and 10 is removed if a point has a "match". The remaining points in BP are reported as breaking points, using the following form: $((D[i], null), True)$ (Algorithm 6.1 lines 7–8).

Algorithm 6.2 TRJPlaneSweep($D[], \epsilon_{sp}, \epsilon_t$)

```
1: Input: $D[]$, $\epsilon_{sp}$, $\epsilon_t$
2: Output: All pairs of $JP$, $BP$ and candidate $sNJP$
3: if $D[i].partFlag$=True then
4:    for each element $D[j] \in [D[i].t - \epsilon_t, D[i].t]$ do
5:       if $DistS(D[i], D[j]) \leq \epsilon_{sp}$ then
6:          output($(D[i], D[j])$, True)
7:          remove $D[i]$ from $BP[]$
8:          if $D[j].partFlag$=True then
9:             output($(D[j], D[i])$, True)
10:            remove $D[j]$ from $BP[]$
11:         $k \leftarrow$ getPrevTrPoint($j, D[]$)
12:         if FindMatch($D[], i, k, \epsilon_{sp}, \epsilon_t$) = False then
13:            output($(D[i], D[k])$, False)
14:         $k \leftarrow$ getPrevTrPoint($i, D[]$)
15:         if FindMatch($D[], j, k, \epsilon_{sp}, \epsilon_t$)= False then
16:            if $D[j].partFlag$=True then
17:               output($(D[j], D[k])$, False)
18:    if there is no "match" for $D[i]$ then
19:       $BP[] \leftarrow D[i]$
```

By examining only the previous point of a JP in a trajectory, we might not examine a possible temporary adjacent point that might lie after the last JP of a trajectory in each partition. For this reason, we post-process the last JPs in order to check for candidate $sNJP$s by invoking the *TreatLastTrPoints* function (Algorithm 6.1 line 6).

Example 6.1. *As illustrated in Figure 6.3(a), we suppose that the current point inserted into D is q_2. In Figure 6.3(b), assuming that $DistS(q_2, r_2) \leq \epsilon_{sp}$, we get a "match" and pair $((q_2, r_2), True)$ is reported (the symmetric pairs are omitted for simplicity). Subsequently, we need to find the previous point of r and in order to achieve this we should traverse our data backwards until we find it, as presented in Figure 6.3(c). When we find r_1, we need to check whether it is a NJP for each point $\in q$, as illustrated in Figure 6.3(c). If there exists a point $\in q$ that "matches", in our case q_1, nothing is reported and we proceed to examine whether q_2 and p_2 are JPs. If $DistS(q_2, p_2) \leq \epsilon_{sp}$ then we output the pair $((q_2, p_2), True)$, as shown in Figure 6.3(d). Subsequently, we need to find p_1 and check whether it is NJP for each point $\in q$. As depicted in Figure 6.3(e) there is no "match" between p_1 and any of the points of q. For this reason, we report the pair $((q_2, p_1), False)$. The same procedure is continued to the next point inserted to memory as delineated in*

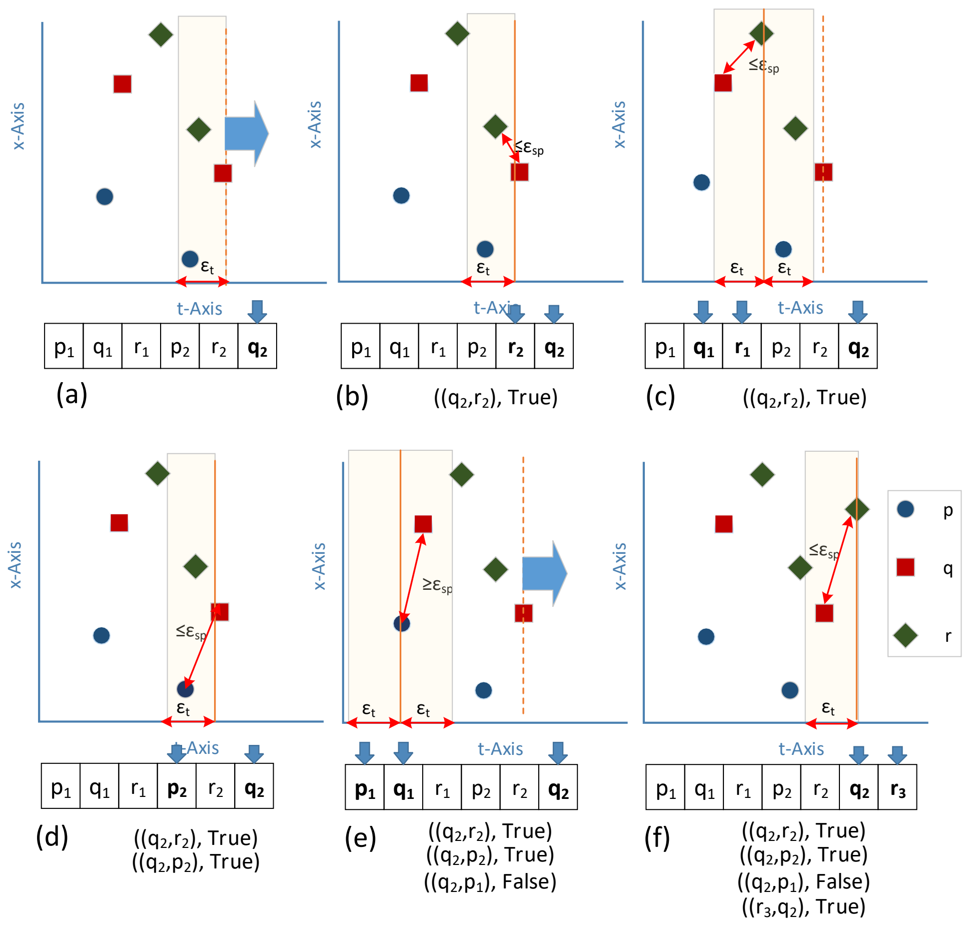

x-Axis
t-Axis
ε_t
$\leq\varepsilon_{sp}$
$\geq\varepsilon_{sp}$
p1 q1 r1 p2 r2 q2 r3
(a)
(b) ((q2,r2), True)
(c) ((q2,r2), True)
(d) ((q2,r2), True) ((q2,p2), True)
(e) ((q2,r2), True) ((q2,p2), True) ((q2,p1), False)
(f) ((q2,r2), True) ((q2,p2), True) ((q2,p1), False) ((r3,q2), True)
p
q
r

Figure 6.3: Join phase - The TRJPlaneSweep algorithm.

Figure 6.3(f) until there are no more points inserted.

The complexity of the *Join* procedure is $O(|D| \cdot a \cdot ((1-b) \cdot |D| + b \cdot |D| \cdot (2 \cdot (L + 2 \cdot a \cdot |D|))))$, where $|D|$ is the number of points, a is the selectivity of ϵ_t and b is the selectivity of ϵ_{sp}. L is the number of points that have to be traversed in order to find the previous point of a specific trajectory. It is obvious that when a tends to reach 1 the complexity tends to reach $O(|D|^2)$. In the worst case, the complexity can be analogous to $O(|D|^3)$, when both a and b tend to reach 1. However, for a typical analysis task ϵ_t and ϵ_{sp} are much smaller than the dataset duration and the dataset diameter respectively. Roughly, we can say that the complexity is $O(a \cdot b \cdot |D|^2)$.

Refine Phase

The output of the *Join* phase is actually pairs of points. From now on, let us refer to the left point of such a pair as *reference point* and the trajectory that it belongs to, *reference trajectory*. The *Refine* phase consists of a second MR job that reads the output of the *Join* step and groups points by the *reference trajectory*. Each *Reduce* task receives all pairs of points belonging to a specific trajectory, sorted first by the *reference point's* time and the by the *non-reference trajectory* ID. Figure 6.4 shows an example where the output pairs of points from the *Join* step are grouped, sorted and fed as input to three *Reduce* tasks (for trajectories p, q, and r respectively). The general idea here is to scan the set of JP in a sliding window fashion so as to identify "maximally" matching subtrajectories and at the same time "consult" the sets of BP and $sNJP$ in order to avoid false identifications, as described in Section 6.2.2.

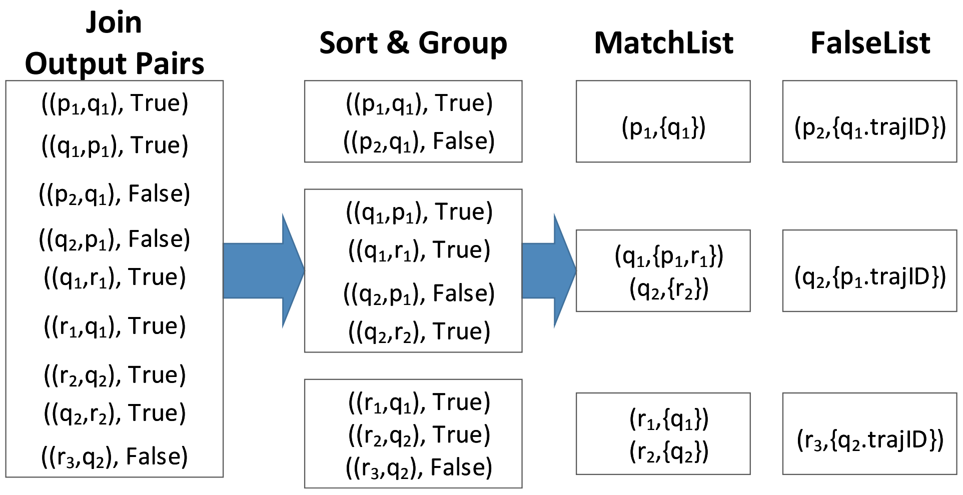


Figure 6.4: Output of *Join* and input of *Refine* phase.

Hence, each *Reducer* accesses all the pairs of a *reference trajectory* (say p) sorted by time, i.e., $\{p_1, p_2, \ldots, p_n\}$. Algorithm 6.3 describes the pseudo-code of the *Refine* phase which aims to identify all the "maximally matching" pairs of subtrajectories of p with other subtrajectories of any trajectory x ($x \neq p$). For each accessed pair $((p_i, x_j)$,*flag*), the algorithm assigns it in one of the two structures that it maintains: the *MatchList* and the *FalseList*. All JP and BP will be kept in the *MatchList*, whereas the candidate $sNJP$ is kept in the *FalseList* (lines 10–13). Again, this is more clearly depicted in the example of Figure 6.4. Also, notice that for each *reference point* in the *MatchList*, we maintain a list of points sorted by trajectory ID.

Algorithm 6.3 Refine(δt, ϵ_t)

```
1: Input: Pairs of points $((p_i, x_j),flag)$ for a given trajectory $p$, sorted by time
2: Output: Result of Distributed Subtrajectory Join for $p$
3: for each pair of points $((p_i, x_j),flag)$ do
4:   if ($p_i$ is encountered for the first time) then
5:     if DistT(MatchList.lastEntry, MatchList.firstEntry) $\geq \delta t$ then
6:       resultT $\leftarrow$ intersect lists in MatchList and exclude FalseList
7:       resultF $\leftarrow$ apply sliding window of $\delta t$ to resultT
8:       resultFinal $\leftarrow$ resultFinal $\bigcup$ resultF
9:       remove MatchList.firstEntry
10:  if (flag = True) then
11:    addToMatchList($p_i, x_j$)
12:  else
13:    addToFalseList($p_i, x_j$)
14: output(resultFinal)
```

Lemma 6.4. *The set of candidate $sNJP$ is sufficient so as to identify the set of $sNJP$ at the* Refine *phase.*

Proof. Consider two pairs of JP, (r_i, s_j) and (r_{i+1}, s_{j+n}), where $r_i, r_{i+1} \in r$ and $s_j, s_{j+n} \in s$. Moreover, let us assume that for each point s_m where $m \in [j+1, j+n-1]$, s_m is a *non-joining point* for each point of r. Then, points r_i, r_{i+1}, s_j, s_m and s_{j+n} will span at most to two consecutive temporal partitions: $part_k$ and $part_{k+1}$. This means that either $s_j, s_m \in part_k$ or $s_m, s_{j+n} \in part_{k+1}$. In both cases s_m will be recognized as a *non-joining point* for each point of r, by the procedure that generates the set of $sNJP$(Definition 6.7). □

The algorithm proceeds as follows: as soon as all pairs of points of a specific *reference point* p_i have been accessed, it initiates processing on the *MatchList*. The processing takes place only if the first and last point of p in *MatchList* have temporal distance greater than or equal to δt (line 5). The processing essentially identifies points of other trajectories that join with points of p in the whole temporal window. This is performed by intersecting the lists in *MatchList* and excluding points existing in the *FalseList* (line 6). List intersection is efficiently performed in linear time to the length of the lists, since the lists are sorted by trajectory ID. Figure 6.5 depicts the result of this processing as *resultT*.

Subsequently, the points in *resultT* are processed as follows. We start from the first point and take into consideration all points with temporal distance at most $\delta t - 2\epsilon_t$ from the first point. From this set of points, we derive the subtrajectories that "match" for the entire $\delta t - 2\epsilon_t$ window, and insert them in *resultF* (line 7). The temporary results of the *resultF* structure are added to the final result structure *resultFinal*, if not already contained in it (line 8). Then, a new set of points is considered, of temporal distance at most $\delta t - 2\epsilon_t$ from the second point of *resultT* and the process is repeated, similarly to a sliding a window of duration $\delta t - 2\epsilon_t$ on *resultT*. In the end, the first entry of the *MatchList* $(p_1, \{q_1, r_1, s_1\})$ is removed (line 9), as all potential results containing p_1 have already been produced. The algorithm terminates when the entire trajectory is traversed, the *resultFinal* is returned and each element of this list is emitted.

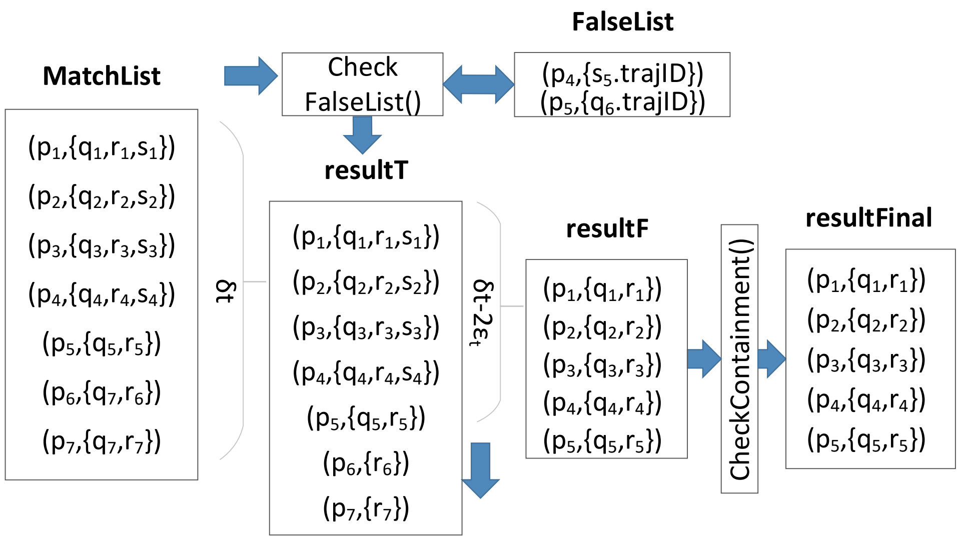


Figure 6.5: Refine procedure.

Example 6.2. *Figure 6.5 presents a working example of the Refine algorithm given the specific* MatchList *and* FalseList *of trajectory p.Assuming that* $DistT(p_1.t, p_7.t) \geq \delta t$, *we intersect all the lists contained in the specific window of the* MatchList *and we pass the result to* resultT. *In this way, the list of the last entry of* resultT *will contain only the points that belong to the subtrajectories that move "close" enough with p for the whole* δt *window. During list intersection, we take into account the* FalseList *structure in order to deal with points that belong to* $sNJP$. *Specifically, even though for each* p_i, *with* $i \in [1,7]$ $\exists$ *a "match" with q, however* q_6 *has no "match" with p, as depicted in the* FalseList. *For this reason, q should be excluded from* resultT *after* p_5. *Then, a sliding* $\delta t - 2\epsilon_t$ *window is created that traverses* resultT, *and* *for each such window we intersect all lists and the result is stored in* resultF*. For the first $\delta t - 2\epsilon_t$ window, as depicted in Figure 6.5, subtrajectories $r_{1,5}$ and $q_{1,5}$ are identified. The reason for this is to discover the subtrajectories that move "close" enough, with p for the whole $\delta t - 2\epsilon_t$ window. Subsequently, before proceeding to the next δt window, the contents of* resultF *are inserted to the final result, if not already contained.*

The complexity of the *Refine* procedure is $O(T \cdot SW \cdot dt \cdot l)$, where T is the average number of points in a trajectory, SW is the number of points contained in the δt window, dt is the number of points contained in the $\delta t - 2\epsilon_t$ window and l is the size of the list. The complexity, here, clearly depends on the average number of points per trajectory, the ϵ_t and δt parameter, and the number of pairs emitted by the join phase which in turn depends on ϵ_t and ϵ_{sp}.

6.4 Subtrajectory Join with Repartitioning

Even though the *DTJb* algorithm provides a correct solution to the *Distributed Subtrajectory Join* problem, it has some limitations. In particular, it does not address the *load balancing* challenge, since it does not handle the case of temporally skewed data. Also, due to the two chained MR jobs, the intermediate output of the first job is written to HDFS and must be read again by the second job, which imposes a significant overhead as its size is comparable and can be even bigger than the original dataset.

Motivated by these limitations, we propose an improved two-step algorithm (*DTJr*), which consists of the *repartitioning* and the *query* step. Each step is implemented as a MR job. However, the repartitioning step is considered a preprocessing step, since it is performed once and is independent of the actual parameters of our problem, namely ϵ_{sp}, ϵ_t, and δt.

6.4.1 Repartitioning

The aim of the repartitioning step is to split the input dataset in M equi-sized, temporally-sorted partitions (files), which are going to be used as input for the join algorithm. This is essential for two reasons: (a) it will provide the basis for load balancing, by addressing the issue of temporal skewness in the input data, and (b) it will result in temporal collocation of data, thus drastically reducing processing and network communication costs.

The repartitioning step is performed by means of a MR job as follows. We sample the input data, using Hadoop's *InputSampler*, and construct an equi-depth histogram on the temporal dimension. The histogram contains M equi-sized bins, i.e. the numbers of points in any two bins are equal, where the borders of each bin correspond to a temporal interval $[t_i, t_j)$.

