THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

A Distributed In-Memory Database System for Large-Scale Spatial-Temporal Trajectory Data

Douglas Alves Peixoto

B.Sc. (Computer Science)

A thesis submitted for the degree of Doctor of Philosophy at The University of Queensland in 2018

School of Information Technology and Electrical Engineering.

Abstract

Spatial-temporal trajectory data contains rich information about moving objects and phenomena, hence have been widely used for a great number of real-world applications. However, the ubiquity and complexity of spatial-temporal trajectory data has made it challenging to efficiently store, process, and query such data. Furthermore, the increasing number of users also challenges the ability of trajectory-based services and analytics to handle the query workload and response to multiple requests in a satisfactory time.

Over the last few years, a new class of systems has emerged to handle large amounts of data in an efficient manner, referred as distributed in-memory database systems. These systems were designed to overcome the difficulties to scale traditional structured and unstructured data loads that some applications have to handle. Spark has became the framework of choice for large-scale low-latency data processing using distributed in-memory computation. However, Spark-based systems still lack the ability to handle several trajectory database tasks in a memory-wise manner. Some desirable feature of trajectory database systems include, data preparation and preprocessing, large-scale data storage and retrieval, and multi-user concurrent query processing. Providing a full-fledged system architecture supporting these features is challenging, and yet an issue. Firstly, trajectories are unstructured data types, coupled with spatial and temporal attributes, and organized in a sequential manner, which is hard to fit into traditional relational and spatial database systems; furthermore, trajectory data is available in a myriad of formats, each of which contains its own data schema and attributes. Moreover, trajectory datasets are highly skew and inaccurate, due to hotspots, transmission errors, and collecting devices inaccuracy, for instance. In addition, since Spark is a distributed parallel framework, we must account for data partitioning and load-balancing. In spatial and spatial-temporal databases, balanced data partitioning structures are built in a dynamic fashion as the data is consumed, nevertheless, Spark provides a read-only data structure that does not directly support adaptive partitioning after the partitioning model is constructed. Finally, data storage and query processing on top of Spark should be memory-wise, since the datasets may be too large to comfortably fit in the cluster memory; moreover, memory space may be wasted by storing unnecessary data partitions. Optimizing load-balancing and memory usage are essential to a good Spark application.

Therefore, driven by the increasing interest in scalable and efficient systems for trajectory-based analytics, we propose a distributed in-memory database system for memory-wise storage and scalable processing of spatial-temporal trajectory data, with low query latency and high throughput. We build our system on top of the Spark MapReduce framework, which provides an in-memory and fault-tolerant environment for distributed parallel processing of large-scale data. Existing works on spatial data in MapReduce, however, either lack support for spatial-temporal trajectory data, or only provide disk-based storage with costly I/O, which negatively affects query performance. Furthermore, none of the state-of-the-art applications address the problem of memory-wise utilization, which is the main drawback of in-memory based frameworks such as Spark. In this thesis we propose new features to the Spark framework, in order to provide native support for spatial-temporal trajectory data, with

low latency, high throughput, and memory-wise storage.

Our architecture follows a complete framework for trajectory data storage and processing, with trajectory data preparation, data preprocessing, data storage, and concurrent query processing. Firstly, we provide a novel model for trajectory data representation, and a system for loading, parsing, integration, and compression of trajectory data. Secondly, we introduce a novel framework for trajectory preprocessing using map-matching on top of Spark, in order to achieve data quality by means of data cleaning and simplification. Finally we introduce two novel approaches for data storage and multi-user trajectory query processing on top of Spark. In the first approach, we proposed a novel partitioning and storage methods focused on distance-based queries; in addition, we provide a system for trajectory distance measures evaluation, due to the extensive number of techniques available. In the second approach, we propose a novel memory-wise and workload-aware system for trajectory data storage, focused on data retrieval and spatial-temporal queries over large scale trajectory data; a key feature of our system is the ability to identify query hotspots, and exchange data between main-memory and disk based on the query workload, yet leveraging the scalability, fault-tolerance, efficiency, and concurrency control features of Spark.

Our extensive experiments demonstrate that our system architecture is efficient on integrating, cleaning, and storing large-scale trajectory data on top of Spark, in a distributed and memory-wise manner, addressing the Spark's limitations. Furthermore, experiments demonstrates the superiority of our approach in processing traditional and complex trajectory data queries, outperforming the state-of-the-art systems in throughput and memory usage. Although the efforts of current techniques provide a good starting point for trajectory data management on top of Spark, they are unable to provide all the features of our work. The superiority of our architecture comes from the research and development of both novel and state-of-the-art techniques for trajectory data management, using a well established framework for large-scale data applications. We believe our system will support scientists and professionals working with large-scale trajectory-based applications. For the best of our knowledge, this is the first work to cover all this range of important functionalities for large-scale trajectory data.

Declaration by author

This thesis is composed of my original work, and contains no material previously published or written by another person except where due reference has been made in the text. I have clearly stated the contribution by others to jointly-authored works that I have included in my thesis.

I have clearly stated the contribution of others to my thesis as a whole, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, financial support and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my higher degree by research candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that an electronic copy of my thesis must be lodged with the University Library and, subject to the policy and procedures of The University of Queensland, the thesis be made available for research and study in accordance with the Copyright Act 1968 unless a period of embargo has been approved by the Dean of the Graduate School.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material. Where appropriate I have obtained copyright permission from the copyright holder to reproduce material in this thesis and have sought permission from co-authors for any jointly authored works included in the thesis.

Publications During Candidature

Conference Papers:

- Douglas Alves Peixoto and Nguyen Quoc Viet Hung. "Scalable and Fast Top-k Most Similar Trajectories Search using MapReduce In-Memory". In *Australasian Database Conference (ADC)*, 2016.

- Douglas Alves Peixoto, Xiaofang Zhou, Nguyen Quoc Viet Hung, Dan He, Bela Stantic. "A System for Spatial-Temporal Trajectory Data Integration and Representation". In *International Conference on Database Systems for Advanced Applications (DASFAA)*, 2018.

- Douglas Alves Peixoto, Han Su, Nguyen Quoc Viet Hung, Bela Stantic, Bolong Zheng, Xiaofang Zhou. "Concept for Evaluation of Techniques for Trajectory Distance Measures". In *IEEE International Conference on Mobile Data Management (MDM)*, 2018.

Journal Papers:

- Douglas Alves Peixoto, Xiaofang Zhou, Nguyen Quoc Viet Hung, Bolong Zheng. "A Framework for Parallel Map-Matching at Scale using Spark". To appear *Distributed and Parallel Databases journal (DAPD)*, 2017.

Publications included in this thesis

Incorporated conference papers:

1. Douglas Alves Peixoto and Nguyen Quoc Viet Hung. "Scalable and Fast Top-k Most Similar Trajectories Search using MapReduce In-Memory". In *Australasian Database Conference (ADC)*, 2016. – Incorporated as Chapter 5.

Contributor	Statement of contribution
Author D. A. Peixoto (Candidate)	Conceptual theoretical design (80%). Theoretical simulations, measurements, data analysis and implementation (100%). Wrote the paper (80%).
Author N. Q. V. Hung	Conceptual theoretical design (20%) Wrote and edited paper (20%).

Incorporated journal papers:

1. Douglas Alves Peixoto, Xiaofang Zhou, Nguyen Quoc Viet Hung, Bolong Zheng. "A Framework for Parallel Map-Matching at Scale using Spark". Submitted to *ACM SIGSPATIAL Interna-*

tional Conference on Advances in Geographic Information Systems, 2018. – Incorporated as Section 3.2.

Contributor	Statement of contribution
Author D. A. Peixoto (Candidate)	Conceptual theoretical design (70%). Theoretical simulations, measurements, data analysis and implementation (100%). Wrote the paper (80%).
Author N. Q. V. Hung	Conceptual theoretical design (10%). Wrote and edited paper (10%).
Author X. Zhou	Conceptual theoretical design (20%). Checking theoretical simulations and correcting the manuscript.
Author B. Zheng	Checking theoretical simulations and correcting the manuscript. Wrote the paper (10%).

Incorporated demonstration papers:

1. Douglas Alves Peixoto, Xiaofang Zhou, Nguyen Quoc Viet Hung, Dan He, Bela Stantic. "A System for Spatial-Temporal Trajectory Data Integration and Representation". In *International Conference on Database Systems for Advanced Applications (DASFAA)*, 2018. – Incorporated as Section 3.1

Contributor	Statement of contribution
Author D. A. Peixoto (Candidate)	Conceptual theoretical design (80%). Theoretical simulations, measurements, data analysis and implementation (90%). Wrote the paper (90%).
Author D. He	Theoretical simulations, measurements, data analysis and implementation (10%).
Author N. Q. V. Hung	Wrote and edited paper (10%) Checking theoretical simulations and correcting the manuscript.
Author X. Zhou	Conceptual theoretical design (20%). Checking theoretical simulations and correcting the manuscript.
Author B. Stantic	Checking theoretical simulations and correcting the manuscript.

2. Douglas Alves Peixoto, Han Su, Nguyen Quoc Viet Hung, Bela Stantic, Bolong Zheng, Xiaofang Zhou. "Concept for Evaluation of Techniques for Trajectory Distance Measures". In *IEEE*

International Conference on Mobile Data Management (MDM), 2018. – Incorporated as Appendix A

Contributor	Statement of contribution
Author D. A. Peixoto (Candidate)	Conceptual theoretical design (70%). Theoretical simulations, measurements, data analysis and implementation (70%). Wrote the paper (80%).
Author H. Su	Conceptual theoretical design (10%). Theoretical simulations, measurements, Data analysis and implementation (20%). Wrote the paper (10%).
Author N. Q. V. Hung	Wrote and edited paper (10%). Checking theoretical simulations and correcting the manuscript.
AuthorB. Zheng	Theoretical simulations, measurements, data analysis and implementation (10%). Checking theoretical simulations and correcting the manuscript.
Author B. Stantic	Checking theoretical simulations and correcting the manuscript.
Author X. Zhou	Conceptual theoretical design (20%). Checking theoretical simulations and correcting the manuscript.