The equi-depth histogram is exploited by the *Map* phase in order to assign each incoming data object in the corresponding histogram bin, based on the value of its temporal dimension. Each "map" function outputs each data object using as key a value $[1, M]$ that corresponds to the bin that the object belongs to. During shuffling, all data objects that belong to a specific bin are going to be sorted in time and will be collected by a single "reduce" function (thus having M "reduce" functions). As a result, each "reduce" function writes an output file to *HDFS* that contains all data objects in a specific temporal interval $[t_i, t_j)$ sorted by increasing time. A graphical view of the MR job is provided in Figure 6.6(a).

A subtle issue is how to determine the number M of bins (and, consequently, output files). A small value of M, smaller than the number of nodes in the cluster, would be opposed to the collocation property because data would have to be transferred through the network. On the other hand, a large value of M would result to many small files, smaller than the *HDFS* block size, and would lead to inefficient use of resources as well as increasing the management cost of these *HDFS* files. A good compromise is to have files of equal size to the *HDFS* block. Hence, the number of files can be calculated as $M = \lceil \frac{InputTotalSize}{hdfsblocksize} \rceil$. Collocation can be further improved by extending the *BlockPlacementPolicy* interface and forcing temporally adjacent files to be written to the same nodes.

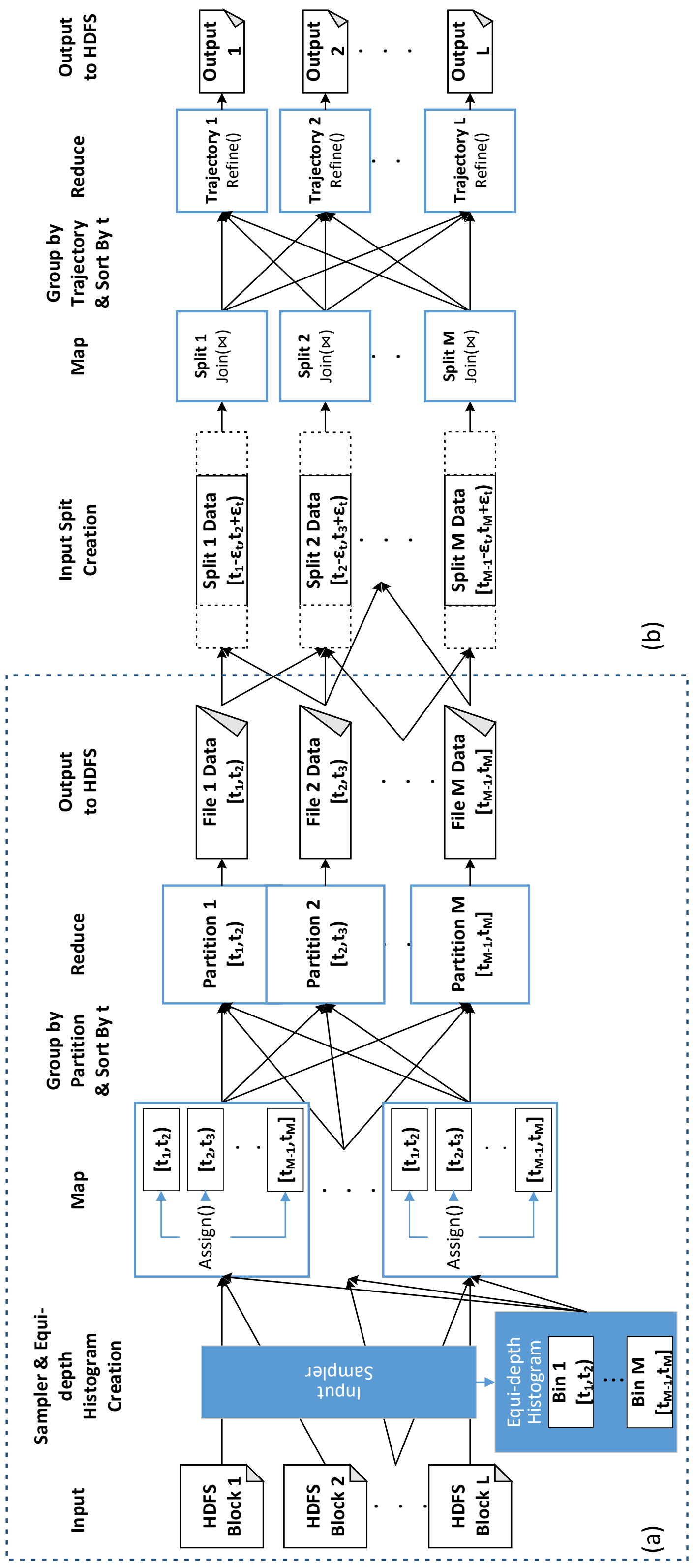


Figure 6.6: The DTJr algorithm in MapReduce: (a) Repartitioning step and (b) Query step.

6.4.2 The DTJr Algorithm

In order to minimize the I/O cost, the MR job that implements the proposed algorithm performs the *Join* procedure in the *Map* phase, and the *Refine* in the *Reduce* phase. To achieve this, we need to provide to a *Map* task as input, a data partition that contains all necessary data in order to perform part of the *Join* procedure *independently* from other *Map* tasks. Thus, an *HDFS* block produced by the repartitioning phase is expanded with additional points that exist at time (+/-)ϵ_t, and this is the process of *InputSplits* creation. In this way, points are duplicated to other *HDFS* blocks, which means that the same point may be output by two different *Map* tasks. To avoid this pitfall, a different *duplicate avoidance mechanism* is introduced which practically determines that a point is going to be output only by a single *Map* task; the *Map* task processing the *HDFS* block where the point belongs to.

As already mentioned, each data partition (*InputSplit*) that is fed to a *Map* task should contain all the data needed to perform the join of points for the specific partition, i.e. data for the period $[t_s^{part} - \epsilon_t, t_s^{part} + \epsilon_t]$. However, an output file produced by the repartitioning step is not sufficient due to the temporal tolerance ϵ_t, thus we need to augment these output files with extra data points, so that they form independent data partitions. At technical level, we devised and implemented a new *FileInputFormat* called *BloatFileInputFormat*, along with the corresponding *FileSplitter* and *RecordReader*, which selectively combines different files in order to create splits that carry all the necessary data points. Furthermore, during the creation of input splits we augment (as metadata) each split with the starting and ending time of the original partition of each split, termed t_s^{base} and t_e^{base}. The utility is to provide us with a simple way to perform duplicate avoidance at the *Join* phase.

Figure 6.6(b) shows that each *Map* task takes as input a split and performs the join at the level of point for a specific data partition. The input of this phase is a set of tuples of the form $\langle t, x, y, trajID \rangle$ sorted in ascending time t order. Since the data are already sorted with respect to the temporal dimension, we can apply the *Join* procedure, presented in Section 6.3.2. The output of the *Map* phase will be the JP, BP and $sNJP$ sets. Finally, the *Refine* procedure presented in Section 6.3.2 can be performed at the *Reduce* phase.

6.5 Index-based Subtrajectory Join with Repartitioning

The *Join* step of the previous algorithms is common and operates on the array D that contains temporally sorted points. However, it can be improved in two ways. First, by employing spatial filtering in order to avoid attempting to join points that are far away. Second, by having an index structure that given a point p_i can efficiently locate the (temporally) previous point p_{i-1} of p. Motivated by these observations, we devised and implemented an indexing scheme in order to speed up the processing of the join.

6.5.1 Indexing Scheme

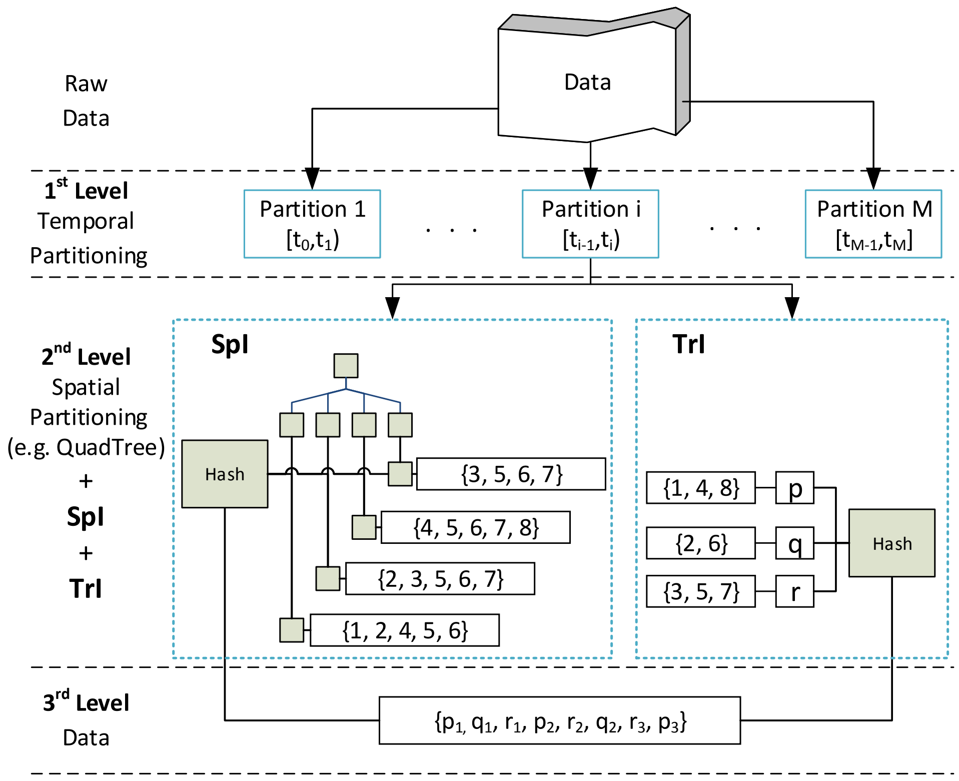


Figure 6.7: Indexing Scheme of *DTJi* algorithm

As illustrated in Figure 6.7, this scheme consists of 3 levels. We already covered the first level in Section 6.4.1, where the initial data are partitioned to equi-sized temporal partitions (Section 6.4). At the second level, we partition the space. In order to have load balanced partitions we utilize the spatial partitioning provided by QuadTrees. More specifically, an “empty” QuadTree is created once, by sampling the original data, as in [26], and is written to *HDFS*. It is important to mention here that the QuadTree contains only the spatial partitions and not the actual points. Then, when a new query is posed, the QuadTree is loaded into Hadoop’s *distributed cache* in order to be accessible by all the nodes. Moreover, at the same level, we employ two indexes. The first index is a spatial index (SpI) which enables pruning of points based on their spatial distance, thus decreasing significantly the number of points that need to be examined within the ϵ_t window. The second index is an index that keeps track of the representation of each individual trajectory within the temporally sorted structure D (TrI), thus providing an efficient way to access the previous trajectory point. The two indexes are created gradually, as the data are read from HDFS. Finally, at the third level, we have the temporally sorted data that correspond to the specific temporal partition.

Spatial Index (SpI)

The spatial index, called SpI, utilizes a given space partitioning, in our case QuadTrees. For each spatial partition of the QuadTree, SpI keeps a temporally sorted array where each entry is the position of a point that is contained in the given partition expanded by ϵ_{sp}. SpI is implemented as a HashMap with key the partition id and value the sorted array. Thus, a partition can be accessed in O(1), while a point in a partition can be accessed in $O(logP_i)$, where P_i here is the number of points in the corresponding sorted array. The construction of SpI has $O(|D| \cdot h)$ complexity, where $|D|$ is the number of points in the specific temporal partition and h is the height of the QuadTree, since for each point we need to traverse the QuadTree in order to find out in which expanded partition it is contained. Note that each point is enriched with the id of its original (i.e. not expanded) spatial partition, thus consisting of $\langle trajID, x, y, t, PartitionID \rangle$.

Trajectory Index (TrI)

The TrI index keeps track of each individual trajectory within D. TrI is also implemented as a HashMap with key the trajectory id. For each trajectory, the value is a temporally sorted array, where each entry corresponds to a point of a trajectory, and the value of the entry is an integer indicating the point’s position in D. Thus, a trajectory point can be efficiently accessed in

Algorithm 6.4 Join^I (Split, ϵ_{sp}, ϵ_t, t_s^{base}, t_e^{base})

```
1: Input: A split, $\epsilon_{sp}$, $\epsilon_t$, $t_s^{base}$, $t_e^{base}$
2: Output: All pairs of $JP$, $BP$ and candidate $sNJP$
3: $QT \leftarrow LoadQuadTree()$
4: for each $point\ i \in Split$ do
5:    if $point.t \in [t_s^{base} - \epsilon_t, t_e^{base} + \epsilon_t]$ then
6:       $D[i], TrI, SpI \leftarrow point$
7:    TRJPlaneSweep$^I$($D[], TrI, SpI, \epsilon_{sp}, \epsilon_t, t_s^{base}, t_e^{base}$)
8: TreatLastTrPoints()
9: for each $point\ j \in BP[]$ do
10:    output(($BP[j], null$), True)
```

$O(logT)$, where T is the number of points of a trajectory. To exemplify, the first element of the array holds the position of the first point of the trajectory inside D and so on. The construction of this index has $O(T)$ time complexity since the data is already sorted in time.

6.5.2 The DTJi Algorithm

Having these two indexes at hand we can utilize them in order to perform the join operation in an efficient way. Algorithm 6.4, presents the index-enhanced plane sweep procedure. Initially, the QuadTree is loaded into memory from the *distributed cache* (line 3) and then, each accessed point is inserted not only to an array D, which contains points sorted in increasing time, but also to the SpI and TrI indexes. Finally, the TRJPlaneSweep^I() algorithm is invoked for each accessed point (lines 4–7).

Algorithm 6.5, presents the TRJPlaneSweep^I() algorithm. Here, given a point $p_i \in p$, instead of scanning the whole ϵ_t window before it, in order to find "matches", we perform a search in SpI and get only the points that belong to the same partition as p_i by invoking the getCandidatePoint() method (line 4). The partition id is retrieved in $O(1)$ and then binary search is performed in the temporally sorted list of points in order to find the position of p_i inside it. Having that, we can get the previous element, which will be the previous point in time that lies within the same partition, and check if the temporal and spatial constraint are satisfied. If they are satisfied, we have a "match", we proceed to the previous element of SpI and so on and so forth. Assuming that we have a "match" with q_j that belongs to trajectory q we need to find the previous point of q. This is achieved by

Algorithm 6.5 TRJPlaneSweep^I($D[], TrI, SpI, \epsilon_{sp}, \epsilon_t, t_s^{base}, t_e^{base}$)

```
1: Input: $D[]$, $\epsilon_{sp}$, $\epsilon_t$, $t_s^{base}$, $t_e^{base}$
2: Output: All pairs of $JP$, $BP$ and candidate $sNJP$
3: if DuplCheck($D[i].t, t_s^{base}, t_e^{base}$)=True then
4:    for each element $D[j]$ returned by getCandidatePoint($i, SpI, D[]$) do
5:       if DistS($D[i], D[j]$) $\leq \epsilon_{sp}$ then
6:          output(($D[i], D[j]$), True)
7:          remove $D[i]$ from $BP[]$
8:          if DuplCheck($D[j].t, t_s^{base}, t_e^{base}$)=True then
9:             output(($D[j], D[i]$), True)
10:            remove $D[j]$ from $BP[]$
11:         $k \leftarrow$ getPrevTrPoint$^I$($j, D[], TrI$);
12:         if FindMatch$^I$($D[], i, k, \epsilon_{sp}, \epsilon_t, TrI$)= False then
13:            output(($D[i], D[k]$), False)
14:         $k \leftarrow$ getPrevTrPoint$^I$($i, D[], TrI$);
15:         if FindMatch$^I$($D[], j, k, \epsilon_{sp}, \epsilon_t, TrI$) = False then
16:            if DuplCheck($D[j].t, t_s^{base}, t_e^{base}$)=True then
17:               output(($D[j], D[k]$), False)
18:   if there is no "match" for $D[i]$ then
19:      $BP[] \leftarrow D[i]$
```

invoking getPrevTrPoint^I, which performs a search in TrI in order to retrieve in O(1) the entry of q (lines 11, 14). Then, by performing binary search in the temporally sorted list, we can find the position of q_j and can easily get q_{j-1}. Having that, we need to find if it "matches" with any point that belongs to p. Here, instead of scanning the whole $2\epsilon_t$ window of q_{j-1} in order to check for "matches" with p, we perform a search in TrI in order to get the points of p that exist "close" to the time of q_{j-1} (lines 12, 15). Then, if the spatial and temporal constraints are satisfied we have a "match" and the *FindMatch^I ()* method returns True. Otherwise, the whole procedure continues, until the temporal constraint is not satisfied anymore.

Example 6.3. *Following the example of Figure 6.3(a)-(f), in Figure 6.8 we can see how the two indexes are utilized in order to perform the TRJ-PlaneSweep operation. More specifically, in Figure 6.8(a), in order to find the point that "matches" with q_2, we do not scan the entire ϵ_t window, instead we utilize the SpI index in order to find the candidate "matches". Subsequently, in order to find the previous point of r_2 in r, as shown in Figure 6.8(b), we employ the TrI index. Finally, so as to find if r_1 "matches" with any of the points of q, we make use of the TrI index again, as depicted in Figures 6.8(c) and (d). This time we use $q_2.trajID$ and $r_1.t$ in order to find the points, if* *any, of q that exist "close" to the time of r_1.*

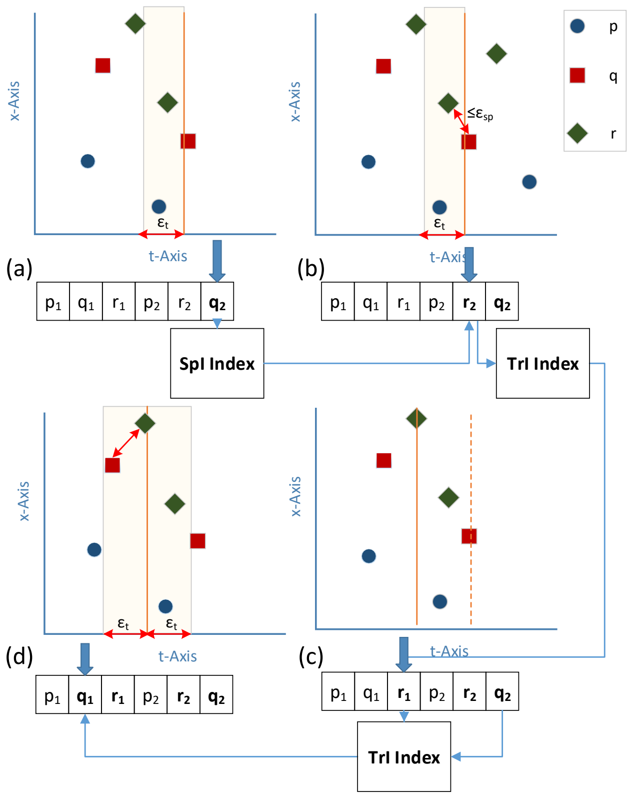


Figure 6.8: Example of TRJPlaneSweep^I.