Submitted manuscripts included in this thesis

No manuscripts submitted for publication.

Contributions by others to the thesis

No contributions by others.

Statement of parts of the thesis submitted to qualify for the award of another degree

No works submitted towards another degree have been included in this thesis.

Research Involving Human or Animal Subjects

No animal or human subjects were involved in this research.

Acknowledgements

Firstly, I would like to thank God for my life and health, for his grace and blessings, which made it possible for me to be here and achieve my goals. Secondly, I would like express my deep love and gratitude towards my family, in special my mother and beloved wife, for their priceless support, love and care. I also would like to extend my gratitude to all my friends, for their support when I most needed, in special, my friend Hung, for his kindness and guidance.

My sincere gratitude to my main advisor Prof. Xiaofang Zhou, for the continuous support of my Ph.D study and related research, and for his immense knowledge. I also extend my gratitude for all advisors I ever had, who paved the way to my Ph.D study, thank you, Dr. Jugurta Lisboa-Filho, Dr. Andre Gustavo dos Santos, Dr. Fabio Ribeiro Cerqueira, Dr. Lexing Xie, Dr. Hung Nguyen Quoc Viet, and Prof. Bela Stantic.

A special thanks to all my research group mates and the DKE academic staff, as well as the University of Queensland staff and Griffith University staff, for their support, and to make my Ph.D study time enjoyable.

Last but not least, for my government, sponsors, and the Brazilian people for their financial support; and for the people of Australia for their respect and friendship.

"All things work together for good to them that love God. (Romans 8:28)."

Financial Support

This research was supported by the Brazilian National Council for Scientific and Technological Development (CNPq), and The University of Queensland Graduate School.

Keywords

Trajectory, Distributed database, Parallel systems, In-Memory database, Spark, MapReduce, Data preparation, Data preprocessing, Large-scale data storage and processing.

Australian and New Zealand Standard Research Classifications (ANZSRC)

- ANZSRC code: 080302, Computer System Architecture, 60%
- ANZSRC code: 080304, Concurrent Programming, 30%
- ANZSRC code: 080309, Software Engineering, 10%

Fields of Research (FoR) Classification

- FoR code: 0803, Computer Software, 40%
- FoR code: 0804, Data Format, 20%
- FoR code: 0805, Distributed Computing, 40%

Contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Research Proposal

Due to the pervasiveness and high availability of GPS-equipped devices, the efficient and reliable storage and processing of spatial-temporal trajectory data, are necessary, and play a key role in trajectory data-driven applications, which often demand querying and processing complex algorithms over large-scale datasets. Trajectory data contains rich information about moving objects, thus have been used in several real-life applications, from recommendation systems to moving-objects pattern analysis. Some applications include, routes recommendation based on the path use frequency in a specified time period [97]; time prediction of public transportation by means of bus routes analysis in a given area [34]; find points of interest (POI), such as touristic attractions, hotels or restaurants, and the popular routes in a given spatial region [32] [196]; identify gathering patterns in specific urban areas for recommendation systems [194]; support travelers to plan a trip itinerary to an given unfamiliar location (trip recommendation) [197] [198]; predict the best transportation mode based on the time taken and frequency of moving objects within a given area [196]; drivers pastern analysis, city traffic planing, dynamic event identification, human interaction, and so on [33] [36] [58] [62] [198].

1.1.1 Motivation and Importance

Database systems dedicate their efforts towards reliable and efficient data storage and fast query performance. The complexity of the query highly depends on the type of data in the database. However, the massive amount of GPS data available, as well as the increasing number of users of location-based services, challenge the efficiency and scalability of conventional centrally-disk-based database systems, since they have been optimized for I/O, they face great performance deterioration as the dataset grows [168]. For instance, one of our trajectory datasets contains over 250 billions GPS records (over 1.2TB data) collected in only one month from three different cities in China. Therefore, providing a scalable system for trajectory data management is important due to the increasing amount of GPS data available, and the high value of information aggregated with trajectory data.

Furthermore, trajectory data are a complex and unstructured data types, coupled with spatial,

temporal, and semantic data organized in a sequential manner, which makes it difficult for conventional database systems to manage such data. Moreover, several data sources and collection devices are available, and they collect and store their data in many different formats; thus, this heterogeneity of formats also makes it challenging for conventional database systems to model the data. In addition, trajectory data is very susceptible to noise, due to errors and inaccuracy of collection devices. Consequently, to model, manage, and query such large amount of heterogeneous and complex spatial-temporal data is still a challenge. Therefore, since data quality can negatively affect data analytics, trajectory data preprocessing is very important to improve the quality and add more value to data analytics and trajectory data-based systems. Moreover, data representation and integration are necessary, since trajectory data are available in a great number of sources and heterogeneous formats.

Moreover, with the increasing demand for low-latency services over large-scale data, as well as the increasing number of users of spatially-aware services, a trajectory database system should be able to serve multiple requests over large-scale datasets, providing good scalability, high throughput, and fast query response. For instance, Google revealed that more than 1 Billion users access Google Maps every month, and 30% of Google searches have local intent or geographic aspect [1]. To address this important problem, an alternative is to use distributed parallel computation, while storing data in main-memory to reduce I/O cost, with reliability and fault-tolerance. However, distributed systems should account for data partitioning and workload-balancing. Furthermore, a multi-user and in-memory database system should be resource-wise, since memory availability can be a bottleneck, specially in commodity hardwares. Load balancing and memory usage are key points in distributed in-memory applications over multi-user environments.

1.1.2 State-of-the-art and Limitations

Existing distributed systems for spatial data employ balanced partitioning structures, such as Quadtree, kd-Tree, and Rtree, to organize the data space into partitions of spatially close objects, in order to support distributed storage and parallel processing, with workload balancing. However, current distributed systems for spatial and spatial-temporal data are unable to provide all the features previously discussed. For instance, SpatialHadoop [48], HadoopGIS [6], MD-HBase [103], ScalaGiST [94], and AQWA [12] [11] are disk-based systems and do not support trajectory data. GeoSpark [175], SparkGIS [16], and SpatialSpark [174] provide an in-memory-based system for spatial data on top of Spark; however, they do not provide support for trajectory data. CloST [146], TRUSTER [172] and PRADASE [98] provides support for trajectory data storage and query using spatial partitioning (i.e. grid and quadtree), however they are disk-based systems thus do not address memory usage and storage.OceanST [178] provides a distributed in-memory storage for trajectories on top of Spark, however it does not consider the query workload (i.e. query hotspots), and assumes the entire data fits in the cluster memory.Finally, Simba [168] is a Spark-based framework for spatial data analytics, and supports spatial indexing and spatial operations natively, such as range query, k-NN, distance

[1] Source: Google Research, London, 2015

join, and k-NN join. Simba extends the SQL grammar for spatial predicates so that users can express spatial objects and operations in a SQL-like fashion (e.g. `POINT, RANGE, KNN, DISTANCE JOIN`). However, Simba still does not provide native support for trajectory data. Furthermore, none of the aforementioned systems address the problems of trajectory data integration, representation, and preprocessing. Table 1.1 summarizes the related works and their main features and limitations. In the literature review, Chapter 2, we provide a thorough review and comparison of these systems, as well as a discussion on their contributions and limitations.

Table 1.1: Related systems and features.

System	Memory Based	Trajectory Data	Workload Aware	Pre-processing
SpatialHadoop [48]	×	×	×	×
Hadoop-GIS [6]	×	×	×	×
MD-HBase [103]	×	×	×	×
AQWA [12]	×	×	✓	×
ScalaGiST [94]	×	×	✓	×
Simba [168]	✓	×	×	×
GeoSpark [175]	✓	×	×	×
SpatialSpark [174]	✓	×	×	×
SparkGIS [16]	✓	×	×	×
OceanST [178]	✓	✓	×	×
TRUSTER [172]	×	✓	×	×
PRADASE [98]	×	✓	×	×
CloST [146]	×	✓	×	×
Our Proposal	✓	✓	✓	✓

In summary, these are the main limitations of existing works, and the main desirable features we identified for large-scale GPS trajectory data management system.

- **Data Representation and Integration:** Since raw GPS trajectory data are available in many different sources and formats, a system should be able to load and integrate data from different sources into a single data representation. Although conceptual models for trajectory data exist [138], it is challenging to interpret and integrate trajectory data from the multitude of textual formats and sensors available, and it is still an issue. Furthermore, some devices collect data with high sampling rates, thus, trajectory data compression and simplification is also necessary to reduce storage consumption.

- **Trajectory Data Preprocessing:** Raw trajectory data can be noisy and inaccurate, therefore data preprocessing techniques, such as map-matching, are necessary to ensure good data quality. Accuracy-driven algorithms such as [71] [92] [101] [164], can achieve high accuracy, but are limited to small datasets, since they focus on the accuracy of the matching rather than its performance and scalability. Performance-based algorithms such as [72] [147] [150] [167], on the other hand, do not account for load balancing and memory usage, and are limited for disk-based computation.

- **Resource-Wise and Reliable Storage:** A database system for large-scale trajectory data should support distributed in-memory storage of spatial-temporal data, with fault-tolerance and load-balancing, in order to achieve scalability, efficiency, and reliability. Furthermore, the system should provide workload-aware data storage, with resource-wise utilization, in order to reduce memory consumption without considerably affecting the system's performance. Existing distributed systems, however, either lack resources- and memory-wise utilization, or do not consider memory storage.

- **Efficient Query Processing:** Furthermore, a system should provide high throughput and low latency queries, by leveraging parallel and concurrent query processing for multi-user applications, due to the increasing demand for real-time systems with multiple requests. Existing works on trajectory data management, however, either lack support for efficient trajectory data query processing, since they employ centrally based-processing, which affects performance, and are not scalable; or they do not consider multi-user environments and concurrency control.

We go beyond the state-of-the-art and propose a novel system that holds all mentioned features for large-scale GPS trajectory data. We leverage the distributed in-memory properties of the Spark framework, and introduce new features to Spark in order to achieve the aforementioned goals.

1.1.3 The Case for Spark

Spark [181] goes beyond distributed database systems, and can fill the gap between performance, scalability, and fault-tolerance, as well as efficient resources allocation of concurrent jobs. Spark provides a robust distributed data structure for MapReduce tasks in main-memory, and have been used in a handful number of data-intensive analytics [14] [105] [182] [185], including spatial databases [168] [175]. Spark also provides a SQL language engine to support relational data processing in a SQL-like fashion [14]. However, storage and processing of trajectory data using Spark is challenging, since Spark is not equipped for supporting sequential and spatial-temporal data in its core. Furthermore, since Spark is a in-memory-based framework, data storage and query processing on top of Spark should be memory-wise, for instance, the trajectory datasets may be too large to comfortably fit in the cluster memory. However, even though Spark possesses both in-memory and on-disk storage, the exchange of data from memory to disk is not based on the query workload, but in the memory availability. Optimizing load-balancing and memory usage are essential to a good Spark application.

Performance: Performance can be measured as the response time for a given piece of work or task submitted to the system. High performance systems are important in real-time applications in order to improve the response time and resources utilization of tasks that are either computationally heavy; or would spend large amounts of computational and hardware resources (e.g. CPU, memory network). In our project, performance improvement is achieved using Spark for in-memory data storage, parallel query processing, and spatial-aware partitioning in order to reduce the amount of data necessary to process a single query, as well as reduce network data exchange in the Spark cluster. In addition, our

Spark system uses a work-load aware storage to keep hot data in memory improving query response time and saving Spark memory usage, which also improves query throughput by improving resources availability.

System Throughput: Throughput can be measured as the maximum number of tasks (e.g. user queries, operations) a system can execute within a time interval, in another words, it's the processing rate of the system, and account for the sum of the data and responses that are delivered to all terminals in the cluster. Improving throughput plays a key role in multi-user system where large amount of user queries need to be executes and large amount of data needs to be moved through the network. In our system throughput enhancement is achieved using query scheduling and concurrency control, in addition we ensure balanced data partitioning and resource-wise utilization in order to increase resources availability in the Spark cluster, reducing memory usage and allowing more queries to be executed concurrently.

1.2 Contributions

Motivated by the growing interest in scalable and efficient systems for spatial-temporal data, and the high value of information aggregated to trajectory data, in this research project we extend the Spark framework, and introduce new features to the system's architecture in order to cover the state-of-the-art limitations. The ultimate goal is to provide a robust Spark-based system for large-scale trajectory data engineering and management. In summary, the main contributions and achievements of this research project are the following:

- **Trajectory Data Integration and Representation:** We designed a new spatial-temporal data loading and integration system. We propose a representation format for raw trajectory data (e.g. spatial-temporal attributes), in order to integrate data from different sources into a single format. The data loader is also responsible for data compression and to collect statistical information about the input datasets (i.e. Metadata). Further details can be found at Section 3.1 of this thesis, and in our published work [117].

- **Trajectory Data Preprocessing using Map-Matching:** We introduce a new trajectory preprocessing framework using Map-Matching in order to improve data quality, and provide data simplification. We provide an estimative of the distributed map-matching workload cost, in order to tune the framework parameters. In addition, we employ a safe boundary threshold for trajectory segmentation and replication to reduce uncertainty. Further details can be found at Section 3.2 of this thesis, and in our published work [119].

- **Workload-aware Trajectory Data Partitioning and Retrieval:** We employ a hierarchical Quadtree-based partitioning for load-balancing. We provide an estimative of the distributed spatial-temporal query workload cost, in order to tune the system parameters and optimize both data retrieval and sampling-based partitioning. In addition, we introduce a workload-aware

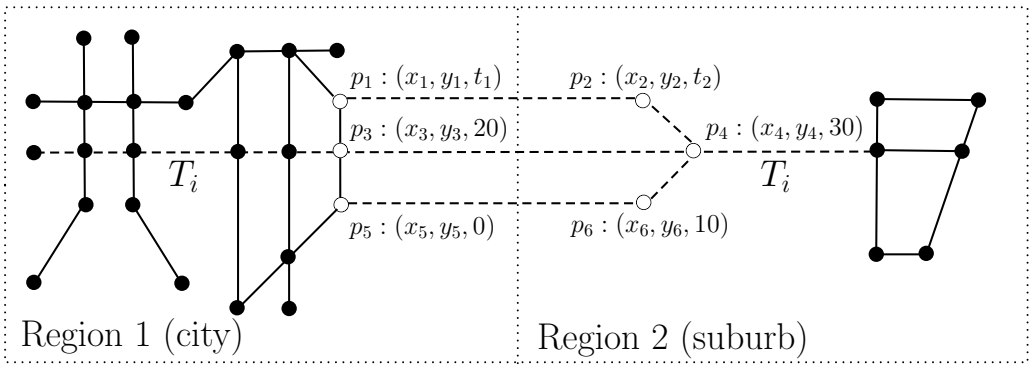


Figure 1.1: Road network example.

storage controller to Spark in order to reduce memory usage. We provide an *Active-Time Window* mechanism to adapt the data storage based on the query workload, such that only partitions containing high density query areas (i.e. query hotspots) are kept in memory. Further details can be found at Chapter 4 of this thesis, and in our published work [118].

- **Trajectory Distance-Based Partitioning:** We propose a second approach for data partitioning using a Voronoi-diagram based approach, focusing on k-NN trajectory queries. We propose a bulk-loading in-memory partitioning strategy based on Voronoi diagrams and time pages, named *Voronoi Pages*, to support multiple *k*-NN trajectory query in MR, and a spatial-temporal composite index, named *VSI* (Voronoi Spatial Index) and *TPI* (Time Page Index), to prune the search space and speed up trajectory similarity search. Further details can be found at Chapter 5 of this thesis, and in our published work [115]. In addition,

Even though our work is done on top of the Spark framework, achieving a robust in-memory-based system that provides all the aforementioned goals is not trivial, and demands significant research, design, implementation, and experimentation efforts. For the best of our knowledge, this is the first work to cover all this range of important functionalities for large-scale trajectory data.

1.2.1 Challenges

Trajectory data is difficult to fit into the Spark MapReduce computation model, since Spark does not natively support sequential and spatial-temporal data. Following we present the main challenges of trajectory data management on top of Spark.

- **Data Complexity and Temporal Dimension:** Since trajectory data are queried by both spatial and temporal attributes, temporal dimension must be taken into account [163]; for instance, consider the road network in Figure 1.1 connecting two spatial regions, there may be thousands of trajectories passing thought the road T_i, however the application may be interested in retrieving trajectories within a specific region and time period. However, a simply partitioning the data space may not suffice for some queries, for instance, imagine three trajectories passing through roads $T_1 : \{p_1, p_2\}$, $T_2 : \{p_3, p_4\}$ and $T_3 : \{p_5, p_6\}$, if we want to retrieve the Nearest-Neighbor

trajectory to T_1 within time $t = [0, 10]$, the application should return T_3 instead of T_2. Only grouping trajectories by spatial region in that case is not strict enough.

- **Skewness, Data Partitioning, and Load-Balancing:** Furthermore, since Spark is a distributed parallel framework, we must account for data partitioning and load-balancing. However, trajectory datasets are highly skew, for instance, in Figure 1.1 the density of moving object's trajectories passing through Region 1 (city region) is much larger than in Region 2 (suburb region). Therefore, we must provide a partitioning strategy as uniform as possible to avoid high dense partitions and load imbalance, yet keeping spatial proximity, which is a key factor in spatial data processing in MR [43]. However, balanced space partitioning structures should be built in a dynamic fashion as the data is consumed, nevertheless, Spark's RDD is a read-only data structure that does not directly support adaptive partitioning after the RDD is constructed. Thus, related works based on dynamic spatial partitioning such as [146] cannot be applied directly, since we need to find the best partitioning schema beforehand.

- **Query Workload and Memory Usage:** Moreover, since Spark is a in-memory-based framework, data storage and query processing on top of Spark should be memory-wise, for instance, the GPS trajectory datasets may be too large to comfortably fit in the cluster memory; moreover, as an effort to reduce main-memory consumption, the application should be able to react to changes in the query workload efficiently, since some spatial regions, such as urban areas, receive more query requests (hotspots), thus data records in such areas should receive priority for in-memory storage over least requested data. Optimizing load-balancing and memory usage are essential to a good Spark application.

1.3 System Architecture Overview

The main components of our system and their features, and how the components developed during this research are interrelated, are shown in Figure 1.2.

Since our system build on top of Spark, and therefore inherits the frameworks' functionalities. Indexing and storage are done in-memory on top of Spark's RDD structure. Least required data, however, is stored on disk (HDFS). Both Spark-based querying and storage are done using the MapReduce model. In order to fill the gaps previously described, we designed our system into four components to support trajectory data management on top of Spark, namely, (1) data representation and integration, (2) data preprocessing using map-matching, (3) workload-aware trajectory data storage and retrieval, (4) distance-based storage and similarity query processing. The components are briefly described in the next sections.