The complexity of the index-based solution is $O(|D| \cdot h \cdot (log_2 P_i \cdot a \cdot P_i((1 - b) \cdot P_i + b \cdot P_i \cdot (2 \cdot (log_2 T + (log_2 T + a \cdot T)))))))$, with $|D|$ being the number of points, h the height of the QuadTree, a and b the selectivity of ϵ_t and ϵ_{sp} respectively. P_i is the number of points within the i-th partition expanded by ϵ_{sp}, where $P_i \ll |D|$, and T is the number points per trajectory. In the worst case, where a and b tend to 1, the complexity can reach $O(|D| \cdot log_2 P_i \cdot P_i^2)$. However, again this only occurs for values of ϵ_t and ϵ_{sp} that are comparable to the dataset's duration and diameter respectively. Roughly speaking, the complexity drops to $O(|D| \cdot (log_2 P_i \cdot a \cdot b \cdot P_i^2))$, which clearly shows the benefit attained when employing the proposed indexing scheme.

6.6 Experimental Study

In this section, we provide our experimental study on the comparative performance of the three variations of our solution, namely (1) *DTJb* that uses two MR jobs (Section 6.3), (2) *DTJr* that employs repartitioning and a single job to perform the join (Section 6.4), and (3) *DTJi* that additionally uses the SpI and TrI indexes for more efficient join processing (Section 6.5). Furthermore, we compare our solution with the work presented in [105].

The experiments were conducted in a 49 node Hadoop 2.7.2 cluster, provided by *okeanos*[2], an IAAS service for the Greek Research and Academic Community. The master node consists of 8 CPU cores, 8 GB of RAM and 60 GB of HDD while each slave node is comprised of 4 CPU cores, 4 GB of RAM and 60 GB of HDD. Our configuration enables each slave node to launch 4 containers, thus resulting that at a given time the cluster can run up to 192 jobs (*Map* or *Reduce*). The real dataset employed for our experiments is IMIS, as described in Section 1.5.

Our experimental methodology is as follows: Initially, we verify the scalability of our algorithms by varying (a) the dataset size, and (b) the number of cluster nodes (Section 6.6.1). Then, we examine the benefits of the repartitioning step as well as the associated cost (Section 6.6.2). Successively, we compare our solution with the work presented in [105] (Section 6.6.3). Subsequently, we perform a sensitivity analysis in order to evaluate the effect of different parameters to our algorithms (Section 6.6.4). Finally, we perform a set of experiments so as to examine the creation time and the size of the proposed indexes with respect to to varying the number of spatial partitions and ϵ_{sp} (Section 6.6.5).

Table 6.3 shows the experimental setting, where we vary the following parameters: ϵ_t, ϵ_{sp}, δt, the maximum number of points per cell, and the number of cluster nodes, which are the main parameters affecting the performance of our algorithms.

Table 6.3: Parameters and default values (in bold) used in the experimental study of Chapter 6

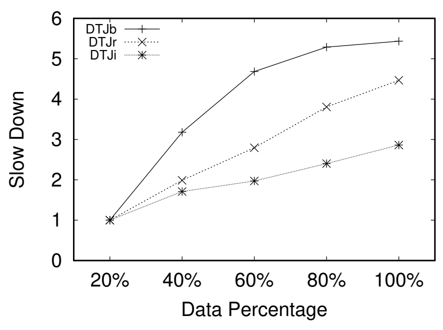
Parameter	**Values**
ϵ_t (%)	100%, 150%, **200%**, 250%, 300%
ϵ_{sp} (%)	10%, 20%, **30%**, 40%, 50%
δt (in minutes)	10, 15, **20**, 25, 30
max # of points per cell (%)	1%, 2%, **3%**, 4%, 5%
# of Nodes	12, 24, 36, **48**

In more detail, the values of ϵ_t were calculated as a percentage of the average

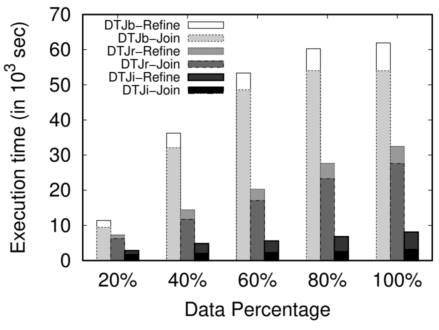
[2]https://okeanos.grnet.gr/home/

duration between two consecutive trajectory samples (≈ 600 sec) and ϵ_{sp} was calculated as a percentage of the diameter of the smallest cell produced by the QuadTree. Parameter δt depends on the application scenario. For example, when trying to identify transshipment behavior, where two vessels might illegally exchange goods, 20 minutes is too small. The application scenario that we had in mind when setting this parameter was the clustering scenario, where the goal is to identify groups of objects that moved together for at least some duration, so 20 minutes seemed appropriate. In fact, as it was expected, setting different values to δt, as illustrated in Figure 6.12, does not affect significantly the execution time of our solution, since it affects only a small part of the refine procedure. Finally, the maximum number of points per cell is calculated as a percentage over the total population.

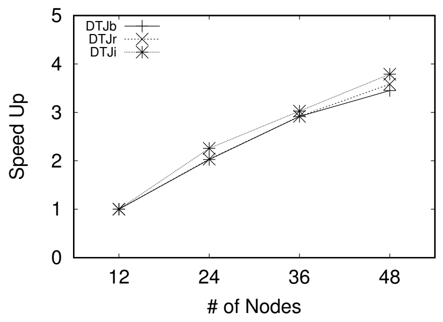
6.6.1 Scalability



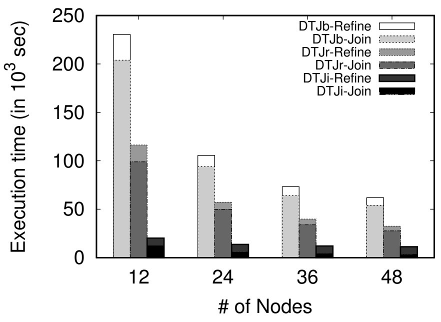
(a)



(b)



(c)



(d)

Figure 6.9: Scalability analysis varying (a),(b) the size of the dataset and (c),(d) the number of nodes.

Initially, we vary the size of our dataset and measure the execution time of our algorithms. To study the effect of dataset size, we created 4 portions

(20%, 40%, 60%, 80%) of the original dataset. As the dataset size increases and the number of nodes remains the same, it is expected that the execution time will increase. In order to measure this, for each portion D_i of the dataset with $i \in [1, 5]$, we calculate $SlowDown = \frac{T_{D_i}}{T_{D_1}}$, where T_{D_1} is the execution time of the first portion (i.e. 20%) and T_{D_i} the execution time of the current one. As shown in Figure 6.9(a), as the size of the dataset increases, the *DTJr* linear and the *DTJi* appears to have linear behaviour, with *DTJi* presenting better scalability. On the other hand, *DTJb* appears to have a somehow "abnormal" behaviour. This can be justified if we study Figure 6.10, which presents the standard deviation of the different portions of the dataset. In fact, we can observe that *DTJb* in Figure 6.9(a) is affected by how imbalanced is the partitioning in each portion of the dataset, as depicted in *DTJb* Figure 6.10. To investigate further the performance of the different algorithms, we measure separately the execution time of the *Join* and *Refine* phases for all algorithms. Concerning the *Join* phase, as illustrated in Figure 6.9(b), *DTJi* outperforms *DTJb* by 16× and the *DTJr* by almost one order of magnitude. Regarding the *Refine* phase, as depicted in Figure 6.9(b), *DTJr* and *DTJi*, perform exactly the same, as anticipated, since they use an identical algorithm. Instead, *DTJb* performs worse due to the fact that the *Refine* phase is implemented as a second MR job, which means that the output of the *Join* phase, which is typically several times larger than the input data, needs to be read from *HDFS* and get sorted, grouped and shuffled to the *Reduce* tasks.

Subsequently, we keep the size of the dataset fixed (at 100%) and vary the number of nodes. As the number of nodes increase and the dataset size remains the same, it is expected that the execution time will decrease. In order to measure this, for each portion N_i of the dataset with $i \in [1, 5]$, we calculate $SpeedUp = \frac{T_{N_i}}{T_{N_1}}$, where T_{N_1} is the execution time when using the minimum number of nodes (i.e. 12) and T_{N_i} the execution time of the current one. In this experiment, as illustrated in Figure 6.9(c) and (d), we observe that all three approaches present linear scaling, with *DTJi* demonstrating slightly better scalability. The reason why the behaviour is different here, is that in this experiment the dataset that was employed was fixed (100%). This means that, despite the fact that we vary the number of nodes, the effect of the different algorithms over the data is the same. On the contrary, when we increase the amount of data, as already shown in Figure 6.9(b), we can see than the performance of *DTJb* is affected by the skewness of the different portion of the dataset that were used. As depicted in Figure 6.10, we can observe that the standard deviation of *DTJb* affect significantly *DTJb* in Figure 6.9(a) and (b).

6.6.2 Repartitioning and Load Balancing

In this set of experiments, we evaluate the cost of the repartitioning step employed by *DTJr* and *DTJi*. In Figure 6.10(a), we compare *DTJb* (which does not use this step) against *DTJr* and *DTJi* after including in the latter two algorithms the time needed for repartitioning. The result shows that even for a single query both algorithms outperform *DTJb*. Obviously, for multiple queries with different query parameters (ϵ_t, ϵ_{sp}, δt), the gain is multiplied, since the repartitioning cost needs to be paid only once, before processing the first query. This experiment justifies the use of the repartitioning step, while demonstrating its low overhead in the case of a single query, which in the case of multiple queries becomes negligible.

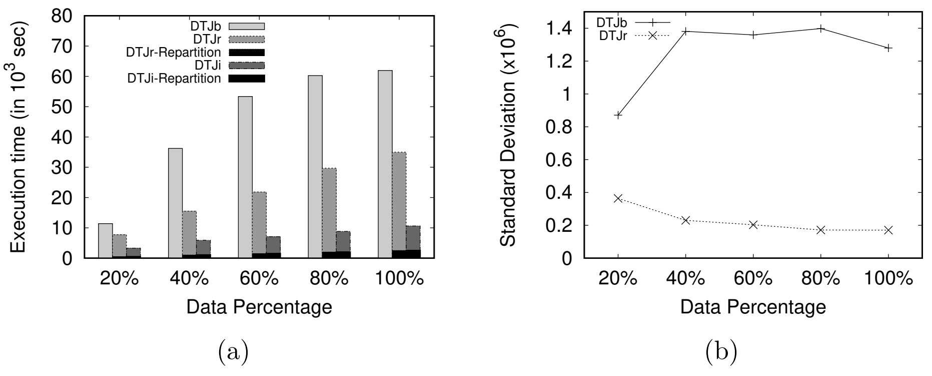


Figure 6.10: (a) Repartitioning cost and (b) Load balancing

In order to quantify whether the work allocation of the *Join* is balanced to the different parallel tasks, we compare the input size of the *Join* phase of *DTJb*, against the *Join* phase of *DTJr*. In Figure 6.10(b), we report the standard deviation of the size of input data for the various tasks. Smaller values of the standard deviation, indicate that the different tasks are assigned with similar-sized input data, thus the load is more fairly balanced. *DTJr* demonstrates significantly lower standard deviation, approximately one order of magnitude, than *DTJb*. This also partly justifies the overall better performance of *DTJr* illustrated in Figure 6.9(b) and Figure 6.9(d).

6.6.3 Comparative Evaluation

As already mentioned, the problem of *Distributed Subtrajectory Join* has not been addressed yet in the literature and it is not straightforward (if and) how state of the art solutions to trajectory similarity search and trajectory join can be adapted to solve the problem. However, if we utilize only a specific instance of our problem, when $\delta t = 0$, then we only need to identify the set of JP during the *Join* phase. Based on this observation, we select to compare with the work presented in [105], called $SJMR$, a state of the art MapReduce-based spatial join algorithm, which is able to identify efficiently the set of JP that will be passed to the *Refine* procedure and produce the desired result. The reason why $SJMR$ was chosen is that it is a generic solution which could form the basis for any distributed spatial join algorithm and thus required the minimum amount of modifications so as to match with our problem specification.

More specifically, $SJMR$ repartitions the data at the Map phase and Joins them at the Reduce phase by performing a plane sweep join. For the sake of comparison, we modified $SJMR$ by injecting time as a third dimension and introducing parameters ϵ_{sp} and ϵ_t. In more detail. at the Map phase the spatiotemporal space is divided to tiles using a fine grained grid. Then, each data point is expanded by ϵ_{sp} and ϵ_t and is assigned to the tiles with which it intersects. Subsequently, the tiles are mapped to partitions using the method described in [105]. At the Reduce phase, the points are grouped by partition and sorted by one of the dimensions (we chose the temporal dimension so as to be aligned with our solution). Finally, we sweep through the time dimension and report the set of JP. In addition, we implemented a modified version of $SJMR$, named $SJMRi$, that makes use of our quadtree index.

So, in this set of experiments we compare *DTJi*-Join, which outperforms *DTJb*-Join and *DTJr*-Join, with $SJMR$. In more detail, we vary the size of our dataset and measure the execution time of the three algorithms. The results, as illustrated in Figure 6.11 show that *DTJi*-Join not only performs significantly better than $SJMR$ but more importantly, the gain of *DTJi*-Join over $SJMR$ increases for larger data sets. The reason for this behaviour lies mainly due to the utilization of the indexing structure of *DTJi* ([105] uses no indexes) and the fact that *DTJi*-Join is a Map-only job where the repartitioning cost is “paid” only once (as a preprocessing step), unlike $SJMR$, where this cost is “paid” every time at the Map phase, as explained

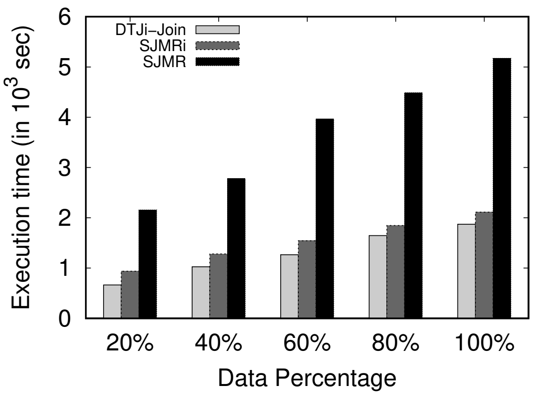


Figure 6.11: Comparative evaluation between *DTJi* and *SJMR*

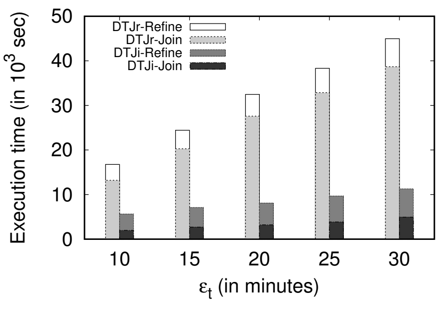
earlier.

On the other, the results show that *SJMRi* performs significantly better than *SJMR* but worse than *DTJi*-Join. The reason why *DTJi*-Join performs better *SJMRi* is mostly because *DTJi*-Join is a Map-only job while *SJMRi* is a Map-Reduce job, hence *DTJi*-Join avoids sorting, shuffling and network transfer cost between the Map and Reduce phase that *SJMRi* has to undergo.

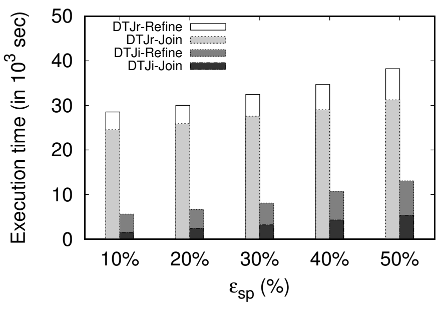
6.6.4 Sensitivity Analysis

In the following experiment, we perform a sensitivity analysis of algorithms *DTJr* and *DTJi*. We exclude *DTJb* from this set of experiments, as it consistently performs significantly worse than the other two algorithms.

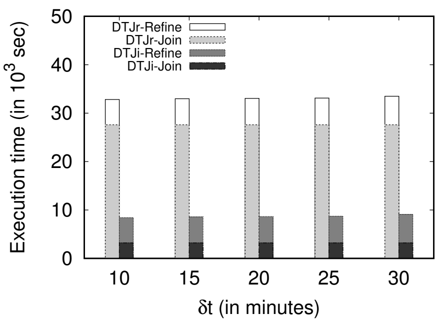
Initially we vary the value of ϵ_t while retaining fixed the values of the other parameters. As shown in Figure 6.12(a), the execution time of both algorithms, as expected, increases with ϵ_t. In more detail, the *Join* phase of *DTJr* is more sensitive to the fluctuation ϵ_t than the *Join* phase of *DTJi*, due to the fact that the latter is utilizing the SpI index which, for a given ϵ_t, performs spatial filtering instead of scanning the entire space in order to find "matching" pairs of points. What is more interesting is that as ϵ_t increases the difference between the two approaches increases, which means that for higher values of ϵ_t the difference, in terms of execution time, will be higher than one order of magnitude. As far as it concerns the *Refine* step, both approaches present the same increasing behavior when ϵ_t increases, since both of them employ the same algorithm, due to the fact that the higher the ϵ_t, the larger the sliding window that is created.