1.3.1 Trajectory Data Preparation and Preprocessing

Trajectory Data Loader: We designed a novel parallel system for trajectory data integration and representation, with support for synthetic trajectory generation, and trajectory data compression

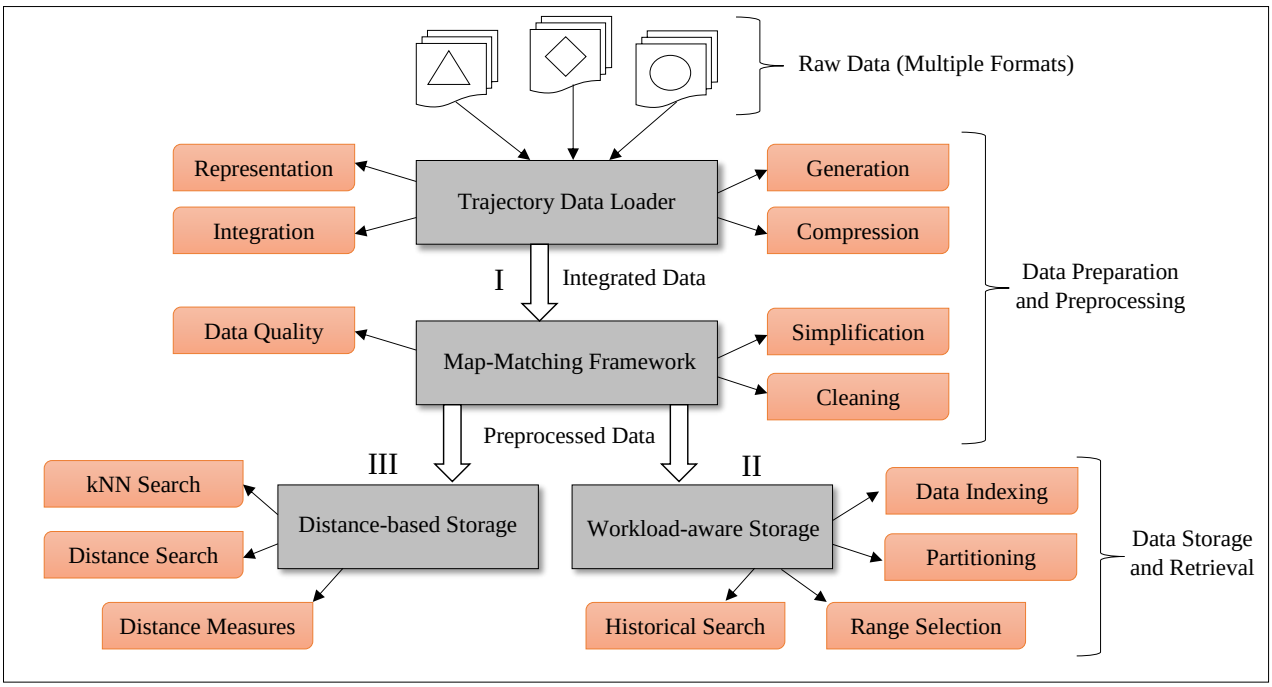


Figure 1.2: System Main Features.

(lossless *Delta* compression). This system provides templates for trajectory data representation (e.g. spatial-temporal attributes, textual attributes) providing a single data model for integration of different input datasets. Furthermore, this module is responsible to manage the system metadata, such as system and user information, and data statistics.

The application parses a given input data to a predefined output and compressed data format, and stores the formated data into any of the provided primary storage platforms, i.e. Local directory, MongoDB [100], and HDFS [68]. This allows our Spark system to process data from multiple datasets in a single storage platform, without the need of re-implementation. In order to represent and integrate trajectory data from different sources, models and schema, we introduce the *Trajectory Data Description Format (TDDF)*, a data description format for spatial-temporal trajectory data (Chapter 3 and [117]).

Map-Matching Framework We introduce a novel Spark-based framework to perform map-matching on the integrated data, in order to enhance data quality. We combine a sampling-based Quadtree space partitioning construction, and Spark-based computation in batches, to achieve horizontal scaling of map-matching, as well as reduce cluster memory usage. We also employ a safe spatial-boundary approach to preserve matching accuracy of boundary objects. In addition, a cost function for the distributed map-matching workload is provided. Our extensive experiments demonstrate that our framework is efficient and scalable to process map-matching on large-scale data (see Chapter 3.2 and [119]).

Figure 1.3 shows the workflow from the trajectory *Data Loader* system to the *Map-Matching Framework*.

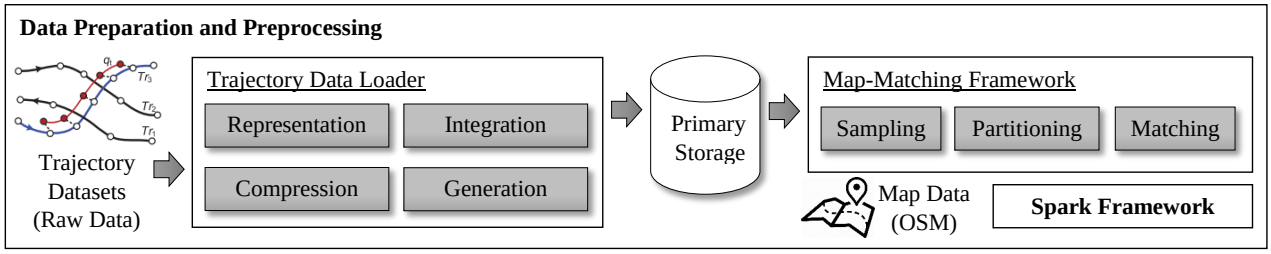


Figure 1.3: Preparation and Preprocessing Workflow.

1.3.2 Workload-aware Trajectory Storage

We designed a Spark-based system for scalable and memory-wise storage of GPS trajectory data, and low-latency workload-aware query processing with fault-tolerance. We exploit the in-memory nature and distributed parallel properties of Spark for scalable and low-latency trajectory data storage and processing (Chapter 4 and [118]).

We take advantage of the hierarchical partitioning of CloST [146] for trajectory data loading efficiency, and extend CloST to allow a memory-wise and workload-aware data storage and query on top of Spark. Since building a dynamic spatial index model from a large dataset can be cumbersome, and a data partitioning model must be provided to Spark beforehand, we address this limitation by providing a sampling-based quad-index construction using a cost-based model, and finally employ the sample-based model to Spark RDD partitioning.

In addition, we add a workload-aware storage mechanism to CloST in order to reduce memory usage. We provide an *Active-Time* mechanism to adapt the data storage based on the query workload, such that only RDD partitions containing high density query areas are kept in memory.

Finally, we provide an estimative of the distributed spatial-temporal query workload cost, in order to tune the system parameters and optimize both data retrieval and sampling-based partitioning.

1.3.3 Distance-Based Storage and Query

We proposed a novel parallel approach for the k-NN trajectories problem in a distributed and multi-user environment using Spark. k-NN trajectory is a complex an expensive operation, therefore, we proposed a space/time data partitioning based on Voronoi diagrams and time pages, named *Voronoi Pages*, in order to provide both spatial-temporal data organization and process decentralization, so that we are able to process multiple k-NN queries in parallel using Spark. We proposed a spatial-temporal composite index, named *Voronoi Spatial Index* (VSI) and *Time Page Index* (TPI), to prune the search space and speed up trajectory similarity search. Finally, we provide an algorithm to calculate the k-NN using our spatial-temporal index on top of Spark, with high throughput and fast query response (Chapter 5 and [115]).

Trajectory Distance Techniques Evaluation: In addition, we accomplished a survey on trajectory similarity measures, comparing and discussing the effectiveness of many classical algorithms to identify the similarity between trajectories with different characteristics, such as: trajectories with noise, trajectories with non-uniform sampling rate, trajectories with point shifting, different scales and different speed. We developed a benchmarking system containing a wide range of trajectory distance measure techniques, and a means to compare these techniques under different circumstances and parameters (Appendix A and [116]).

1.4 Thesis Organization

The remainder chapters of this thesis are organized as follows.

In Chapter 2 we give some background knowledge to help the reader, and present the literature review, including a thorough discussion and comparison of the related work, and a deep introduction of the domains related to this thesis.

In Chapter 3 we address the problems of trajectory data preparation and preprocessing. Firstly, we describe a novel system to represent and integrate trajectory data from different sources and formats. We introduce the *Trajectory Data Description Format (TDDF)*, a data description format for spatial-temporal trajectory data representation. The TDDF was designed based on a survey on several real GPS trajectory datasets, both public and private, accessible by our research groups. Then, based on the user-provided TDDF, our application loads and parses the input data into the integrated format in a compressed way. Our system also generates statistical information (Metadata) about the input datasets, which are used in the Spark algorithms, and can also be used to generate synthetic data for experimental purposes. Secondly, we introduce a novel framework for trajectory data preprocessing using parallel map-matching on top of Spark, for scalable and fast processing of offline map-matching in a distributed in-memory fashion. We provide an estimative of the distributed map-matching workload cost, in order to tune the system parameters and optimize the sampling-based data partitioning. We employ Quadtree partitioning for trajectory and map data. Our experiments demonstrates that Quadtree provides an efficient and fairly uniform space partitioning when compared with other commonly used dynamic structures, such as k-d Tree and STR-Tree, achieving better performance and scalability. Since building a dynamic spatial index model from a large dataset can be cumbersome, and a data partitioning model must be provided to Spark beforehand, we address this limitation by providing a sampling-based quad-index construction using a cost-based model. Finally, we co-partition both map and trajectory data using the quad-index model into the Spark's RDD. Once the map data is loaded, trajectory records can be matched independently, thus we provide a batch-based loading and processing of the input trajectory dataset to reduce distributed memory consumption, specially in situations where the cluster memory size is a constraint. Our experiments demonstrate that our approach can achieve efficient and scalable map-matching processing.