(a)



(b)



(c)

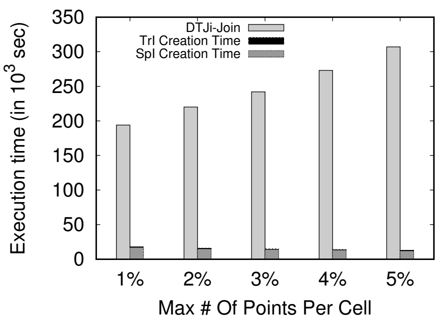
Figure 6.12: Sensitivity analysis varying (a) ϵ_t, (b) ϵ_{sp} and (c) δt

Then, we set different values to ϵ_{sp} while keeping the values of the other parameters fixed. As illustrated in Figure 6.12(b), ϵ_{sp} affects directly the *Join* and indirectly the *Refine* phase of both approaches. More specifically, the *Join* phase of *DTJr* is slightly affected by setting different values to ϵ_{sp} due to the fact that, for a given ϵ_t, *DTJr* will search the whole space in order to find "matches". Hence, ϵ_{sp} will only affect the number of "matches". On the other hand, *DTJi* does not search the whole space but utilizes the SpI index which, consequently, makes it more sensitive to ϵ_{sp}. The only case where the *Join* phase of *DTJr* performs the same as *DTJi* is when ϵ_{sp} spans the whole dataset space. Regarding the *Refine* step, as expected, both approaches perform the same and the higher the ϵ_{sp}, the higher the execution time. The reason for this behaviour is that as ϵ_{sp} increases, the product of the *Join* phase increases.

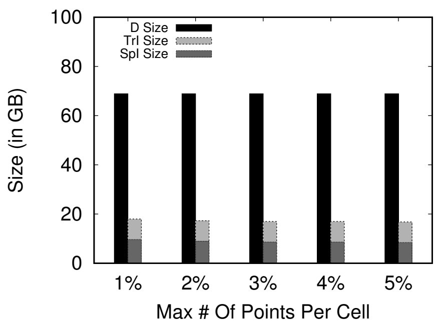
Finally, we vary the values of δt while keeping the values of the other parameters fixed. As presented in Figure 6.12(c), this parameter affects only the *Refine* phase, as anticipated. More specifically, the higher the δt the slightly higher the execution time of both approaches. This takes place due to the fact that as δt increases, the sliding window gets larger.

6.6.5 Indexing

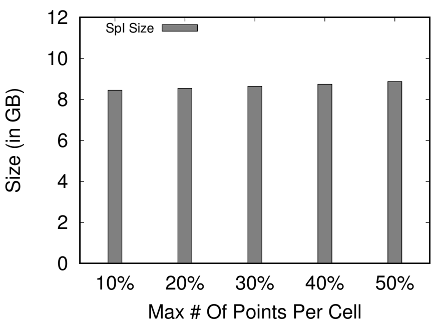
In order to measure the effect of having different number of spatial partitions in spatial index size and spatial index construction time, we perform a final set of experiments. More specifically, we vary the maximum number of points per cell parameter of the QuadTree, and we measure the index creation time and the index size. As illustrated in Figure 6.13(a), the SpI construction time increases as the maximum number of points per cell decrease, while the TrI construction time is, as expected, not affected by that. This occurs due to the fact that as the maximum number of points per cell decreases, the number of spatial partitions increases. Furthermore, as depicted in Figure 6.13(a), the fewer the maximum number of points per cell the smaller the execution time of the *Join* algorithm. It is worth mentioning that the overall index construction time as a percentage over the execution time of the *Join* algorithm varies only between 4% and 11%.



(a)



(b)



(c)

Figure 6.13: (a) Index construction time, (b) Index size and (c) Effect of ϵ_{sp}

As far as the size of the indexes is concerned, Figure 6.13(b) illustrates how the size is affected when varying the maximum number of points per cell. As expected, the TrI index is not affected, whereas the SpI index slightly increases its size as the number of partitions increase. At this point, we should mention that compared to the size of D, the percentage of the total size of the indexing scheme over D varies only between 24% and 28%.

Another parameter that can affect the SpI index is ϵ_{sp} due to the fact that each spatial partition is enlarged by ϵ_{sp}. As depicted in Figure 6.13(c), the size of SpI increases as ϵ_{sp} increases.

6.7 Summary

In this chapter, we introduced the *Distributed Subtrajectory Join* query, an important operation in the spatiotemporal data management domain, where very large datasets of moving object trajectories are processed for analytic purposes. To address this problem in an scalable manner following the MapReduce programming model, we initially provided a well-designed basic solution which is used as a baseline in order to propose two efficient improvements, called *DTJr* and *DTJi* which can boost the performance by up to 16× and 10×, respectively. Our experimental study was performed on a very large real dataset of trajectories from the maritime domain,consisting of 56 GB of data (or 1.5 billion time-stamped locations).

7 Scalable Distributed Subtrajectory Clustering

Having already tackled the problem of *Distributed Subtrajectory Join*, in this chapter, we address the problem of *Distributed Subtrajectory Clustering* in an efficient and highly scalable way. The problem is challenging because the subtrajectories to be clustered are not known in advance, but they need to be discovered dynamically based on adjacent subtrajectories in space and time. Towards this objective, we split the original problem to three sub-problems, namely *Subtrajectory Join*, *Trajectory Segmentation* and *Clustering and Outlier Detection*, and deal with each one in a distributed fashion by utilizing the MapReduce programming model. The efficiency and the effectiveness of our solution is demonstrated experimentally over a synthetic and two large real datasets from the maritime and urban domains and through comparison with two state of the art subtrajectory clustering algorithms. An earlier version of the content of this chapter appears in [91].

7.1 Introduction

Nowadays, the unprecedented rate of trajectory data generation, due to the proliferation of GPS-enabled devices, poses new challenges in terms of storing, querying, analyzing and extracting knowledge from big mobility data. One of these challenges is cluster analysis, which aims at identifying clusters of moving objects (thus, unveil hidden patterns of collective behavior), as well as detecting moving objects that demonstrate abnormal behaviour and can be considered as outliers.

The research so far has focused mainly in methods that aim to identify specific collective behavior patterns among moving objects, such as [48, 45, 44, 60, 51, 50, 92, 107, 28]. However, this kind of approaches operate at specific predefined temporal “snapshots” of the dataset, thus ignoring the route of each moving object between these sampled points. Another line of research, tries to identify patterns that are valid for the entire lifespan of the moving objects [56, 67, 21, 79]. However, discovering clusters of complete trajectories can overlook significant patterns that might exist only for some portions of their lifespan. The following motivating example shows the merits of subtrajectory clustering.

Example 7.1. (Subtrajectory clustering) *Figure 7.1(a) illustrates six trajectories moving in the xy-plane, where each one of them has a different origin-destination pair. More specifically, these pairs are $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow D$, $B \rightarrow A$, $B \rightarrow C$ and $B \rightarrow D$. These six trajectories have the same starting time and similar speed. A typical trajectory clustering technique would fail to identify any clusters. However, the goal of a subtrajectory clustering method is to identify 4 clusters ($A \rightarrow O$ (red), $B \rightarrow O$ (blue), $O \rightarrow C$ (purple), $O \rightarrow D$ (orange)) and 2 outliers ($O \rightarrow A$ and $O \rightarrow B$ (black)), as depicted in Figures 7.1(b).*

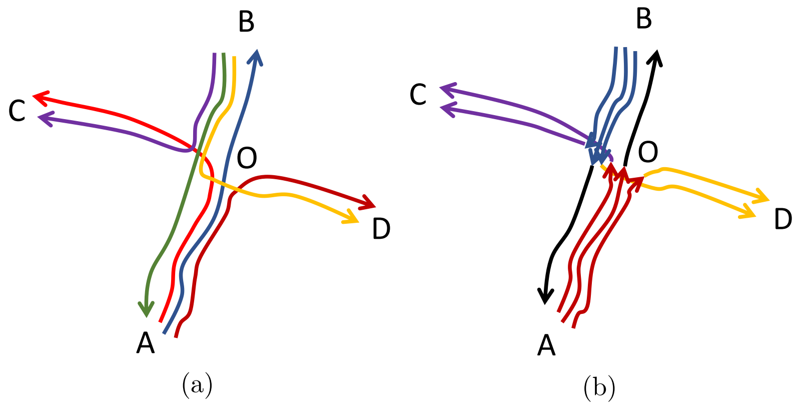


Figure 7.1: (a) Six trajectories moving in the xy-plane and (b) 4 clusters (red, blue, orange and purple) and 2 outliers (black).

The problem of subtrajectory clustering is shown to be NP-Hard (cf. [3]). In addition, the objects to be clustered are not known beforehand (as in entire-trajectory – from now on – clustering algorithms), but have to be identified through a trajectory segmentation procedure. Efforts that try to deal with this problem in a centralized way do exist. More specifically, an approach that segments the trajectories based on their geometric features, and then clusters them by ignoring the temporal dimension is presented in [49]. Instead, the authors in [70] take into account the temporal dimension, and the segmentation of a trajectory takes place whenever the density of its spatiotemporal “neighborhood” changes significantly. The segmentation phase is followed by a sampling phase, where the most representative subtrajectories are selected and finally the clusters are built “around” these representatives. A similar approach is adopted in [3], where the goal is to identify common portions between trajectories,with respect to some constraints and/or objectives, thus taking into account the “neighborhood” of each trajectory. These common subtrajectories are then clustered and each cluster is represented by a pathlet, which is a point sequence that is not necessarily a subsequence of an actual trajectory.

Unfortunately, applying centralized algorithms for subtrajectory clustering (e.g., [70, 49, 3]) over massive data in a scalable way is far from straightforward. This calls for parallel and distributed algorithms that address the scalability requirements. In this context, one challenge is how to partition the data in such a way so that each node can perform its computation independently, thus minimizing the communication cost between nodes, which is a cost that can turn out to be a serious bottleneck. Another challenge, related to partitioning, is how to achieve load balancing, in order to balance the load fairly between the different nodes. Yet another challenge is to minimize the iterations of data processing, which are typically required in clustering algorithms. Interestingly, there have been some recent efforts towards mining mobility data in a distributed way, such as mining co-movement patterns [28], identifying frequent patterns [79] or adapting already existing distributed solutions to trajectory data [21], yet no approach for distributed subtrajectory clustering exists as of now.

Motivated by these limitations, we study the *Distributed Subtrajectory Clustering (*DSC*)* problem, which has not been addressed yet in a scalable and efficient way. Moreover, salient features of our approach include: (a) the discovery of clusters of subtrajectories, instead of whole trajectories, (b) spatiotemporal clustering, instead of spatial only, and (c) support of trajectories with variable sampling rate, length and with temporal displacement.

Our main contributions are the following:

- We formally define the problem of *Distributed Subtrajectory Clustering*, investigate its properties and discuss the main challenges.
- We propose two neighborhood-aware trajectory segmentation algorithms, which are tailored to DSC problem, covering different application requirements.

- We design an efficient and scalable solution for the problem of *Distributed Subtrajectory Clustering*.
- We perform an extensive experimental study, where the performance and the effectiveness of the proposed algorithms is evaluated by using a synthetic and two large, real trajectory datasets from different domains (urban and maritime). The merits of our solution are demonstrated with respect to two state of the art subtrajectory clustering algorithms, [70] and [49].

The rest of the chapter is organized as follows. In Section 7.2 we introduce the *DSC* problem, in Section 7.3 we present our proposed solution and in Section 7.4 we perform a complexity analysis of the algorithms that constitute our solution. Then, in Section 7.5, we present the results of our experimental study. Finally, we conclude the chapter in Section 7.6.

7.2 Problem Formulation

Given a set D of moving object trajectories, a trajectory $r \in D$ is a sequence of timestamped locations $\{r_1, \dots, r_N\}$. Each $r_i = (loc_i, t_i)$ represents the i-th sampled point, $i \in 1, \dots, N$ of trajectory r, where N denotes the length of r (i.e. the number of points it consists of). Moreover, loc_i denotes the spatial location (2D or 3D) and t_i the time coordinate of point r_i, respectively. A subtrajectory $r_{i,j}$ is a sub-sequence $\{r_i, \dots, r_j\}$ of r which represents the movement of the object between t_i and t_j where $i < j$ and $i, j \in 1, \dots, N$. Let $d_s(r_i, s_j)$ denote the spatial distance between two points $r_i \in r$, $s_j \in s$. In our case we adopted the Euclidean distance, however, other metric distance functions might be applied. Also, let $d_t(r_i, s_j)$ denote the temporal distance, defined as $|r_i.t - s_j.t|$. Furthermore, let Δt_r symbolize the duration of trajectory r (similarly for subtrajectories).

7.2.1 Similarity between (sub)trajectories

Subtrajectory clustering relies on the use of a similarity function between subtrajectories. Although various similarity measures have been defined in literature, our choice of similarity function is motivated by the following (desired) requirements:

Variable sampling rate and lack of alignment. We make the realistic assumption that the trajectories do not have a fixed sampling rate and that different trajectories might not report their position at the same timestamp.

Variable trajectory length. We also assume that different trajectories might have different length (i.e. number of samples). This specification excludes euclidean-based similarity measures which deal with trajectories of equal length.

Temporal displacement. A property that a desired similarity measure for (sub)trajectory clustering should hold, is to allow trajectories that have some temporal displacement to participate to the same cluster.

Symmetry. Given a pair of (sub)trajectories r and s, an appropriate similarity measure between r and s should have the property of symmetry (i.e. $Sim(r,s)$=$Sim(s,r)$).

Efficiency. The computation of the similarity should be efficient enough in order to be able to deal with massive volumes of data, without compromising the quality of the results.

In order to meet with the aforementioned specifications we utilize the Longest Common Subsequence (LCSS) for trajectories, as defined in [97]. However, other trajectory similarity functions, which meet with the specifications set, are also applicable. More specifically, the LCSS utilizes two parameters, the parameter ϵ_t indicating the temporal range wherein the method searches to match a specific point, and the ϵ_{sp} parameter which is a distance threshold to indicate whether two points match or not. Hence, the similarity between two (sub)trajectories r and s is defined as:

$$Sim(r,s) = \frac{LCSS_{\epsilon_t,\epsilon_{sp}}(r,s)}{min(|r|,|s|)} \tag{7.1}$$

where $|r|$ ($|s|$) is the length of r (s respectively). Moreover, it holds that $Sim(r,s) = Sim(s,r)$.

However, LCSS returns the length of the longest common subsequence, which means that for a given point $r_i \in r$ that is matched with a specific point $s_j \in s$ the LCSS will consider the similarity between r_i and s_j as 1, regardless of their actual distance $d_s(r_i, s_j)$, which could vary from 0 to ϵ_{sp}. Put differently, LCSS considers as equally similar all the points that exist within an ϵ_{sp} range from r, which is a fact that might compromise the quality of the clustering results. Ideally, given two matching points $r_i \in r$ and $s_j \in s$, s_j (r_i, respectively) should contribute to $LCSS_{\epsilon_t,\epsilon_{sp}}(r,s)$, proportionally to the distance $d_s(r_i, s_j)$. For this reason, we propose a "weighted" LCSS similarity between trajectories, that incorporates the aforementioned distance proportionality. In more detail, for each discovered longest common subsequence the similarity is defined as:

$$Sim(r,s) = \frac{\sum_{k=1}^{min(|r|,|s|)} (1 - \frac{d_s(r_k,s_k)}{\epsilon_{sp}})}{min(|r|,|s|)} \tag{7.2}$$

where (r_k, s_k) is a pair of matched points.

7.2.2 A Closer Look to the Subtrajectory Clustering Problem

Our approach to subtrajectory clustering splits the problem in three steps. The first step is to retrieve for each trajectory $r \in D$, all the moving objects, with their respective portion of movement, that moved close enough in space and time with r, for at least some time duration. Actually, this first step is a well-defined problem in the literature of mobility data management, known as *subtrajectory join*, and more specifically the case of self-join. In detail, the subtrajectory join will return for each pair of (sub)trajectories, all the common subsequencies that have at least some time duration, which are actually candidates for the longest common subsequence. Formally:

Problem 7.1. ***(Subtrajectory Join)*** *Given a temporal tolerance ϵ_t, a spatial threshold ϵ_{sp} and a time duration δt, retrieve all pairs of subtrajectories $(r', s') \in D$ such that: (a) for each pair $\Delta t_{r'}, \Delta t_{s'} \geq \delta t$, (b) $\forall r_i \in r'$ there exists at least one $s_j \in s'$ so that $d_s(r_i, s_j) \leq \epsilon_{sp}$ and $d_t(r_i, s_j) \leq \epsilon_t$, and (c) $\forall s_j \in s'$ there exist at least one $r_i \in r'$ so that $d_s(s_j, r_i) \leq \epsilon_{sp}$ and $d_t(s_j, r_i) \leq \epsilon_t$.*

Figure 7.2 illustrates two trajectories r and s and their respective *matching* subtrajectories $(r_{4,8}, s_{3,7})$. Each point of a trajectory defines a spatiotemporal 'neighborhood' area around it, i.e. a cylinder of radius ϵ_{sp} and height ϵ_t. In order for a pair of subtrajectories to be considered matching, each point of a subtrajectory must have at least one point of the other subtrajectory in its neighborhood, thus making the result symmetrical. Furthermore the

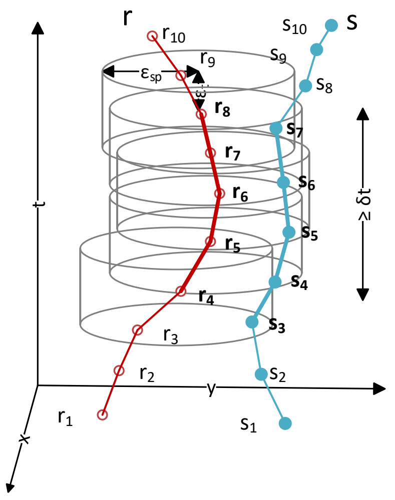


Figure 7.2: A pair of “matching” subtrajectories $(r_{4,8}, s_{3,7})$.

duration of the match should be at least δt.

The second step takes as input the result of the first step, which is actually a trajectory and neighboring trajectories and aims at segmenting each trajectory $r \in D$ into a set of subtrajectories. The way that a trajectory is segmented into subtrajectories is neighbourhood-aware, meaning that a trajectory will be segmented every time its neighbourhood changes significantly, so as to result in homogeneous subtrajectories (with respect to their surrounding moving objects). Returning to Example 7.1, trajectory $A \rightarrow D$ should be segmented to $A \rightarrow O$ and $O \rightarrow D$, since at O the cardinality and the composition of its neighbourhood changes significantly. The problem of trajectory segmentation can now be formulated as follows.