In Chapter 4 we introduce our workload-aware system for trajectory data storage and retrieval, we describe the features we developed in order to allow Spark to manage trajectory data in a memory-wise

manner, and low-latency workload-aware search with fault-tolerance. Firstly, we provide an estimative of the distributed spatial-temporal query cost, in order to tune the system parameters and optimize both data retrieval and sampling-based partitioning. We employ a hierarchical Quadtree based partitioning proposed in CloST [146], for it provides a fairly uniform partitioning of spatial-temporal trajectory records. We extend the CloST index to store the status of the data partitions, so that we can efficiently identify query hotspots. Since building a dynamic spatial index model from a large dataset can be cumbersome, and a data partitioning model must be provided to Spark beforehand, we address this limitation by providing a sampling-based quad-index construction, using a cost-based model, and finally employ the sample-based model to Spark RDD partitioning. In addition, the system can react to changes in the query workload in order to organize the storage level to reduce memory usage. We provide an *Active-Time Window* mechanism to adapt the storage level based on the query workload, such that only RDD partitions containing query hotspots are kept in memory. Finally, we provide an efficient data retrieval module for concurrent queries; in this chapter we focus on spatial-temporal range queries, for they are the most fundamental operations in trajectory databases.

In Chapter 5 we describe our contribution on trajectory data storage aiming distance-based queries using Spark. We propose a parallel approach to the k-NN trajectories problem in a distributed and multi-user environment using Spark. We propose a space/time data partitioning based on Voronoi diagrams and time pages, named Voronoi Pages, in order to provide both spatial-temporal data organization and process decentralization. In addition, we propose a spatial-temporal index to prune the search space, improve query latency and system throughput. Briefly, we uniformly partition the space into Voronoi cells using k-Means clustering, and each Voronoi cell into static temporal partitions (i.e. pages). Trajectories are split into sub-trajectories according to their spatial-temporal extent, such that each sub-trajectory is mapped to one Voronoi Page. We process a k-NN query in parallel in a *filter-and-refinement* manner, first filtering candidate pages using our proposed spatial-temporal index, and then running a precise check on each candidate page. Each process unit can manage a number of pages within a Spark RDD in parallel, and multiple concurrent queries can be served by Spark over its RDD. We perform extensive experiments to demonstrate the performance and scalability of our approach.

Finally, In Chapter 6 we present the conclusions of this thesis. The Appendix sections contain auxiliary material and additional research done during this thesis timeline.

Chapter 2

Literature Review

Our work is at the intersection of trajectory data management, distributed and parallel computation using MapReduce framework, and in-memory data storage and processing using Spark framework.

This literature review is organized as follows. In Section 2.1 we firstly give a background knowledge on trajectory data management, spatial partitioning, MapReduce, and Spark to help the reader. In Sections 2.1 and 2.2 we introduce some related work in Spatial and Trajectory data management respectively. In Section 2.4 we discuss the related work in distributed parallel computation, including a background knowledge and related work in the MapReduce framework (Section 2.4.2). In Section 2.5 we discuss in-memory BigData management, including a background knowledge and literature using the Spark MapReduce framework (Section 2.5.1). Finally, in Section 2.6 we introduce the related work in spatial and trajectory data management using MapReduce and Spark.

2.1 Spatial Data Management

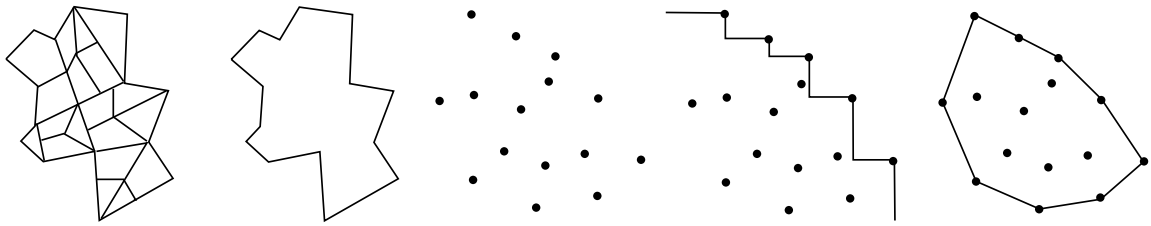
In recent years, the volume of spatial and spatial-temporal data available has grown exponentially, due to the easy access to inexpensive location-aware sensors. Spatial data has great commercial and social value; individuals and organizations rely on location-based services to make critical decisions every day. Therefore, the management of the increasing volume of spatial and spatial-temporal data has become an urgent issue.

A number of SQL and NoSQL GIS technologies have been developed with the purpose of collect, store, process, and share spatial data, such as Spatial Data Warehouses (SDW) [17] [110], Spatial Data Infrastructures (SDI) [60] [129], and Geographic DBMS (e.g. Oracle Spatial and PostGIS). However, the huge amount of unstructured and semi-structured spatial data available have increased the interest to incorporate spatial data into cloud environments and distributed databases, in order to maintain large-scale spatial data and support efficient query processing. In this section we introduce the main ideas on spatial data management.

2.1.1 Spatial Queries

An important component of spatial databases and GIS applications is the ability to process spatial operations and spatial queries. We can classify spatial operations and spatial queries into the main following categories:

- **Geometric Operations:** [37] operations based on the geometry and spatial dimension of the objects in the dataset (e.g. points, polygons, line segments). Usually return a number or create new geometries from the existing ones. This category includes operations such as: *line length, objects distance, shortest-distance, farthest-distance, polygon area, polygon union, skyline, convex hull, minimum bounding rectangle (MBR)*, etc. Figure 2.1 shows an example of three geometric operations over points and polygons.



(a) Input Polygons (b) Polygons Union (c) Input Points (d) Skyline (e) Convex Hull

Figure 2.1: Geometric operations example: (b) polygons union, (d) points set skyline, (e) points set convex hull.

- **Boolean Queries:** receives a predicate as argument, and evaluates the spatial objects for the given predicate. Usually returns one or many objects from the dataset. We divide this category into the two main sub-groups.
  - **Topology Based Queries:** [44] [122] evaluates the objects based on a spatial relationship as predicate. This category includes: *intersect, contains, disjoint, touch, overlap*, etc. Figure 2.2 shows an example of four topological queries over points, polygons and line segments.
  - **Distance Based Queries:** [95] [186] evaluate the objects based on proximity or spatial distance as predicate. This category includes queries such as: *nearest neighbors, reverse nearest neighbors, distance join, farthest pair, closest pair, distance search*, etc. Figure 2.3 shows an example of three distance-based queries over a points dataset.

Primitive queries such as *Range Selection* can be classified here as topology-based query, once the goal is to select spatial objects that overlap with a given query region. *Distance Join* and *k*-NN *Join*, on the other hand, can be classified as a distance-based queries.

The complexity of the query highly depends on the type of data in the dataset, for instance, in geo-textual databases, where objects contains the location and a set of keywords as attributes, one can use both textual and spatial predicates to process spatial keyword queries [22] [29] [192]. For

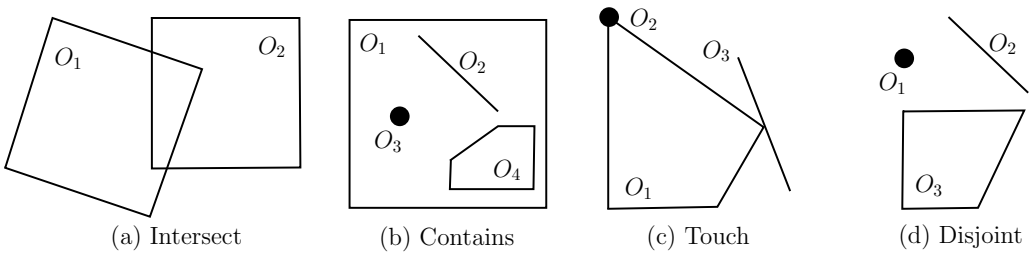


Figure 2.2: Topological queries example: (a) spatial object O_1 intersects with O_2, (b) object O_1 contains $\{O_2, O_3, O_4\}$, (c) objects $\{O_2, O_3\}$ touch O_1, (d) objects $\{O_1, O_2, O_3\}$ are disjoint.

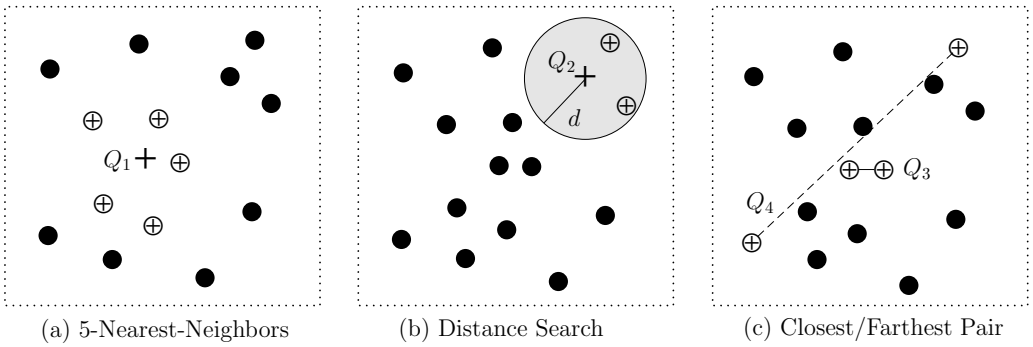


Figure 2.3: Distance-based queries example: (a) the 5-Nearest-Neighbors (5-NN) of the query point Q_1, (b) distance d selection from query point Q_2, (c) closest Q_3 and farthest Q_4 pairs in the input dataset.

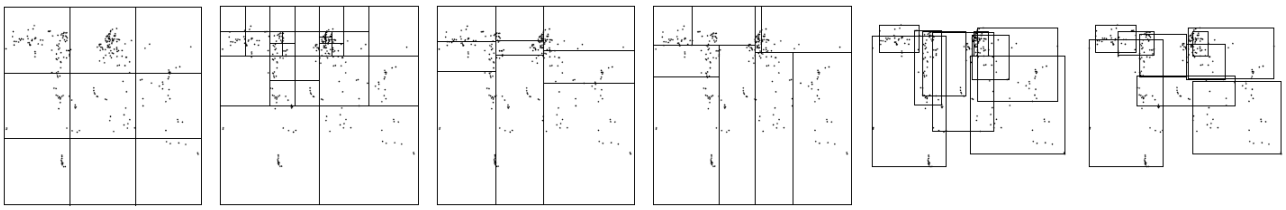
trajectory databases the problem is even more challenging, due to the temporal dimension, sequential nature, and asynchronous sampling rate of trajectory points. We discuss trajectory queries in more details in Section 2.2.2.

2.1.2 Spatial Partitioning and Indexing

Most queries over spatial data, such as spatial selection, spatial join and k-NN, can be performed by simply looking at nearby objects, without the need to scan the entire dataset. Spatial partitioning techniques, such as uniform Grid, K-d Tree, Quad Tree, and Voronoi Diagrams, are used to organize spatial objects in terms of geographic proximity in order to prune the search space, hence reducing the number of disk I/O and memory overhead [35] [166]. Furthermore, for distributed applications, each partition becomes the unit for parallel processing, providing orders of magnitude speedup in system latency and throughput [43]. Therefore, providing an efficient strategy to organize spatial data and process queries over large datasets is a key point to build scalable systems.

In a good partitioning strategy for parallel computation the size of data per partition is fairly uniform in order to achieve good load-balancing and avoid idle processes. The goal is to distribute objects so that each process unit will perform roughly equal work. After partitioning the dataset, the application can process a query in every group in a *filter-and-refinement* manner. In the *filter* step we prune the search space – eliminate objects that cannot be part of the query result – and select intermediate candidates. Finally, in the *refinement* step, candidate objects are checked with a precise algorithm to select the objects satisfying the query [201].

In this work we will focus on spatial partitioning techniques that have been recently applied for distributed computation of spatial queries using MapReduce [12] [43] [168]; and extend these techniques for spatial-temporal trajectories. These techniques have demonstrated to provide good load balance and decentralization, essentials for MapReduce computation. Figure 2.4 shows some examples of spatial partitioning techniques over a points dataset from [43].



(a) Grid (b) Quad Tree (c) STR and STR+ (d) K-d Tree (e) Z-curve (f) Hilbert Curve

Figure 2.4: Spatial partitioning techniques in SpatialHadoop [43].

Spatial-aware partitioning strategies in MR can achieve up to 10x faster performance than multi-core *divide-and-conquer* by maintaining data locality [44] [48] [200], since only a smaller number of partitions containing query candidates are selected for processing, reducing query latency and CPU cost. Spatial-aware partitioning approaches are preferred for MR environments and concurrent threads; first because the faster the query response time, the sooner it gives resources back to the application; and secondly, location-based services and MR systems are often serving more than one application at same time, e.g. Spark and Hadoop might be serving other applications through their wide set of tools, or serving concurrent jobs on the same application. Hence, reducing query latency and resources use allows the system to serve more concurrent requests, and permit our application to work with other MR systems in a non-intrusive way.

2.2 Trajectory Data Management and Applications

In this section we introduce the main ideas, applications, challenges, and related work on spatial-temporal trajectories. Trajectory data management aims the modeling, organization and storage of trajectory data for efficient and scalable data retrieval and query processing.

Figure 2.5 shows a framework that summarizes a procedure of trajectory data management. Briefly, raw trajectory data are available from several sources, such as electronic devices, maps, web, and documents. From bottom to top, the first step on trajectory data management is to model and prepare the raw data, by collecting and integrating the data into a single representation; trajectory data can also be synthetic generated for experimental purposes. Next, the raw data must be preprocessed in order to enhance data quality, and make the trajectory data more meaningful [55] [73] [195]. Next, the integrated and preprocessed data is organized for storage; whether centralized or distributed, on-disk or in-memory, storage systems for trajectory data employ spatial-temporal partitioning and indexing to organize the data for efficient retrieval and querying [35] [93] [158]. Trajectory storage systems either focus on storage and organization of data for a single and expensive query, such as k-nearest-neighbors

(k-NN) join, or for efficient data retrieval, such as range search, historical search, topological-based, and distance-based queries. Finally, the data and queries are ready to be used in trajectory data mining and real-world applications [58] [195] [199].

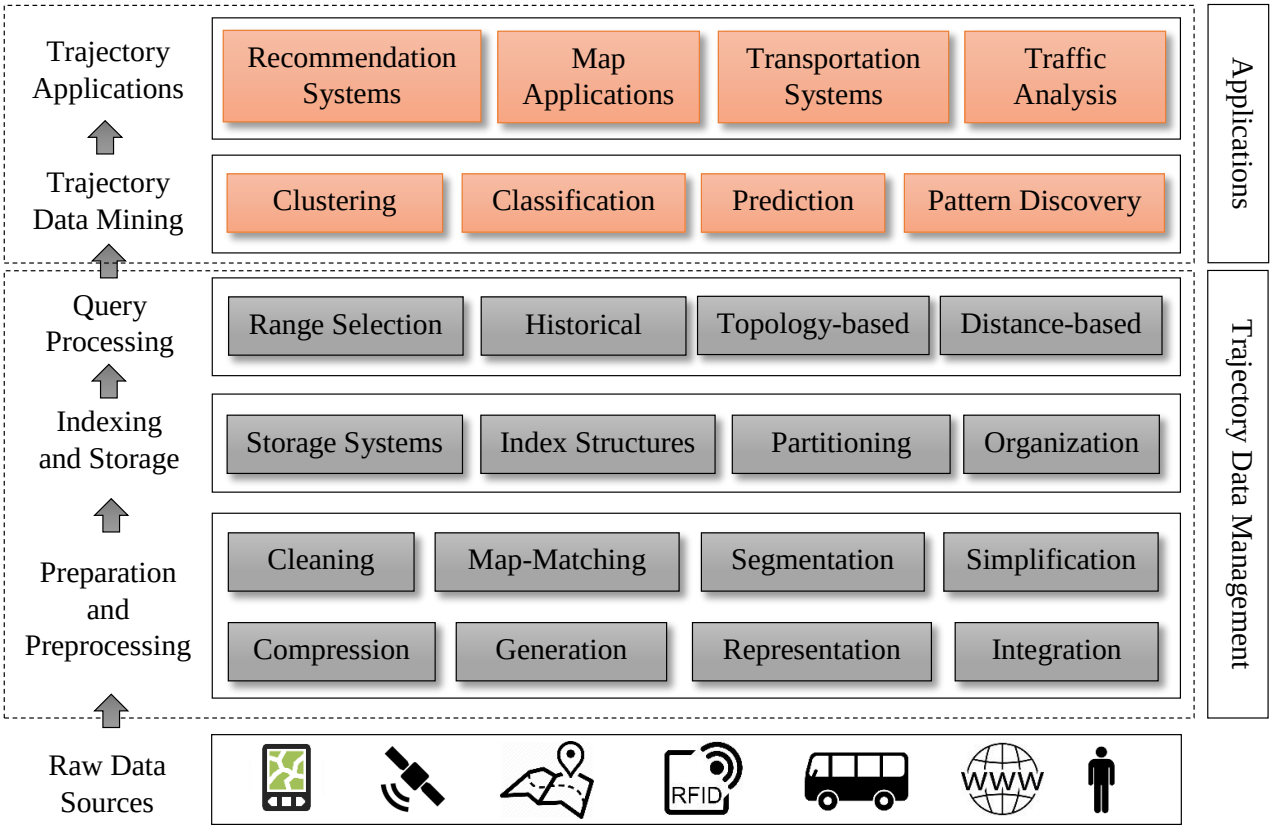


Figure 2.5: Trajectory data management and mining overview.

A trajectory describes the motion history of any kind of moving object, such as people, animals and natural phenomena. Trajectories of moving objects are continuous in nature, but captured and stored as a collection of spatial-temporal points.

Trajectories may also be defined as a position versus time continuous function $(x,y) = f(t)$ in a 2D space, or $(x,y,z) = f(t)$ in a 3D space; however, for the sake of simplicity, and without loss of generality, trajectories discussed in this work are represented as in Definition 1.

Definition 1. (Trajectory) A trajectory T of a moving object is a sequence of spatial-temporal points, where each point is described as a triple (x,y,t), where (x,y) are the spatial location of the moving object, such as its *latitude* and *longitude* coordinates, at a time t, that is, $T = [(x_1,y_1,t_1),(x_2,y_2,t_2),\ldots,(x_n,y_n,t_n)]$ in a two-dimensional space, where n is the number of sample points, and $t_1 < t_2 < \ldots < t_n$.

Definition 2. (Trajectory Segment) A trajectory segment s is defined as any segment connecting two consecutive GPS points p_i to p_{i+1} of T, that is $s_i = \overline{p_i p_{i+1}} \in T$. Consequently, a trajectory with n points has $(n-1)$ segments.

Notice that the focus of the database system proposed in this work is on the layers of trajectory data management, briefly described in the next sessions. For the best of our knowledge, no system for trajectory data management addresses all the described layers.