Problem 7.2. ***(Trajectory Segmentation)*** *Given a trajectory r, identify the set of timestamps CP (cutting points), where the density (or alternatively the composition) of the neighborhood of r changes significantly. Then according to CP, r is partitioned to a set of subtrajectories $\{r'_1, \ldots, r'_M\}$, where $M = |CP| + 1$ is the number of subtrajectories for a given trajectory r, such that $r = \bigcup_{k=1}^{M} r'_k$ and $k \in [1, M]$.*

Given the output of Problem 7.1, applying a trajectory segmentation algorithm for the trajectories D will result in a new set of subtrajectories D'.

The third step takes as input D' and the goal is to create clusters (whose cardinality is unknown) of similar subtrajectories and at the same time identify subtrajectories that are significantly dissimilar from the others (outliers). More specifically, let $C = \{C_1, \ldots, C_K\}$ denote the clustering, where K is the number of clusters, and for every pair of clusters C_i and C_j, with $i, j \in [1, K]$, it holds that $C_i \cap C_j = \emptyset$. Now, let us assume that each cluster $C_i \in C$ is represented by one subtrajectory $R_i \in C_i$, called *Representative*. Furthermore, let R denote the set of all representatives. Actually, the problem of clustering is to discover clusters of objects such that the intra-cluster similarity is maximized and the inter-cluster similarity is minimized. Therefore, if we ensure that the similarity between the representatives is zero, then the problem of subtrajectory clustering can be formulated as an optimization problem as follows.

Problem 7.3. ***(Subtrajectory Clustering and Outlier Detection)*** *Given a set of subtrajectories D', partition D' into a set of clusters C and a set of outliers O, where $D' = C \cup O$, in such a way so that the Sum of Similarity between Cluster members and cluster Representatives (*SSCR*) is maximized:*

$$SSCR = \sum_{\forall R_i \in R} \sum_{\forall r'_j \in C_i} Sim(R_i, r'_j) \tag{7.3}$$

However, trying to solve Problem 7.3 by maximizing Equation (7.3) is not trivial, since the problem to segment trajectories to subtrajectories, select the set of representatives R and its cardinality $|R|$ that maximizes Equation (7.3), has combinatorial complexity.

7.2.3 Distributed Subtrajectory Clustering

In this chapter, we address the challenging problem of subtrajectory clustering in a distributed setting, where the dataset D is stored distributed in different nodes, and centralized processing is prohibitively expensive.

Problem 7.4. ***(Distributed Subtrajectory Clustering)*** *Given a distributed set of trajectories, $D = \cup_{i=1}^{P} D_i$, where P is the number of partitions of D, perform the subtrajectory clustering task in a parallel manner.*

Actually, Problem 7.4 can be broken down to solving Problems 7.1, 7.2 and 7.3 (in that order) in a parallel/distributed way. In the following, we adopt this approach and outline a solution that is based on MapReduce.

7.3 Problem Solution

7.3.1 Overview

An overview of our approach is presented in Algorithm 7.1 and illustrated in Figure 7.3. Initially, we *Repartition* the data into P equi-sized, temporally-sorted partitions (files), which are going to be used as input for the join algorithm in order to perform the subtrajectory join in a distributed way (line 3). Note that this is actually a preprocessing step that only needs to take place once for each dataset D.

Algorithm 7.1 *DSC*(D)

```
1:  Input: $D$
2:  Output: set $C$ of clusters, set $O$ of outliers
3:  Preprocessing: Repartition $D$;
4:  for each partition $D_i \in \cup_{i=1}^{P} D_i$ do
5:    perform Point-level Join;
6:  group by Trajectory;
7:  for each Trajectory $r \in D$ do
8:    perform Subtrajectory Join; – Sect. 7.3.2
9:    perform Trajectory Segmentation; – Sect. 7.3.3
10: group by $D_i$;
11: for each subtrajectory $r' \in D_i$ do
12:   calculate Similarity with other subtrajectories; – Sect. 7.3.3
13: perform Clustering; – Sect. 7.3.4
14: perform Refine Results;
15: return $C$ and $O$;
```

Subsequently, for each partition $D_i \in \cup_{i=1}^{P} D_i$ and trajectory we discover parts of other trajectories that moved close enough in space an time (line 5). Successively, we group by trajectory in order to perform the subtrajectory join (line 8). At this phase, since our data is already grouped by trajectory, we also perform trajectory segmentation in order to split each trajectory to subtrajectories (line 9). In turn, we utilize the temporal partitions created during the *Repartition* phase and re-group the data by temporal partition. For each $D_i \in \cup_{i=1}^{P} D_i$ we calculate the similarity between subtrajectories and perform the clustering procedure (line 12). If a subtrajectory intersects the borders of two partitions, it is replicated in both of them. This results in having duplicate and possibly contradicting results. For this reason, as a final step, we treat this case by utilizing the *Refine Results* procedure (line 14). Finally, a set C of clusters and O of outliers are produced.

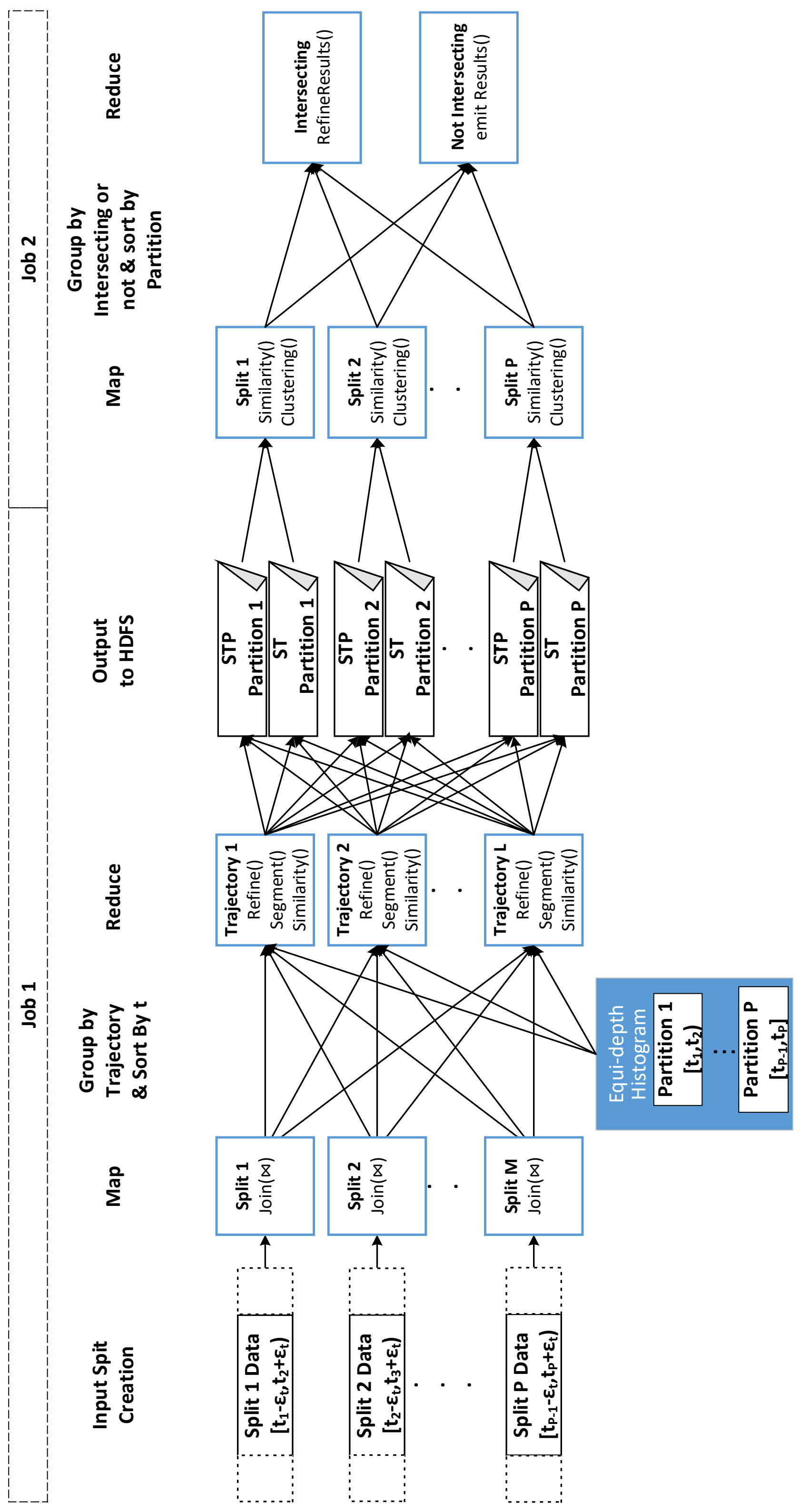

Job 1
Job 2
Input Spit Creation
Map
Group by Trajectory & Sort By t
Reduce
Output to HDFS
Map
Group by Intersecting or not & sort by Partition
Reduce
Split 1 Data $[t_1-\varepsilon_t, t_2+\varepsilon_t)$
Split 2 Data $[t_2-\varepsilon_t, t_3+\varepsilon_t)$
Split P Data $[t_{P-1}-\varepsilon_t, t_P+\varepsilon_t)$
Split 1 Join(⋉)
Split 2 Join(⋉)
Split M Join(⋉)
Equi-depth Histogram
Partition 1 $[t_1, t_2)$
Partition P $[t_{P-1}, t_P]$
Trajectory 1 Refine() Segment() Similarity()
Trajectory 2 Refine() Segment() Similarity()
Trajectory L Refine() Segment() Similarity()
STP Partition 1
ST Partition 1
STP Partition 2
ST Partition 2
STP Partition P
ST Partition P
Split 1 Similarity() Clustering()
Split 2 Similarity() Clustering()
Split P Similarity() Clustering()
Intersecting RefineResults()
Not Intersecting emit Results()

Figure 7.3: The *DSC* algorithm. (Job 1) *DTJ* and *Trajectory Segmentation* and (Job 2) *Clustering* and *Refine Results.*

7.3.2 Distributed Subtrajectory Join

As already mentioned, the first step is to perform the subtrajectory join in a distributed way. For this reason, we exploit the work presented in Chapter 6, coined DTJ, which introduces an efficient and highly scalable approach to deal with Problem 7.1, by means of MapReduce.

7.3.3 Distributed Trajectory Segmentation

The *Trajectory Segmentation* algorithm (TSA) takes as input a single trajectory, along with information about its neighborhood, and partitions it to a set of subtrajectories. We have already presented a neighborhood-aware trajectory segmentation algorithm in Section 3.3.1. However, the voting function introduced in Equation 3.3 considers that the space is unbounded (in our case the space is bounded by ϵ_{sp}) and, for this reason, it utilizes an extra parameter, coined σ, in order to control how fast the "voting influence" decreases with distance. It turns out that setting parameter σ is not a straightforward task and can affect significantly the outcome of the segmentation procedure. Furthermore, the segmentation algorithm presented in Section 3.3.1 takes into account only the density and not the actual composition of the neighborhood of a trajectory. For the above reasons, in this chapter, we propose two alternative segmentation algorithms.

The first algorithm, coined TSA_1, identifies the beginning of a new subtrajectory whenever the density of its neighborhood changes significantly. Such a segmentation algorithm is reminiscent of the flock definition [48], where the identified groups need to be composed of at least m objects. For this purpose, we use the concept of *voting* as a measure of density of the surrounding area of a trajectory. For a given point r_i and any trajectory s, the voting $V(r_i)$ is defined as:

$$V(r_i) = \sum_{\forall s \in D} \frac{d_s(r_i, s_k)}{\epsilon_{sp}} \tag{7.4}$$

where, s_k is the matching point of s with r_i, as emitted by the subtrajectory join procedure. For a trajectory r that consists of N points $\{r_1, \ldots, r_N\}$, we compute its normalized voting vector $\overline{V}(r)$ as follows:

$$\overline{V}(r)[] = \{\frac{V(r_1)}{\max_{i=1}^{N} V(r_i)}, \ldots, \frac{V(r_N)}{\max_{i=1}^{N} V(r_i)}\} \tag{7.5}$$

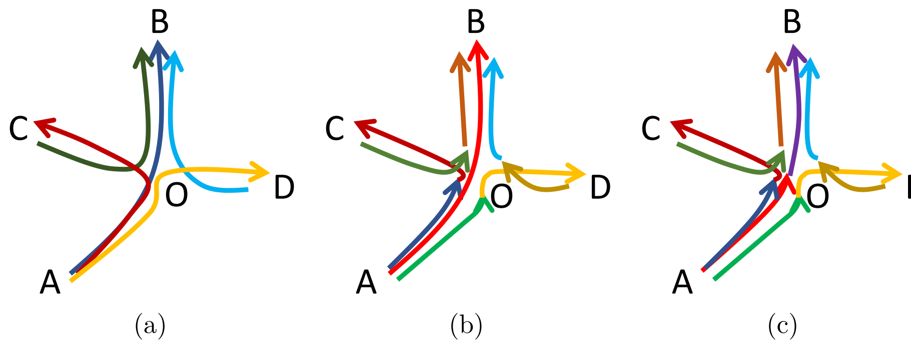


Figure 7.4: (a) Five trajectories $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow D$, $C \rightarrow B$ and $D \rightarrow B$, (b) TSA_1 segmentation, (c) TSA_2 segmentation

Finally, the voting of a trajectory (or subtrajectory) is defined as:

$$V(r) = \frac{1}{N} \sum_{i=1}^{N} V(r_i) \tag{7.6}$$

The difference of TSA_1

The second segmentation algorithm, coined TSA_2, identifies the beginning of a new subtrajectory whenever the composition of its neighborhood changes substantially. This segmentation algorithm is reminiscent of the moving cluster definition [45], where the identified groups need to share a sufficient number of common objects. Such an algorithm does not take as input the $\overline{V}(r)[]$ but instead, for each point $r_i \in r$, it takes as input a list $L(r_i)[]$ of the trajectory ids that have been produced as output by the DTJ procedure.

The following example explains intuitively the difference between the two segmentation algorithms.

Example 7.2. *Consider the example of Figure 7.4(a) that illustrates five trajectories: $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow D$, $C \rightarrow B$ and $D \rightarrow B$. Figures 7.4(b) and (c) depict the result of TSA_1 and TSA_2, respectively. In more detail, we can observe that both TSA_1 and TSA_2 segmented trajectory $A \rightarrow D$ to subtrajectories $A \rightarrow O$ and $O \rightarrow D$, due to the fact that after O, both the density and the composition of the neighborhood changes. The same holds for trajectories $A \rightarrow C$, $C \rightarrow B$ and $D \rightarrow B$, which are segmented to subtrajectories $A \rightarrow O$, $O \rightarrow C$, $C \rightarrow O$, $O \rightarrow B$, $D \rightarrow O$ and $O \rightarrow B$. However, when it comes to trajectory $A \rightarrow B$, we can observe that while TSA_2 segments it to subtrajectories $A \rightarrow O$ and $O \rightarrow B$, TSA_1 does not*

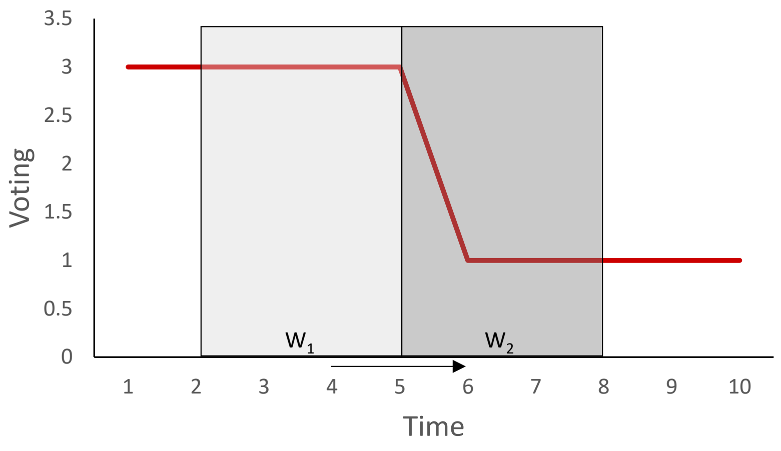


Figure 7.5: The two consecutive sliding windows W_1 and W_2 used by the segmentation algorithms.

perform any segmentation. This is due to the fact that, after O, even though the density of the neighborhood remains the same (i.e. 3 moving objects), the composition of the neighborhood changes completely. In a subsequent step this will drive the clustering algorithm to identify, in the case of TSA_1 a flock-like cluster from A to B, while in the case of TSA_2 two moving clusters from A to O and from O to B.

Both segmentation algorithms share a common methodology, which employs two consecutive sliding windows W_1 and W_2 of size w (i.e. w samples) to estimate the point $r_i \in CP$ (cutting point) where the “difference” between the two windows is maximized. This methodology has been successfully applied in the past on signal segmentation [63, 61]. To exemplify, let us consider trajectory $A \to D$ of Example 7.2. For simplicity, we assume that the voting of the specific trajectory from A to O is 3 and from O to D is 1. Figure 7.5 illustrates the two sliding windows W_1 and W_2 that traverse the voting signal of trajectory $A \to D$.

Trajectory segmentation. Since the output of the DTJ algorithm is per trajectory, it is straightforward to give it as input to TSA which operates at the level of a trajectory. Moreover, the segmentation is performed in an embarrassingly parallel way, due to the fact that each trajectory can be processed by a different reduce task independently from others, as depicted in Figure 7.3. In more detail, for a given trajectory $r \in D$, TSA_1 first calculates the normalized voting vector $\overline{V}(r)[]$ and then performs the segmentation by utilizing it. Apart from $\overline{V}(r)[]$, the input of the TSA algorithm is two additional parameters: w and τ. The output is a vector $CP[]$, which keeps the starting position of each subtrajectory of r.

Algorithm 7.2 $TSA_1(\overline{V}(r)[], w, \tau)$

```
1: Input: $\overline{V}(r)[], w, \tau$
2: Output: $CP[]$
3: $1 \rightarrow CP[]$;
4: for n = w+1 ... N-w-1 do
5:     $m_1 = \frac{1}{w}\sum_{i=n-w}^{n-1} \overline{V}(r)[i]$;
6:     $m_2 = \frac{1}{w}\sum_{i=n}^{n+w-1} \overline{V}(r)[i]$;
7:     $d[n] = |m_1 - m_2|$;
8:     $d_{max} = \max_{i=w+1}^{N-w-1} d[i]$;
9:     if $d[n] > \tau \wedge d[n] >= d_{max}$ then
10:        $n \rightarrow CP[]$;
```

In more detail, as presented in Algorithm 7.2, two consecutive sliding windows of size w are created over $\overline{V}(r)[]$, named W_1 and W_2 (line 4). These sliding windows move forward in time until $\overline{V}(r)[]$ is traversed. Here, N is the number of points of trajectory $r \in D$. Then, for each window, the average normalized voting is computed (lines 5-6) and their absolute difference is stored in $d[]$, which is an array that stores all the differences between the sliding windows (line 7). Subsequently, we examine whether the current difference $d[n]$ is larger than the maximum difference d_{max} and we update d_{max} accordingly (line 8). Finally, if the difference $d[n]$ is higher than a threshold τ and is locally maximized, then, at that point, we segment the trajectory and we store the starting position of the new subtrajectory to $CP[]$ (lines 9-10).

On the other hand, the input of TSA_2 is a list of lists $L(r)[]$ for each $r \in D$. Similarly, two consecutive sliding windows W_1 and W_2 of size w are created (line 4). Then, for each window, the union of lists is computed and stored in l_1 and l_2, respectively (lines 5-6). Successively, the Jaccard dissimilarity between l_1 and l_2 is computed and is stored to $d[]$, which is an array that stores all the similarities between the sliding windows (line 7). From then on, the algorithm is identical to TSA_1.

Similar subtrajectories. After trajectory segmentation, the next step is to calculate the similarity between all the pairs of subtrajectories, using Equation 7.2. This cannot be done completely after the segmentation at the *Reducer* phase of Job 1, illustrated in Figure 7.3, because at that point each reduce function has information only about the segmentation of the reference

Algorithm 7.3 $TSA_2(L(r), w, \tau)$

```
1:  Input: $L(r), w, \tau$
2:  Output: $CP[]$
3:  $1 \rightarrow CP[]$;
4:  for n = w+1 ... N-w-1 do
5:      $l_1 = \cup_{i=n-w}^{n-1} L(r_i)[]$;
6:      $l_2 = \cup_{i=n}^{n+w-1} L(r_i)[]$;
7:      $d[n] = 1 - \frac{|l_1 \cap l_2|}{|l_1|+|l_2|-|l_1 \cap l_2|}$;
8:      $d_{max} = \max_{i=w+1}^{N-w-1} d[i]$;
9:      if $d[n] > \tau \wedge d[n] >= d_{max}$ then
10:         $n \rightarrow CP[]$;
```

trajectory to subtrajectories. For this reason, at this point we cannot calculate the denominator of Equation 7.2. However, for each subtrajectory $r' \in r$, where r is the reference trajectory, we can calculate the similarity between the matching points (enumerator of Equation 7.2).

In more detail the output of each reduce function (Job 1 Figure 7.3) is a relation, called *STP*, which holds a set of key-value pairs of the form $< (r'.ID, s.ID), \{(s_f.t, Sim(s_f, r')) \ldots (s_l.t, Sim(s_l, r'))\} >$, where s_f, s_l are the temporal first and last point, respectively, of trajectory s that "matches" with subtrajectory r'. Moreover, in a separate relation, coined *ST*, we hold some extra information for each subtrajectory. More specifically, the tuples of *ST* are key-value pairs, where the key is the subtrajectory identifier $< ID >$ and the value is of the form $< t_s, t_e, V, Card >$, where t_s (t_e) is the starting time (ending time, respectively) of the subtrajectory, V is the voting and *Card* is the number of points which constitute the specific subtrajectory. Due to the fact that these two relations can be pretty large, we need to partition them into smaller files. In order to achieve this, we broadcast the load balanced temporal partitions that were created during the *Repartitioning* phase of *DTJ*. As illustrated in Figure 7.3, each reducer loads these partitions and assigns each subtrajectory (tuple of *ST* and *STP*) to all the partitions with which it temporally intersects. Subsequently, the tuples are grouped by temporal partition and each group is fed to a Mapper.

At this point, each *Mapper* has now all the information needed to calculate the similarity between all the pairs of subtrajectories (Equation 7.2), for each temporal partition separately. The similarity between subtrajectories is output in a new relation, called *SP*. Each tuple of this relation holds information about a subtrajectory r' and its similarity with all the other subtrajectories, whenever this similarity is larger than zero. More specifically, *SP* contains a set of key-value pairs where the key is the ID of the subtrajectory ($r'.ID$) and the value is a list *AdjLst* containing elements of the form ($s'.ID, Sim$), where s' is a subtrajectory for which it holds that $Sim(r', s') > 0$.

7.3.4 Distributed Clustering

Clustering. After having calculated the similarity between all pairs of subtrajectories for each temporal partition, we can proceed to the actual clustering and outlier detection procedure. The output of the similarity calculation process, namely SP, is actually an adjacency list. The intuition behind the proposed solution to Problem 7.3 is to select as cluster representatives, highly voted subtrajectories (Equation 7.6) that are not similar with the already selected representatives $R_i \in R$. Then, we assign each subtrajectory r'_k to the R_i (and hence C_i) with which it has the maximum similarity $Sim(r'_k, R_i)$.

Algorithm 7.4 *Clustering*(SP, ST, k, α)

```
1: Input: $SP, ST, k, \alpha$
2: Output: set $C$ of clusters, set $O$ of outliers
3: sort $ST$ by $V$ in descending order;
4: for each element $st \in ST$ do
5:   if $st \notin R$ then
6:     if $st.V \geq k$ then
7:       $st \rightarrow R$;
8:       for each element $l \in st.AdjLst$ do
9:         if $l \notin C$ then
10:          if $Sim(l, st) \geq \alpha$ then
11:            $l \rightarrow C(st)$;
12:            if $l \in O$ then
13:              $O = O - l$;
14:          else
15:            $O = O \cup l$
16:        else
17:          if $Sim(l, st) > Sim(l, R(l))$ then
18:            $C(R(l)) = C(R(l)) - l$;
19:            $l \rightarrow C(st)$;
20:     else
21:       $O = O \cup st$;
22: $C = C \cup R$
```

The input of the clustering algorithm is SP, ST and parameters k and α and the output is the set of clusters C and the set of outliers O. More specifically, k is a threshold for setting a lower bound on the voting of a representative. This prevents the algorithm from identifying clusters with small support. Parameter α is a similarity threshold used to assign subtrajectories to cluster representatives. It ensures that a subtrajectory assigned to a cluster has sufficient similarity with the representative of the cluster. This actually poses a lower bound to the average distance between the representatives and the cluster members and, consequently, guarantees a minimum quality in the identified clusters (intra-cluster distance).

Lemma 7.1. *The average distance $\overline{d_s(r', s')}$, between a representative subtrajectory r' and a cluster member s' will always be at most $\epsilon_{sp} \cdot (1 - \alpha)$.*

$$\overline{d_s(r', s')} \leq \epsilon_{sp} \cdot (1 - \alpha) \tag{7.7}$$

Proof.

$$Sim(r', s') = \frac{\sum_{k=1}^{min(|r'|,|s'|)} (1 - \frac{d_s(r'_k, s'_k)}{\epsilon_{sp}})}{min(|r'|, |s'|)}$$

$$Sim(r', s') = \frac{min(|r'|, |s'|) - \sum_{k=1}^{min(|r'|,|s'|)} \frac{d_s(r'_k, s'_k)}{\epsilon_{sp}}}{min(|r'|, |s'|)}$$

$$But, \sum_{k=1}^{min(|r'|,|s'|)} d_s(r'_k, s'_k) = min(|r'|, |s'|) \cdot \overline{d_s(r', s')}$$

$$So, \ Sim(r', s') = \frac{min(|r'|, |s'|) - \frac{min(|r'|,|s'|) \cdot \overline{d_s(r', s')}}{\epsilon_{sp}}}{min(|r'|, |s'|)}$$

$$Sim(r', s') = \frac{min(|r'|, |s'|) \cdot (1 - \frac{\overline{d_s(r', s')}}{\epsilon_{sp}})}{min(|r'|, |s'|)}$$

$$Sim(r', s') = 1 - \frac{\overline{d(r', s')}}{\epsilon_{sp}}$$

$$But, \ Sim(r', s') \geq \alpha$$

$$So, \ \overline{d_s(r', s')} \leq \epsilon_{sp} \cdot (1 - \alpha)$$

□

To begin with, we want to traverse the subtrajectories by their voting, in descending order (i.e. highly voted subtrajectories first). In order to achieve this, we need to sort ST by V (line 3). Subsequently, for each subtrajectory $st \in ST$ we examine whether it is already assigned to cluster (line 5). If st is not assigned to any cluster and the voting of st is less than k, then we add st to the outliers set (line 21). Otherwise, we create a new cluster and consider st as the representative (lines 6-7). Successively, we consult relation SP and retrieve the adjacency list of st (line 8). Then, for each element l that belongs to the adjacency list of st, we examine if it is assigned to any cluster. If not, we investigate whether the similarity between l and st is greater or equal than the similarity threshold α. If not, we add l to the outlier set O, otherwise we assign it to the cluster led by st and remove it from the outliers O, in case $l \in O$ (lines 9-13). If l is assigned to a cluster, we examine whether the similarity of l with st is greater than the similarity with the representative of the cluster that l is currently assigned. If this is the case, then we remove l from the current cluster and assign it to the cluster led by st (lines 17-19). Finally, we concatenate C with R (line 22) so as to return, except from the outlier set O, both cluster members and representatives.

Refinement of Results. At this point we successfully accomplished to deal with Problem 7.3 for each temporal partition. However, this might result in having duplicates due to the fact that each subtrajectory that temporally intersects multiple partitions is replicated to each one of them. The actual problem that lies here is not the duplicate elimination problem itself but the fact that the result for such a subtrajectory might be contradicting in different partitions. In more detail, for each partition, the clustering procedure will decide whether a subtrajectory is a *Representative* (*Repr*), a *Cluster Member* (*Cl*) or an *Outlier* (*Out*). Hence, for each intersecting subtrajectory q and for each pair of consecutive partitions (i, j) with which q intersects, q can have the following pairs of states: (a) *Out-Out*, (b) *Repr-Repr*, (c) *Cl-Cl*, (d) *Repr-Cl* (*Cl-Repr*), (e) *Repr-O* (*O-Repr*) and (f) *Cl-O* (*O-Cl*).

In order to implement the above procedure we need to have all the information concerning the intersecting subtrajectories (C and O) for all the Partitions sorted in time. To do this, we group the trajectories according to whether they are intersecting or not. As illustrated in Figure 7.3, the non-intersecting are emitted, since they are not affected, while the intersecting subtrajectories get sorted by partition. Hence, a *Reducer* will receive all the required information to make the appropriate decisions. In more detail, we sweep through the temporal dimension and for each pair of consecutive partitions we make the appropriate decisions.

Algorithm 7.5 *RefineResults*(q)

```
1: Input: Intersecting Subtrajectories
2: Output: set $C$ of clusters, set $O$ of outliers
3: for each pair p $\rightarrow (P_i, P_{i+1})$ of Partitions do
4:    $P_i \cap P_{i+1} \rightarrow I$
5:    for each element $e \in I$ do
6:       switch $(p)$
7:       case (a):
8:          remove $q$ from $O_i$;
9:       case (b):
10:         merge $C_i(q)$ and $C_{i+1}(q)$;
11:      case (c):
12:         if $Sim(q, R_i(q)) > Sim(q, R_{i+1}(q))$ then
13:            remove $q$ from $C_{i+1}$;
14:         else
15:            remove $q$ from $C_i$;
16:      case (d):
17:         remove $q$ from $C$;
18:      case (e),(f):
19:         remove $q$ from $O$;
20:      end switch
```

For each of the above cases, as depicted in Algorithm 7.5, a decision has to be made, in order to eliminate duplicates and provide the correct result according to the problem definition. More specifically, in case of (a), q is marked as outlier in both partitions, hence, we only need to eliminate duplicates. In case of (b), the two clusters are "merged", since all of the subtrajectories that belong to them are similar "enough" with q, which is the representative of both clusters. In case of (c), let us assume that q belongs to cluster $C_i(R(q))$ in Partition i and $C_{i+1}(R(q))$ in Partition $i+1$. Then, q is assigned to the cluster with which it has the largest similarity with its representative. In case of (d), q remains to be a cluster representative and is removed from the cluster C in which it is a member. Finally, in case of (e) and (f), q is removed from O.

7.4 Complexity Analysis

The purpose of this section is to analyse and provide insight to the complexity of the different algorithms that are involved to the solution to the *Distributed*

Subtrajectory Clustering problem, presented in this chapter.

DTJ: The complexity of the *Join* algorithm is roughly $O(|D|log_2Q)$, with Q being the average number of points per spatial index partition and $Q << |D|$. The complexity of the *Refine* algorithm is $O(T \cdot SW \cdot dt \cdot l)$, where T is the average number of points per trajectory, SW is the average number of points contained in a $\delta t + 2\epsilon_t$ window, dt the average number of points contained in a δt window and l is average the size of the $MatchingPoints$ list. For more details about the complexity of the algorithms involved in DTJ please refer to [89].

Segmentation: The complexity of the TSA_1 algorithm is $O(l \cdot |T|)$, where l is average the size of the "matching" list and $|T|$ is the average number of points per trajectory. The reason that we include l to this analysis is that in order to perform TSA_1, we first need to calculate the normalized voting vector. The complexity of the TSA_2 algorithm is also $O(l \cdot |T|)$, since l_1 and l_2 are already sorted by trajectory id and the list intersection can take place in linear time to the size of the lists.

Clustering: The complexity of *Clustering* algorithm is $O(|ST| \cdot log|ST| + |ST| \cdot |L|)$, with $|ST|$ being the number of subtrajectories, $|L|$ the average size of the adjacency list *AdjLst* and $|ST| \cdot log|ST|$ is the sorting cost. Here, we should mention that $|ST| << |D|$. Furthermore, ST and SP are implemented as HashMaps, hence key search has an $O(1)$ time complexity. The complexity of the $RefineResults$ algorithm is $O(M \cdot |P| \cdot |I|)$, where M is the number of temporal partitions, $|P|$ is the average number of intersecting subtrajectories per partition and I is the average size of the intersection. We should mention, here, that the intersection between two consecutive partitions is performed in linear time by utilizing HashSets sets.

7.5 Experimental Study

In this section, we present the findings of our experimental evaluation. The experiments were conducted in a 49 node Hadoop 2.7.2 cluster, provided by *okeanos*[1]. The master node consists of 8 CPU cores, 8 GB of RAM and 60 GB of HDD while each slave node is comprised of 4 CPU cores, 4 GB of RAM and 60 GB of HDD. Our configuration enables each slave node to launch 4 containers, thus up to 192 tasks (*Map* or *Reduce*) can be launched

[1]IAAS service for the Greek Research and Academic Community https://okeanos.grnet.gr/home/

simultaneously. For our experimental study, we employed two real datasets that will assist us to evaluate the performance, scalability and effectiveness of our solution. Furthermore, we utilized a synthetic dataset that simulates the case of Figure 7.1 in order to verify that our solution operates as anticipated, given a dataset with a known ground truth. The real datasets are from two different domains, namely the urban and the maritime domain. In more detail, the first one, is SIS and the second one is Brest, as described in Section 1.5.

Table 7.1: Parameters and default values (in bold) used in the experimental study of Chapter 7

Parameter	**Values**				
	(i)	(ii)	(iii)	(iv)	(v)
ϵ_{sp} (%)	10%	15%	**20%**	25%	30%
ϵ_t (%)	0%	25%	**50%**	75%	100%
δt (%)	0%	25%	**50%**	75%	100%
w	10	15	**20**	25	30
τ	0.2	0.4	**0.6**	0.8	1
α (in σ)	-2	-1	**0**	1	2
k (in σ)	-2	-1	**0**	1	2

Our experimental methodology is as follows: Initially, in Section 7.5.2 we verify the correctness of our solution by applying it to a dataset with a known ground truth and compare our findings with T-OPTICS [56], a well-known entire trajectory clustering technique. Moreover, we compare our solution with TraClus [49] and S^{2}T-Clustering [70], two state of the art subtrajectory clustering methods. Subsequently, in Section 7.5.3, we study the scalability of our solution by varying (a) the dataset size, and (b) the number of cluster nodes. Finally, in Section 7.5.4, we perform a sensitivity analysis in order to evaluate the effect of setting different values to the parameters of our solution, in terms of execution time and quality. Table 7.1 shows the experimental setting, where we vary the following parameters: ϵ_{sp}, ϵ_t, δt, w, τ, α and k and measure their effect in the performance and the effectiveness of our algorithms. We should mention that the default segmentation algorithm in our experimental study is TSA_1.

7.5.1 Parameter Setting

Setting the different parameters for different datasets can turn out to be an arbitrary procedure, which, in turn, can jeopardise the quality of the clustering results. For this reason, we provide some simple rules for setting the parameters relatively to the dataset being clustered, that do not compromise the quality of the results. In more detail, ϵ_{sp} can be set as a percentage of the dataset diameter. This, however, can be problematic when dealing with datasets having large spatial variation in their density (e.g., ports in the maritime domain). For this reason, we utilized the partitioning provided by the spatial index (QuadTree) of DTJ and calculated ϵ_{sp} for each point, as a percentage of the diameter of the cell of the QuadTree to which it belongs. Moreover, ϵ_t and δt are calculated relatively to the average duration between two consecutive trajectory samples ($\approx$ 1200 sec for SIS and $\approx$ 950 sec for AIS Brest).

Parameter w sets the size of the windows W_1 and W_2 upon which some measure is calculated. Small values on w can affect the robustness of the estimation, thus resulting to over-segmentation. On the other hand, large values of w can result to overlooking some cutting points due to the large window size. It has been observed that for $w \approx 20$ the robustness of the estimation is not affected and the size of the window is small enough so as not to overlook any cutting points. Concerning parameter τ, our experiments show that the best result in terms of quality is achieved for $\tau \approx 0.4$ Finally, the values of α and k can be set "around" the mean value of the similarity and the voting of the temporal partition, respectively, in terms of standard deviation. In fact, it has been observed that the average similarity and voting can produce clustering results of good quality. For more details about the effect, in terms of quality, of setting different values to the parameters of our solution, please refer to Section 7.5.4

7.5.2 Comparison with related work

Initially, so as to verify that our solution operates as expected, we utilize a synthetic dataset[2] that simulates the case of Figure 7.1. The only difference is that the two outliers mentioned there ($O \rightarrow A$ and $O \rightarrow B$), will now form clusters. Hence, the ground truth for the synthetic cluster becomes $A \rightarrow O$, $B \rightarrow O$, $O \rightarrow C$, $O \rightarrow D$, $O \rightarrow A$ and $O \rightarrow B$.