Applications: Because of its spatial and temporal features, trajectories of moving objects contain rich information about people, locations, wild life and natural phenomena, for instance, and have been widely used for a great number of real-world applications, such as best route suggestion based on GPS trajectories of taxi drivers (i.e. route recommendation) [176] [177], or based on the path use frequency in a specified time period [97]; time prediction of public transportation by means of bus routes analysis [34]; find points of interest (POI) such as touristic attractions, hotels or restaurants, and the popular routes among interesting locations using density of trajectory points [32] [196] [38] [96]; identify gathering patterns in urban areas for recommendation systems [194]; support travelers to plan a trip itinerary to an unfamiliar location based on trajectory data (i.e. trip recommendation) [197] [198]; predict the best transportation mode between locations based on the time taken and frequency of moving objects within a given area [196]; soccer team analysis and hurricane motion patterns by means of clustering and pattern recognition techniques over trajectory data from soccer players and natural phenomena respectively [61]; drivers pastern analysis, city traffic planing, dynamic event identification, human interaction, and so on [33] [36] [58] [62] [198]. To facilitate human understanding of trajectories data, some works have focused on semantic enrichment by associating meaningful annotations to the raw data, in order to highlight significant behavior of the trajectories, such as the motion behaviors and main locations a trajectory intersected [10] [138] [144] [145] [171].

When attached with textual information, trajectory databases also support keyword-based queries, based on both their spatial-temporal attributes and the textual data attached, such as *"suggest the most popular Chinese restaurants"* in a given region [190] [192]. An excellent book on trajectory data processing and applications can be found at [199].

2.2.1 Trajectory Data Preparation and Preprocessing

Raw trajectories should go through a series of preprocessing steps before they become suitable for indexing, querying, and mining. Trajectory data preparation and preprocessing are basic steps performed once the raw data is gathered in order to improve data quality [55] [195].

Trajectory data preparation and preprocessing include several steps and techniques, such as: cleaning, simplification, segmentation, compression, calibration, integration, semantic enrichment, and map-matching. Trajectory data preprocessing mainly focus on improving data quality, or representing trajectories in a more meaningful way for further processing [199]. In the section, we briefly introduce some of the most common operations in trajectory data preprocessing [55] [195].

Cleaning: Since GPS records can be incomplete, inaccurate and noisy due to connection problems and signal loss, urban canyons, sparse collection rates, and law restrictions, etc., GPS trajectories may not accurately reflect the location of moving objects. Therefore, data cleaning is the process of

discard impossible locations or trajectories exploiting some specific constraints, such as maximum speed, coordinates distribution, unreachability constraints, for instance, in order to detect suspicious moving objects, or to capture features of many abnormal trajectories. Map-matching (introduced in Section 2.3) is one of the main techniques employed in trajectory data cleaning and data quality enhancement.

Segmentation: In many applications a trajectory is partitioned into more meaningful and less complex sub-trajectories, such as a path with multiple road segments, or according to behaviors of moving objects [144]. Trajectories may as well be partitioned for storage and indexing purposed, for instance, long trajectories can be divided into smaller sub-trajectories according with their intersections spatial region, their speed, or divided by time intervals [35] [158]. This allows trajectories to be stored and processed in a efficient manner, as well as extract sub-trajectories, regions, and period patterns.

Compression and Simplification: GPS sensors can collect data at high sampling rates, therefore, huge amounts of data with density can be generated by such devices. However, many trajectory data-driven application do not rely on such a precision of location. Furthermore, in data with high density of sample points, several location points in a trajectory are often redundant. Therefore, trajectory compression and simplification algorithms aim to reduce the size, density, or complexity of trajectory data, by removing redundant information, i.e. remove redundant sample points or dimensions, hence reducing storage requirements and communication costs. Trajectory data compression algorithms aims to minimize the size of the data with a minimum of information loss.

Completion: Some devices, on the other hand, collect data in very low sampling rates, only providing partial observations of actual trajectories. Data completion aims to infer missing coordinates in the raw trajectories, in order to reduce uncertain.

Integration: Different devices record and store data using different formats. Even though GPS data often contains the same spatial-temporal and semantic attributes, describing the moving object's trajectory, the integration of these datasets into a single format and storage platform is yet an issue. Therefore, spatial-temporal trajectory data integration is significant to combine data from different sources into a unified format for trajectory data-based applications.

In this work we focus on map-matching as our main preprocessing and quality enhancement technique for trajectory data. Map-matching plays a key role on trajectory preprocessing by improving data quality and reducing uncertainty.

2.2.2 Trajectory Data Queries

In trajectory database, where objects are a non-uniform series of spatial locations, attached with temporal attributes, one must take sequentiality and the temporal dimension into account. For distance-base queries, the problem is even more challenging, once we need a distance function to calculate the

distance (i.e. similarity) between trajectories, which is not a trivial problem, due to the non-uniform sequential nature and temporal dimension of trajectories. Nevertheless, tens of similarity distance measures for trajectory data have been proposed in the literature [157] [162], we discuss trajectory distance measures in more details in Section 2.2.4.

Formally, given an input trajectory dataset $\mathbb{T}$ with n records, for any two trajectories $T_i, T_j \in \mathbb{T}$, $d(T_i, T_j)$ denotes the distance (or similarity) between them. In this work we consider the following operations over trajectory datasets, due to their wide application in practice [31] [33] [35] [56] [130] [153] [158] [172].

A spatial-temporal selection retrieve all trajectories within a given spatial region and time interval, similar to a *SELECT/FROM/WHERE* clause in relational databases, where the predicate is the trajectory overlapping with both the region area and time interval.

Definition 3. (Spatial-Temporal Selection) Given a trajectory dataset $\mathbb{T}$, a spatial region R, and a time interval from t_0 to t_1, a spatial-temporal selection, namely $ST(\mathbb{T}, R, t_0, t_1)$, finds all trajectory segments $s_i \in \mathbb{T}$ active during $[t_0, t_1]$ which intersects with the region of R, that is $ST(\mathbb{T}, R, t_0, t_1) = \{s_i \in \mathbb{T} \mid s_i \subset (\mathbb{T} \cap R \cap [t_0, t_1])\}$.

Selection queries are useful to select a small sample of a big dataset for a given time interval and spatial predicate (e.g. range selection, intersect, overlap), for example: *"select all trajectories from a given neighborhood in New York city, active yesterday during peak time"*. An example of a spatial-temporal selection query area R is given in Figure 2.6.

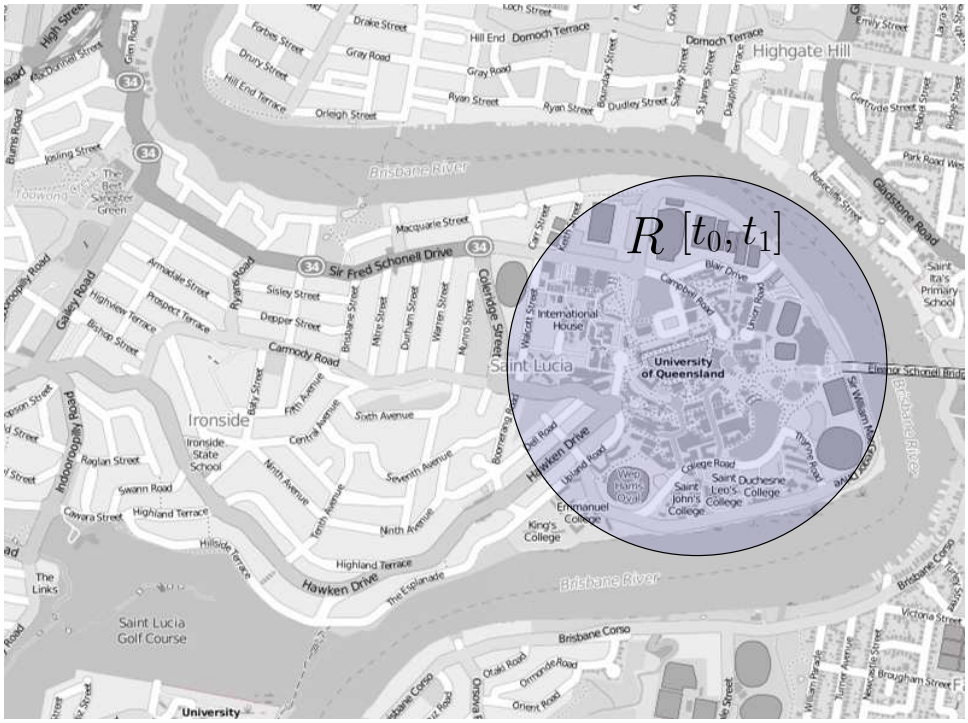


Figure 2.6: Example of spatial-temporal selection, where given a query region R over the city of Brisbane, we want to retrieve only those trajectories inside the area of R and active during a given time interval $[t_0, t_1]$.

Definition 4. (Topological Selection) Given a trajectory dataset $\mathbb{T}$, a query object Q (e.g. a polygon, a trajectory, a circle), and a topological predicate $\otimes$ (e.g. intersect, touch, overlap), a topological selection finds all trajectories $T_i \in \mathbb{T}$, such that $T_i \otimes Q$ is true.

Definition 5. (Distance Selection) Given a trajectory dataset $\mathbb{T}$, a query trajectory Q, a trajectory distance function $d(T_i, T_j)$, and a distance threshold τ, a distance selection finds all trajectories $T_i \in \mathbb{T}$, such that $d(T_i, Q) \leq \tau$.

Definition 6. (Shortest Path) Given a trajectory dataset $\mathbb{T}$ (as a sequence of spatial points, or in map representation), two query locations Q_i and Q_j, the Shortest-Path operation finds the trajectory $T_i \in \mathbb{T}$ connecting Q_i to Q_j with the shortest distance.

The k-Nearest-Neighbors (k-NN)[1] for trajectories is a distance-based query that returns the k closest (i.e. most similar) trajectories from a given trajectory Q, in a given time interval t_0 to t_1.