In fact, as depicted in Figure 7.6(a), T-OPTICS identifies the six original routes: $A \rightarrow B$ (in red), $A \rightarrow C$ (in blue), $A \rightarrow D$ (in orange), $B \rightarrow A$ (in yellow), $B \rightarrow C$ (in light blue) and $B \rightarrow D$ (in purple). On the other

[2]The original dataset was found in [55]

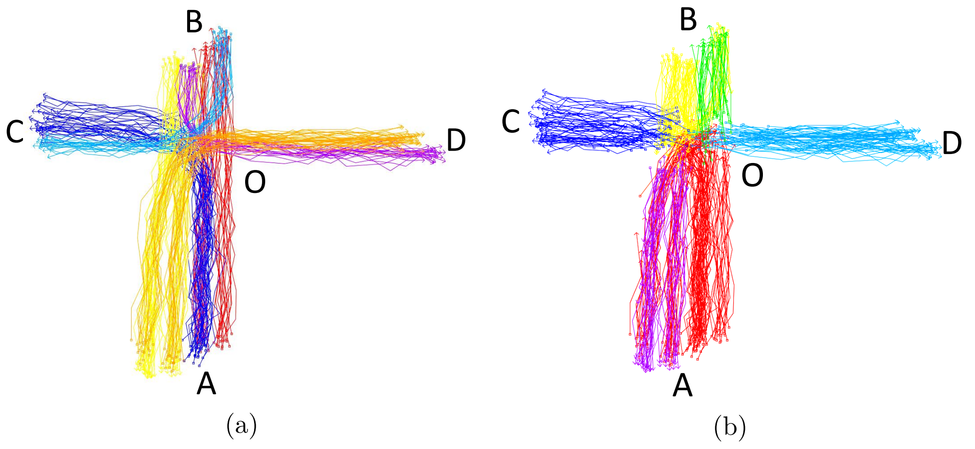


Figure 7.6: Identified clusters by (a) *T-OPTICS* and (b) *DSC*

hand, *DSC* identifies, with *Accuracy* = 100% and *F-measure* = 1, the six expected clusters of subtrajectories: $A \rightarrow O$(in red), $B \rightarrow O$ (in yellow), $O \rightarrow C$ (in blue), $O \rightarrow D$ (in light blue), $O \rightarrow A$ (in purple) and $O \rightarrow B$ (in green).

Subsequently, we compare *DSC* with two state of the art subtrajectory clustering algorithms, *S^2T-Clustering* and *TraClus*. The metric that we employ in order to evaluate the quality of the outcome of the clustering procedure is the well-known *RMSE* metric, which is actually a measure of intra-cluster distance between the representatives and the cluster members. Hence the larger the *RMSE*, the higher the intra-cluster distance and consequently the lower the quality of the clustering. It is obvious that, under this definition, *RMSE* is equivalent to *SSRC* (Equation 7.3). In order to perform this experiment, we utilized the 20% of each dataset which was further partitioned in 4 portions (25%, 50%, 75%, 100%). This choice was necessary because the centralized implementations of *S^2T-Clustering* and *TraClus* could not scale with the full size of the datasets that we utilized.

As illustrated in Figure 7.7, *DSC* outperforms, in terms of *RMSE*, both *TraClus* and *S^2T-Clustering*. In more detail, *TraClus* presents the largest *RMSE* which is somehow anticipated, since the specific algorithm utilizes a density-based approach to cluster subtrajectories, which in turn, through cluster expansion, can lead to spatially extended clusters. On the other hand, *S^2T-Clustering* presents smaller *RMSE* than *TraClus*, due to the fact that it adopts a distance-based approach and discovers more compact clusters. However, *DSC* results in smaller *RMSE* than *S^2T-Clustering*, mostly due

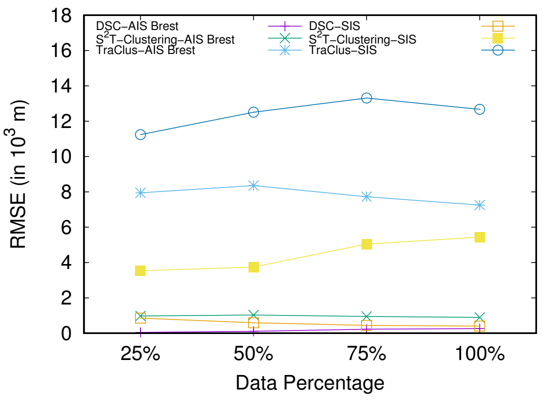


Figure 7.7: Comparison of the RMSE metric between *DSC*, *S^{2}T-Clustering* and *TraClus*

to the fact that in the latter, two trajectories might end-up in the same cluster even if they have small “matching portions”. However, in *DSC* this “matching portions” should have a minimum (δt) duration.

7.5.3 Performance and Scalability

Initially, we vary the size of our datasets and measure the execution time of our algorithms. We show the impact of the individual steps: *Join*, *RSE*, *Clustering* and *RefineResult* using stacked bars. To study the effect of dataset size, we created 4 portions (20%, 40%, 60%, 80%) of the original datasets. *RSE* stands for the *Refine* and *Segmentation* procedure (Figure 7.3, Job 1, Reduce phase). As illustrated in Figures 7.8(a) and (b), as the size of the dataset increases, *DSC* appears to scale linearly. Subsequently, we keep the size of the datasets fixed (at 100%) and vary the number of nodes. As the number of nodes increases and the dataset size remains the same, it is expected that the execution time will decrease. Indeed, as depicted in Figures 7.8(c) and (d), as the number of nodes increases, *DSC* presents linear speedup. This linear behaviour, is somehow anticipated due to the fact that the *DSC* approach is dominated by *DTJ*, in terms of execution time, which presents linear speedup, as shown in [89].

Investigating further the performance of the different steps of our proposal, we can observe that, as expected, the execution time of the whole procedure is dominated by the Join step (Figure 7.3, Job 1, Map phase), followed by *RSE*. Finally, as anticipated, the *Clustering* and the *RefineResults* step (Figure 7.3, Job 2) present very good performance, since the computationally

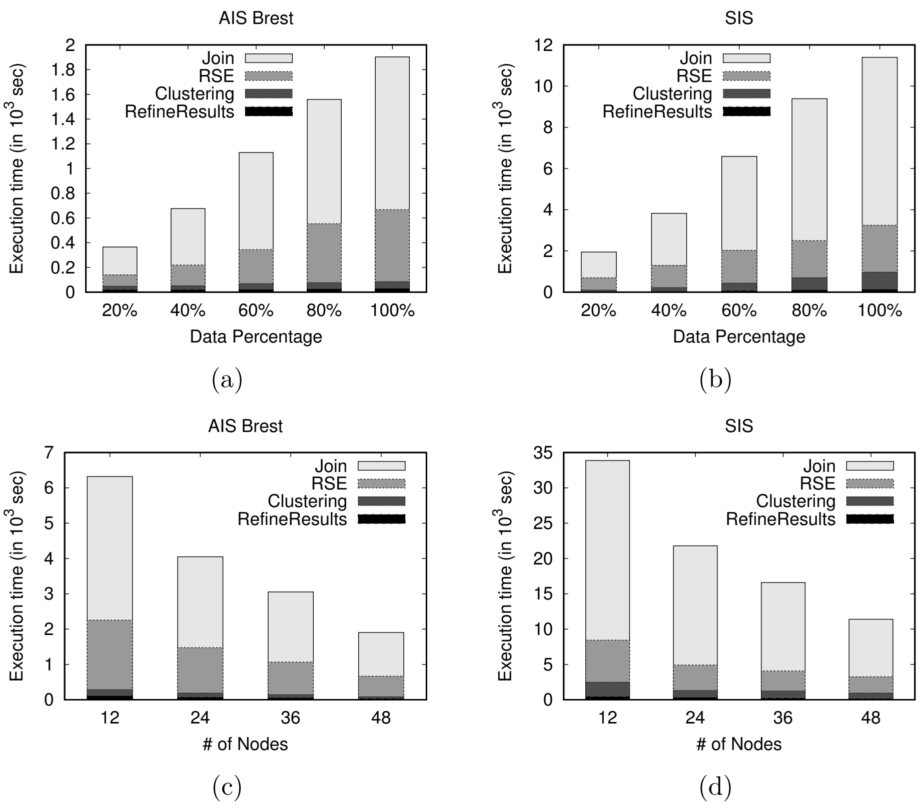


Figure 7.8: Scalability analysis varying the size of the (a) AIS Brest and (b) SIS dataset and the number of nodes over the (c)AIS Brest and (d) SIS dataset

intensive part of the similarity matrix calculation has already been done as part of the previous steps.

7.5.4 Sensitivity Analysis

In this section, we perform a sensitivity analysis of all the involved parameters. More specifically, we vary each parameter presented in Table 7.1, while keeping the rest of them in their default value (bold), and we measure their effect in the execution time and the quality of the clustering results, in terms of *RMSE*. Figures 7.9(a) and (b) show that the parameters that appear to have a significant impact on execution time are ϵ_t and ϵ_{sp}. This is justified from the fact that these parameters actually affect significantly the complexity of the Join step (Figure 7.3, Job 1, Map phase), which is the dominant cost of *DSC*. Another parameter that seems to have a perceivable

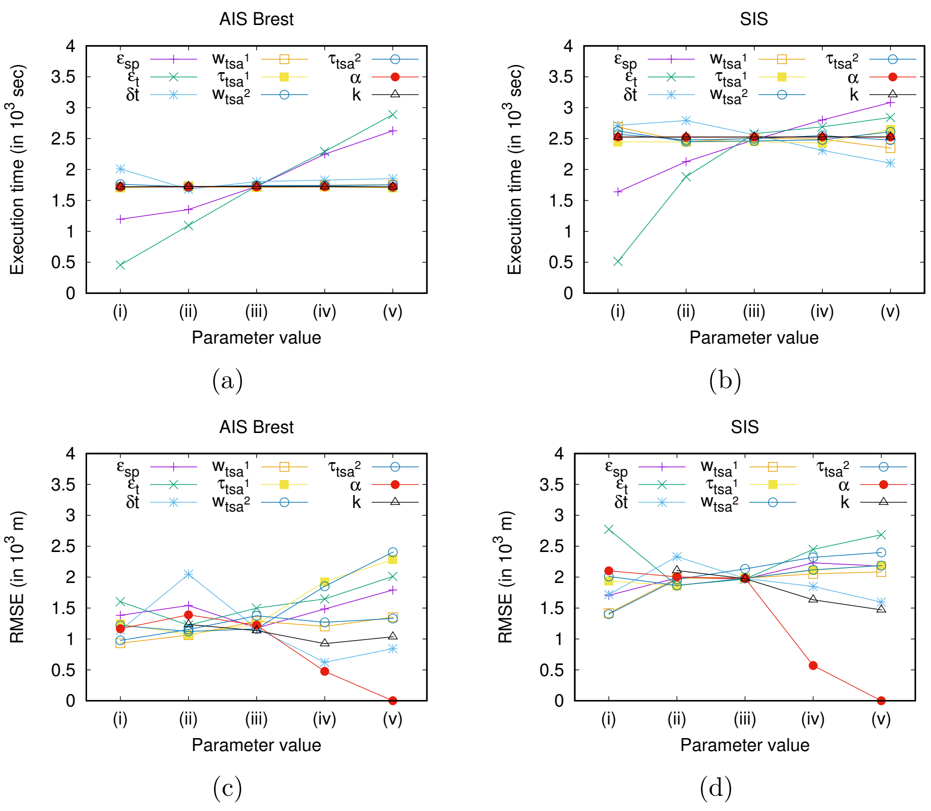


Figure 7.9: Sensitivity analysis in terms of execution time of (a) the AIS Brest and (b) the SIS dataset and in terms of *RMSE* of (c) the AIS Brest and (d) the SIS dataset

effect on the execution time, is δt, which in fact “filters” the results of *DTJ*, thus fewer data reach the next steps.

Regarding the quality of the clustering results, as illustrated in Figure 7.9(c) and (d), all the parameters seem to have an effect over it. In more detail, the larger the values of ϵ_t and ϵ_{sp}, the larger the *RMSE*. This behaviour is expected since we allow objects that are further away from a representative to participate to the same cluster. In contrast, as δt increases, the *RMSE* decreases, which is also anticipated since it sets a lower bound to the longest common subsequence. Furthermore, all the parameters that control the segmentation have the same effect on the *RMSE*, i.e. the smaller (in length) the subtrajectories, the smaller the *RMSE*. This shows that breaking trajectories to subtrajectories has a positive effect on the quality of the clustering and justifies the motivation of our work. Moreover, as α increases the *RMSE* decreases, since for small values of α, less similar objects are allowed to participate in a cluster. Finally, the larger the k the smaller the *RMSE*, since it disallows the identification of clusters with small support.

7.6 Summary

In this chapter, we addressed the problem of *Distributed Subtrajectory Clustering* by building upon a scalable subtrajectory join query operator in order to tackle the problem in an efficient manner. Subsequently, we proposed two alternative trajectory segmentation algorithms. Finally, we proposed a distributed clustering algorithm where the clusters are identified in a parallel manner and get refined as a final step. Our experimental study was performed on a synthetic and two large real datasets of trajectories from the urban and the maritime domain.

Outlook Part IV

8 Conclusions

In this chapter, we summarize the contributions of this thesis and point to future research directions.

In Part II, we discussed the problem of subtrajectory clustering and outlier detection in trajectory databases. In more detail, in Chapter 3, we proposed a novel subtrajectory clustering algorithm, namely S^2T-Clustering, that is novel not only because it solves the problem more effectively than the state-of-the-art (namely, TRACLUS), but also due to the fact that our proposal is designed in-DBMS, i.e., it is registered as a query operator in a real MOD engine over an extensible DBMS (namely, PostgreSQL in our prototype implementation). Having such functionality in their hands, data scientists are able to perform cluster analysis via simple SQL in a real DBMS. Moreover, in Chapter 4, we introduced the *temporally-constrained subtrajectory cluster analysis* problem. To address it, we proposed *ReTraTree*, an indexing scheme which organizes trajectories by using an effective spatiotemporal partitioning technique. Partitions in *ReTraTree* correspond to groupings of subtrajectories, which are incrementally maintained and represented via a hierarchical organization of a small (thus, light-weight in-memory) set of 'representative' subtrajectories. Given this, the problem in hand can be efficiently solved as a query operator on *ReTraTree*, coined *QuT-Clustering.* Our approach further contributes to the mobility data management and mining domain for the additional reason that it has been designed and implemented in a MOD engine. Such functionality enables the application users to perform progressive cluster analysis via simple SQL in real extensible DBMS. Finally in Chapter 5, we proposed an efficient in-DBMS architecture for progressive time-aware subtrajectory cluster analysis, by utilizing the work done in Chapter 3 and 4 along with a Visual Analytics (VA) tool to facilitate real world analysis.

Recognizing the limitations imposed by the Big Data era of the work proposed in Part II, in Part III, we dealt with the problem at hand, in a distributed and highly scalable way by utilizing the popular MapReduce programming paradigm. In more detail, in Chapter 6, we introduced the *Distributed Subtrajectory Join* query, an important operation in the spatiotemporal data management domain, where very large datasets of moving object trajectories are processed for analytic purposes. To address this problem in an efficient manner we followed the MapReduce programming model by proposing a well-designed basic solution and two efficient improvements, called $DTJr$ and $DTJi$ which boosted the performance by up to $16\times$ and $10\times$, respectively. Our experimental study was performed on IMIS, a very large real dataset of trajectories from the maritime domain, consisting of 56 GB of data (or 1.5 billion time-stamped locations), which showed that our proposal scales linearly to the size of the dataset and the number of processing nodes. Finally, in Chapter 7, we addressed the problem of *Distributed Subtrajectory Clustering* by building upon the *Distributed Subtrajectory Join* query in order to tackle the problem in an efficient manner. Subsequently, we proposed two alternative trajectory segmentation algorithms. Finally, we proposed a distributed clustering algorithm where the clusters are identified in a parallel manner and get refined as a final step. Our experimental evaluation, which was performed on a synthetic (CrossSection) and two large real datasets of trajectories from the urban (SIS) and the maritime domain (Brest), showed that our proposal scales linearly to the size of the dataset and the number of processing nodes. Regarding the quality of the clustering results, it was shown that our solution outperforms, in terms of *RMSE*, both *TraClus* and *S^2T-Clustering*, two state of the art subtrajectory clustering algorithms.

9 Ideas for Future Work

In this chapter, we present ideas and potential directions for future work.

Concerning the *temporally-constrained subtrajectory cluster analysis* problem, although our proposal presented in Chapter 4 is orders of magnitude more efficient than state-of-the-art spatial DBMS, the execution time of a clustering analysis for a big dataset is not satisfactory. Thus, a limitation of our approach is that it is not directly applicable to big datasets. To this end, in the future, we plan to investigate real-time solutions, exploiting on modern in-memory big data architectures.

Regarding the *Distributed Subtrajectory Join*, we plan to investigate how the solution provided in Chapter 6 can be applicable to streaming trajectories. Moreover, we plan to examine how this query can be extended and utilized in order to be able to identify efficiently various mobility patterns (e.g., flocks, convoys, moving clusters swarms etc.) over massively distributed data. We also plan to investigate the potential of extending the solution proposed in Chapter 6 to tackle the problem of k-nn trajectory join.

As far as it concerns the *Distributed Subtrajectory Clustering* problem, presented in Chapter 7, we plan to extend our solution with properties of density-based clustering algorithms. Furthermore, since our algorithm employs a single pass from the data we will investigate the possibility of addressing the same problem in a streaming environment.

Finally, we plan to investigate the problem of distributed online monitoring of mobility patterns. The goal will be to build a framework that, given a massive stream of mobility data, will be able to monitor and record various kinds of mobility patterns (flocks, convoys, moving clusters etc.), based on the concept of (sub)trajectory similarity and clustering, in real time.

Bibliography

[1] Hermes@PostgreSQL MOD engine. URL: http://infolab.cs.unipi.gr/hermes.

[2] P. Ramsey (on behalf of PostGIS), personal communication.

[3] P. K. Agarwal, K. Fox, K. Munagala, A. Nath, J. Pan, and E. Taylor. Subtrajectory clustering: Models and algorithms. In *PODS*, pages 75–87, 2018.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE*, pages 3–14, 1995.

[5] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. H. Saltz. Hadoop-gis: A high performance spatial data warehousing system over mapreduce. *PVLDB*, 6(11):1009–1020, 2013.

[6] G. L. Andrienko, N. V. Andrienko, P. Bak, D. A. Keim, and S. Wrobel. *Visual Analytics of Movement*. Springer, 2013.

[7] G. L. Andrienko, N. V. Andrienko, S. Rinzivillo, M. Nanni, and D. Pedreschi. A visual analytics toolkit for cluster-based classification of mobility data. In *SSTD*, pages 432–435, 2009.

[8] M. Ankerst, M. M. Breunig, H. Kriegel, and J. Sander. OPTICS: ordering points to identify the clustering structure. In *SIGMOD*, pages 49–60, 1999.

[9] P. Bakalov, M. Hadjieleftheriou, E. J. Keogh, and V. J. Tsotras. Efficient trajectory joins using symbolic representations. In *MDM*, pages 86–93, 2005.

[10] P. Bakalov, M. Hadjieleftheriou, and V. J. Tsotras. Time relaxed spatiotemporal trajectory joins. In *ACM-GIS*, pages 182–191, 2005.

[11] P. Bakalov and V. J. Tsotras. Continuous spatiotemporal trajectory joins. In *GSN*, pages 109–128, 2006.

[12] M. Benkert, J. Gudmundsson, F. Hübner, and T. Wolle. Reporting flock patterns. *Comput. Geom.*, 41(3):111–125, 2008.

[13] K. Buchin, M. Buchin, M. J. van Kreveld, and J. Luo. Finding long and similar parts of trajectories. *Comput. Geom.*, 44(9):465–476, 2011.

[14] M. Buchin, A. Driemel, M. J. van Kreveld, and V. Sacristán. An algorithmic framework for segmenting trajectories based on spatio-temporal criteria. In *SIGSPATIAL*, pages 202–211, 2010.

[15] H. P. Chen, U. Dayal, and M. Hsu. Prefixspan,: mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE*, 2001.

[16] L. Chen, Y. Gao, Z. Fang, X. Miao, C. S. Jensen, and C. Guo. Real-time distributed co-movement pattern detection on streaming trajectories. *PVLDB*, 12(10):1208–1220, 2019.

[17] Y. Chen and J. M. Patel. Design and evaluation of trajectory join algorithms. In *SIGSPATIAL*, pages 266–275, 2009.

[18] P. Cudré-Mauroux, E. Wu, and S. Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *ICDE*, pages 109–120, 2010.

[19] V. T. de Almeida, R. H. Güting, and T. Behr. Querying moving objects in SECONDO. In *MDM*, page 47, 2006.

[20] J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[21] Z. Deng, Y. Hu, M. Zhu, X. Huang, and B. Du. A scalable and fast OPTICS for clustering trajectory big data. *Cluster Computing*, 18(2):549–562, 2015.

[22] H. Ding, G. Trajcevski, and P. Scheuermann. Efficient similarity join of large sets of moving object trajectories. In *TIME*, pages 79–87, 2008.

[23] X. Ding, L. Chen, Y. Gao, C. S. Jensen, and H. Bao. Ultraman: A unified platform for big trajectory data management and analytics. *PVLDB*, 11(7):787–799, 2018.

[24] S. Dodge, R. Weibel, and A. Lautenschütz. Towards a taxonomy of movement patterns. *Information Visualization*, 7(3-4):240–252, 2008.

[25] C. Doulkeridis and K. Nørvåg. A survey of large-scale analytical query processing in mapreduce. *VLDB J.*, 23(3):355–380, 2014.

[26] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE*, pages 1352–1363, 2015.

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

[28] Q. Fan, D. Zhang, H. Wu, and K. Tan. A general and parallel platform for mining co-movement patterns over large-scale trajectories. *PVLDB*, 10(4):313–324, 2016.

[29] Y. Fang, R. Cheng, W. Tang, S. Maniu, and X. S. Yang. Scalable algorithms for nearest-neighbor joins on big trajectory data. *IEEE Trans. Knowl. Data Eng.*, 28(3):785–800, 2016.

[30] N. Ferreira, J. T. Klosowski, C. E. Scheidegger, and C. T. Silva. Vector field k-means: Clustering trajectories by fitting multiple vector fields. *Comput. Graph. Forum*, 32(3):201–210, 2013.

[31] E. Frentzos, K. Gratsias, and Y. Theodoridis. Index-based most similar trajectory search. In *ICDE*, pages 816–825, 2007.

[32] S. Fries, B. Boden, G. Stepien, and T. Seidl. Phidj: Parallel similarity self-join for high-dimensional vector data with mapreduce. In *ICDE*, pages 796–807, 2014.

[33] S. Gaffney and P. Smyth. Trajectory clustering with mixtures of regression models. In *SIGKDD*, pages 63–72, 1999.

[34] F. García-García, A. Corral, L. Iribarne, M. Vassilakopoulos, and Y. Manolopoulos. Enhancing spatialhadoop with closest pair queries. In *ADBIS*, pages 212–225, 2016.

[35] F. Giannotti, M. Nanni, D. Pedreschi, F. Pinelli, C. Renso, S. Rinzivillo, and R. Trasarti. Unveiling the complexity of human mobility by querying and mining massive trajectory data. *VLDB J.*, 20(5):695–719, 2011.

[36] F. Giannotti and D. Pedreschi, editors. *Mobility, Data Mining and Privacy - Geographic Knowledge Discovery.* Springer, 2008.

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

[38] S. Guha, R. Rastogi, and K. Shim. Cure: An efficient clustering algorithm for large databases. *Inf. Syst.*, 26(1):35–58, 2001.

[39] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Indexing spatiotemporal archives. *VLDB J.*, 15(2):143–164, 2006.

[40] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *VLDB*, pages 562–573, 1995.

[41] C. Hu, X. Kang, N. Luo, and Q. Zhao. Parallel clustering of big data of spatio-temporal trajectory. In *ICNC*, pages 769–774, 2015.

[42] C. Hung, W. Peng, and W. Lee. Clustering and aggregating clues of trajectories for mining trajectory patterns and routes. *VLDB J.*, 24(2):169–192, 2015.

[43] E. H. Jacox and H. Samet. Metric space similarity joins. *ACM Trans. Database Syst.*, 33(2):7:1–7:38, 2008.

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

[45] P. Kalnis, N. Mamoulis, and S. Bakiras. On discovering moving clusters in spatio-temporal data. In *SSTD*, pages 364–381, 2005.

[46] M. Kornacker. High-performance extensible indexing. In *VLDB*, pages 699–708, 1999.

[47] D. Kumar, H. Wu, S. Rajasegarar, C. Leckie, S. Krishnaswamy, and M. Palaniswami. Fast and scalable big data trajectory clustering for understanding urban mobility. *IEEE Trans. Intelligent Transportation Systems*, 19(11):3709–3722, 2018.

[48] P. Laube, S. Imfeld, and R. Weibel. Discovering relative motion patterns in groups of moving point objects. *IJGIS*, 19(6):639–668, 2005.

[49] J. Lee, J. Han, and K. Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD*, pages 593–604, 2007.

[50] Y. Li, J. Bailey, and L. Kulik. Efficient mining of platoon patterns in trajectory databases. *Data Knowl. Eng.*, 100:167–187, 2015.

[51] Z. Li, B. Ding, J. Han, and R. Kays. Swarm: Mining relaxed temporal moving object clusters. *PVLDB*, 3(1):723–734, 2010.

[52] Z. Li, M. Ji, J. Lee, L. A. Tang, Y. Yu, J. Han, and R. Kays. Movemine: mining moving object databases. In *SIGMOD*, pages 1203–1206, 2010.

[53] Z. Li, J. Lee, X. Li, and J. Han. Incremental clustering for trajectories. In *DASFAA*, pages 32–46, 2010.

[54] W. Luo, H. Tan, H. Mao, and L. M. Ni. Efficient similarity joins on massive high-dimensional datasets using mapreduce. In *MDM*, pages 1–10, 2012.

[55] B. Morris and M. M. Trivedi. Learning trajectory patterns by clustering: Experimental studies and comparative evaluation. In *IEEE CVPR*, pages 312–319, 2009.

[56] M. Nanni and D. Pedreschi. Time-focused clustering of trajectories of moving objects. *J. Intell. Inf. Syst.*, 27(3):267–289, 2006.

[57] A. Nanopoulos, Y. Theodoridis, and Y. Manolopoulos. Indexed-based density biased sampling for clustering applications. *Data Knowl. Eng.*, 57(1):37–63, 2006.

[58] J. Ni and C. V. Ravishankar. Indexing spatio-temporal trajectories with efficient polynomial approximations. *IEEE Trans. Knowl. Data Eng.*, 19(5):663–678, 2007.

[59] F. Orakzai, T. Calders, and T. B. Pedersen. Distributed convoy pattern mining. In *IEEE MDM*, pages 122–131, 2016.

[60] F. Orakzai, T. Calders, and T. B. Pedersen. k/2-hop: Fast mining of convoy patterns with effective pruning. *PVLDB*, 12(9):948–960, 2019.

[61] C. Panagiotakis, E. Kokinou, and F. Vallianatos. Automatic p-phase picking based on local-maxima distribution. *IEEE Trans. Geoscience and Remote Sensing*, 46(8):2280–2287, 2008.

[62] C. Panagiotakis, N. Pelekis, I. Kopanakis, E. Ramasso, and Y. Theodoridis. Segmentation and sampling of moving object trajectories based on representativeness. *IEEE Trans. Knowl. Data Eng.*, 24(7):1328–1343, 2012.

[63] C. Panagiotakis and G. Tziritas. A speech/music discriminator based on RMS and zero-crossings. *IEEE Trans. Multimedia*, 7(1):155–166, 2005.

[64] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *SIGMOD*, pages 259–270, 1996.

[65] N. Pelekis, G. L. Andrienko, N. V. Andrienko, I. Kopanakis, G. Marketos, and Y. Theodoridis. Visually exploring movement data via similarity-based analysis. *J. Intell. Inf. Syst.*, 38(2):343–391, 2012.

[66] N. Pelekis, E. Frentzos, N. Giatrakos, and Y. Theodoridis. HERMES: aggregative LBS via a trajectory DB engine. In *SIGMOD*, pages 1255–1258, 2008.

[67] N. Pelekis, I. Kopanakis, E. E. Kotsifakos, E. Frentzos, and Y. Theodoridis. Clustering uncertain trajectories. *Knowl. Inf. Syst.*, 28(1):117–147, 2011.

[68] N. Pelekis, I. Kopanakis, C. Panagiotakis, and Y. Theodoridis. Unsupervised trajectory sampling. In *ECML PKDD*, pages 17–33, 2010.

[69] N. Pelekis, P. Tampakis, M. Vodas, C. Doulkeridis, and Y. Theodoridis. On temporal-constrained sub-trajectory cluster analysis. *Data Min. Knowl. Discov.*, 31(5):1294–1330, 2017.

[70] N. Pelekis, P. Tampakis, M. Vodas, C. Panagiotakis, and Y. Theodoridis. In-dbms sampling-based sub-trajectory clustering. In *EDBT*, pages 632–643, 2017.

[71] N. Pelekis and Y. Theodoridis. *Mobility Data Management and Exploration.* Springer, 2014.

[72] N. Pelekis and Y. Theodoridis. *Mobility Data Management and Exploration.* Springer, 2014.

[73] P. Petrou, P. Nikitopoulos, P. Tampakis, A. Glenis, N. Koutroumanis, G. M. Santipantakis, K. Patroumpas, A. Vlachou, H. V. Georgiou, E. Chondrodima, C. Doulkeridis, N. Pelekis, G. L. Andrienko, F. Patterson, G. Fuchs, Y. Theodoridis, and G. A. Vouros. ARGO: A big data framework for online trajectory prediction. In *SSTD*, pages 194–197, 2019.

[74] P. Petrou, P. Tampakis, H. Georgiou, N. Pelekis, and Y. Theodoridis. Online long-term trajectory prediction based on mined route patterns. In *MASTER workshop in conjuction with ECML/PKDD*, 2019.

[75] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches to the indexing of moving object trajectories. In *VLDB*, pages 395–406, 2000.

[76] C. Ray, R. Dreo, E. Camossi, A.-L. Jousselme, and C. Iphar. Heterogeneous integrated dataset for maritime intelligence, surveillance, and reconnaissance. *Data in Brief*, page 104141, 2019.

[77] S. Ray, B. Simion, A. D. Brown, and R. Johnson. Skew-resistant parallel in-memory spatial join. In *SSDBM*, pages 6:1–6:12, 2014.

[78] T. Seidl, S. Fries, and B. Boden. MR-DSJ: distance-based self-join for large-scale vector data analysis with mapreduce. In *DBIS*, pages 37–56, 2013.

[79] K. Seki, R. Jinno, and K. Uehara. Parallel distributed trajectory pattern mining using hierarchical grid with mapreduce. *IJGHPC*, 5(4):79–96, 2013.

[80] S. Shang, L. Chen, Z. Wei, C. S. Jensen, K. Zheng, and P. Kalnis. Trajectory similarity join in spatial networks. *PVLDB*, 10(11):1178–1189, 2017.

[81] S. Shang, L. Chen, Z. Wei, C. S. Jensen, K. Zheng, and P. Kalnis. Parallel trajectory similarity joins in spatial networks. *VLDB J.*, 27(3):395–420, 2018.

[82] Z. Shang, G. Li, and Z. Bao. DITA: distributed in-memory trajectory analytics. In *SIGMOD*, pages 725–740, 2018.

[83] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *PVLDB*, 8(13):2110–2121, 2015.

[84] S. Shohdy, Y. Su, and G. Agrawal. Load balancing and accelerating parallel spatial join operations using bitmap indexing. In *HiPC*, pages 396–405, 2015.

[85] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *MSST*, pages 1–10, 2010.

[86] Y. N. Silva and J. M. Reed. Exploiting mapreduce-based similarity joins. In *SIGMOD*, pages 693–696, 2012.

[87] Y. N. Silva, J. M. Reed, and L. M. Tsosie. Mapreduce-based similarity join for metric spaces. In *Cloud-I*, page 3, 2012.

[88] N. Ta, G. Li, Y. Xie, C. Li, S. Hao, and J. Feng. Signature-based trajectory similarity join. *IEEE Trans. Knowl. Data Eng.*, 29(4):870–883, 2017.

[89] P. Tampakis, C. Doulkeridis, N. Pelekis, and Y. Theodoridis. Distributed subtrajectory join on massive datasets. *ACM Trans. Spatial Algorithms and Systems*, To appear.

[90] P. Tampakis, N. Pelekis, N. V. Andrienko, G. L. Andrienko, G. Fuchs, and Y. Theodoridis. Time-aware sub-trajectory clustering in hermes@postgresql. In *ICDE*, pages 1581–1584, 2018.

[91] P. Tampakis, N. Pelekis, C. Doulkeridis, and Y. Theodoridis. Scalable distributed subtrajectory clustering. In *IEEE International Conference on Big Data*, 2019.

[92] L. A. Tang, Y. Zheng, J. Yuan, J. Han, A. Leung, C. Hung, and W. Peng. On discovery of traveling companions from streaming trajectories. In *ICDE*, pages 186–197, 2012.

[93] Y. Tao and D. Papadias. Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *VLDB*, pages 431–440, 2001.

[94] Y. Theodoridis, M. Vazirgiannis, and T. K. Sellis. Spatio-temporal indexing for large multimedia applications. In *ICMCS*, pages 441–448, 1996.

[95] R. Trasarti, R. Guidotti, A. Monreale, and F. Giannotti. Myway: Location prediction via mobility profiling. *Inf. Syst.*, 64:350–367, 2017.

[96] M. R. Vieira, P. Bakalov, and V. J. Tsotras. On-line discovery of flock patterns in spatio-temporal data. In *ACM SIGSPATIAL*, pages 286–295, 2009.

[97] M. Vlachos, D. Gunopulos, and G. Kollios. Discovering similar multidimensional trajectories. In *ICDE*, pages 673–684, 2002.

[98] M. Vodas. Building an efficient moving object database engine. Master's thesis, University of Piraeus, 3 2013.

[99] F. Wu, T. K. H. Lei, Z. Li, and J. Han. Movemine 2.0: Mining object relationships from movement data. *PVLDB*, 7(13):1613–1616, 2014.

[100] D. Xie, F. Li, and J. M. Phillips. Distributed trajectory similarity search. *PVLDB*, 10(11):1478–1489, 2017.

[101] H. Xu, Y. Zhou, W. Lin, and H. Zha. Unsupervised trajectory clustering via adaptive multi-kernel-based shrinkage. In *ICCV*, pages 4328–4336, 2015.

[102] G. Yuan, P. Sun, J. Zhao, D. Li, and C. Wang. A review of moving object trajectory clustering algorithms. *Artif. Intell. Rev.*, 47(1):123–144, 2017.

[103] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *USENIX workshop n conjuction with HotCloud*, 2010.

[104] D. Zeinalipour-Yazti, S. Lin, and D. Gunopulos. Distributed spatio-temporal similarity search. In *CIKM*, pages 14–23, 2006.

[105] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu. SJMR: parallelizing spatial join with mapreduce on clusters. In *CLUSTER*, pages 1–8, 2009.

[106] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: an efficient data clustering method for very large databases. In *SIGMOD*, pages 103–114, 1996.

[107] K. Zheng, Y. Zheng, N. J. Yuan, and S. Shang. On discovery of gathering patterns from trajectories. In *ICDE*, pages 242–253, 2013.

[108] K. Zheng, Y. Zheng, N. J. Yuan, S. Shang, and X. Zhou. Online discovery of gathering patterns over trajectories. *IEEE Trans. Knowl. Data Eng.*, 26(8):1974–1988, 2014.

[109] Y. Zheng. Trajectory data mining: An overview. *ACM TIST*, 6(3):29:1–29:41, 2015.

[110] Y. Zheng, X. Xie, and W. Ma. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data Eng. Bull.*, 33(2):32–39, 2010.

[111] E. Zimányi, M. A. Sakr, A. Lesuisse, and M. S. Bakli. Mobilitydb: A mainstream moving object database system. In *SSTD*, pages 206–209, 2019.

[112] N. Zygouras and D. Gunopulos. Corridor learning using individual trajectories. In *MDM*, pages 155–160, 2018.