Definition 7. (k-NN Trajectories) Given a trajectory dataset $\mathbb{T}$, a query trajectory Q (Q might be a series of query locations), a time interval from t_0 to t_1, a trajectory distance function $d(T_i, T_j)$, and an integer $k \geq 1$, the k-Nearest-Neighbor trajectories of Q, denoted as $k\text{-NN}(Q, t_0, t_1)$, is a subset of $\mathbb{T}$, such that for every trajectory $T_i \in k\text{-NN}(Q, t_0, t_1)$, and for every trajectory $T_j \in \mathbb{T} - k\text{-NN}(Q, t_0, t_1)$, $d(Q, T_i) \leq d(Q, T_j)$, where T_i and T_j are active during $[t_0, t_1]$, and $|k\text{-NN}(Q, t_0, t_1)| = k$.

Definition 8. (k-NN Trajectories Join) Given two trajectory datasets $\mathbb{S}$ and $\mathbb{R}$, a time interval from t_0 to t_1, and an integer $k \geq 1$, the k-Nearest-Neighbor trajectories Join, denoted as $\mathbb{S} \bowtie_{kNN} \mathbb{R}$, finds in $\mathbb{R}$ the $k\text{-NN}(s_i, t_0, t_1)$ for all trajectories $s_i \in \mathbb{S}$. This problem is also known in the literature as All-Nearest-Neighbors (ANN), when $\mathbb{S} = \mathbb{R}$.

The Nearest-Neighbor query (NN) is a special case of the k-NN for $k = 1$. The problem of identifying similar (or close) trajectories, in particular, is useful for automatic classification and recommendation systems, origin-destiny analysis, and identify objects that move in a same pattern, for instance. As an illustrative example, suppose that in a big city a subway service has been under construction; it would be of great assistance to the experts in the field to know the similarity between the current public transportation services (e.g. bus lines) and the subway lines under construction; in order to re-organize the public transportation routes, and propose timetables and metro stations, for instance [56]. Another particular case of distance-based queries for trajectory dataset is Reverse-Nearest-Neighbors (RNN).

Definition 9. (RNN Trajectories) Given a trajectory dataset $\mathbb{T}$, a query trajectory Q, and a time interval $[t_0, t_1]$, the Reverse-Nearest-Neighbors of Q, denoted as $\text{RNN}(Q, t_0, t_1)$ finds all trajectories $T_i \in \mathbb{T}$ active during $[t_0, t_1]$ which have Q as their Nearest-Neighbor (NN), that is, a trajectory $T_i \in \mathbb{T}$ belongs to $\text{RNN}(Q, t_0, t_1)$ iff $1\text{-NN}(T_i, t_0, t_1) = \{Q\}$.

[1] The k-NN query for trajectories is also known in the literature as k-Most-Similar-Trajectories (k-MST).

A similar problem to the **k**-NN trajectories introduced by Chen et al. [33] aims to search for the k-Best-Connected trajectories (k-BCT) to a given set of query points (i.e. trajectories that are close to all given locations); the k-BCT algorithm can be applied for trip planing, for instance.

Most approaches for trajectory data processing execute some sort of primitive query beforehand over the entire dataset, or a combination with keyword-based queries [190] [192], for instance: *"Retrieve all trajectories in the city center of Brisbane, between March and April this year"*, so that one can identify points of interest in the city center for a given season [197] [198], and suggest transportation modes [196].

Or nearest neighbor queries, for instance: *"Given the trajectory T of a route between two locations, retrieve the closest (most similar) trajectories from T"*, which can be used, for instance, in alternative routes suggestion [176] [177], public transportation analysis [34], or outliers detection [20] [82].

2.2.3 Trajectory Data Storage and Indexing

The most used structure to index spatial data for single-threaded computation is R-Tree [66]; however R-tree does not directly support moving object's trajectories, once it is for spatial dimension only, index structures for trajectories must be able to cope with both spatial and temporal dimensions. Therefore, several solutions have been proposed to manage trajectory data, they either propose a tree-based index structure to prune the search space and speed up the processing of a specific query, or propose a storage system to organize trajectory data and optimize I/O.

General Index: TB-tree [124] index both space and time in a tree structure using bounding box representation for trajectories; however, it does not handle long trajectories properly, which leads to very large bounding boxes. TPR-tree [133] and TPR*-tree [148] extend R-trees to predictive queries, indexing trajectories using a prediction model, based on time-parametrization and velocity vectors, to predict the future location of a moving object; STRIPES [113] extend TPR-trees to reduce tree updates and I/O cost on predictive queries. SETI [25] uses two index structure, where the space is partitioned into cells and inside each cell a R-tree is used to index the temporal dimension, however SETI does not describe the size or geometry of the cells, leading to a imbalanced partitioning strategy. SEB-tree [137] splits space and time into zones and index spatial-temporal objects by the zone ID, but it is not specifically designed for sequential spatial objects (i.e. moving object's trajectory). Similarly to SEB-tree, PIST [18] uses a tree structure to index spatial-temporal points only, rather than sequential trajectories. Rasetic et al. [131] propose a splitting strategy for trajectories into sub-trajectories, and index these sub-trajectories using R-tree, the model is proposed in order to minimize the I/O needed for selection queries. Similarly, PPR-tree [67] present a trajectory splitting model for minimizing the bounding rectangle representation of trajectories for small range queries. CSE-tree [161] partition the space into a grid and index data using a probabilistic model on the data update frequency. Some works proposed the use of polynomial approximations for trajectories [21] [102], even though they provide a more tight representation of a trajectory than the MBR representation used in tree structures, they have

the requirement that the trajectories should be of the same length (i.e. number of spatial points), and provides a costly polynomial calculation and degrees approximation for large datasets.

The main drawbacks of these centrally-based structures is that they do not provide full decentralization for parallel computation, and do not scale for large datasets, hence they cannot leverage the benefits of the MR model. Furthermore, all aforementioned works are for disk-based computation only, whereas our solution takes advantage of in-memory structure to speed up similarity search.

Storage Systems: Another set of works provide a full storage structure, on both main-memory and disk, to organize and aid the access to trajectory data. SharkDB [158] is a storage architecture for trajectories in column-oriented in-memory databases; SharkDB partitions the trajectory space into time frames, in order to support general purpose trajectory operations, e.g. time window selection and nearest neighbor. SharkDB also provides a system for data visualization [159]. However, SharkDB uses only temporal partitioning of the dataset, and focus on column-oriented environments. SECONDO [65] [64] is a extensible open-source DBMS into which a wide set of data structures and algorithms for moving object's trajectories have been implemented. TrajStore [35] provides an adaptive disk-based storage system for trajectory data. TrajStore indexes trajectories on a quad-tree structure and clustering methods to group spatially close objects, and uses a set of compression methods to reduce storage overhead.

A MR-based environment, on the other hand, can provide a distributed and fault-tolerant solution for massive datasets; by using Spark's RDD structure, we can provide a reliable in-memory solution for parallel and concurrent tasks on both in-memory and on-disk. We will introduce MapReduce and Spark based works further in Section 2.6.

2.2.4 Trajectory Distance Measures

Unlike other simpler spatial objects, such as points and polygons, which the distance can be straightforward measured using Euclidean based metrics, the distance between trajectories needs to be carefully defined in order to reflect the true underlying similarity. This is due to the fact that trajectories are essentially non-uniform sequential data with variable length, attached with both spatial and temporal attributes, which may or may not be considered for similarity measures; one may also need to consider data uncertainty [191]. Besides, one must take into account other variants such as shape, time shifting, non-uniform sampling rates, and rotation, for instance. Overall, trajectories are considered similar if they follow a certain motion pattern, or move in a similar way (i.e. keep spatially close to each other) for the majority of their time extent.

The problem of detecting similar trajectories is useful for decision making applications based on moving objects analysis; for instance, one may be interested in planning a road network capacity, planning municipal transportation or detect usual road paths in a city to avoid traffic jam. In this sort of problem, trajectory similarity analysis and query processing, such as the top k-NN trajectories, play an important role.

Tens of similarity distance measures for trajectory data have been proposed in the literature [157]; every similarity measure claim an advantage over the others in a different aspect, and each one provides its own index structure to prune the search space and speed up similarity search. The most classics distance measures that have been widely cited in the literatures are briefly described below. A benchmark comparing most of these distance measures can be found at [157] [162].

- **Euclidean Distance:** Euclidean distances for trajectories are easy to implement, they take the average Euclidean distances between the two trajectories sample points, and can be indexed with any access method; however, it demands the two trajectories to be on the same size (i.e. same number of sample points).
  - ED: Euclidean Distance for time-series [52] [128];
  - EDSW: Euclidean Distance With Sliding Window [85];
- **Dynamic Time Warping**: DTW and its variants use a recursive search between the trajectories sample points for those with minimum distance; thus if two trajectories vary in time or speed, DTW can still detect their similarity; however, DTW is time-consuming and it is not a metric.
  - DTW: Dynamic Time Warping [132] [173];
  - PDTW: Piecewise Dynamic Time Warping [79];
- **Edit Distance:** Similar to the edit distance for string, Edit Distances for trajectories and its variants calculate the minimum number of edits needed to transform one trajectory into the other, by insertion, deletion, and substitution of points. Edit distances are invariant to scale and points shifts. EDR and EDwP use L_2Norm and are non-metric, EDR uses L_1Norm and it is a metric, but only applies for trajectories with uniform sampling rates and same size.
  - EDR: Edit Distance on Real Sequence [31];
  - ERP: Edit Distance With Real Penalty [30];
  - EDwP: Edit Distance with Projections [130];
- **Longest Common Subsequence:** LCSS for trajectories search for points matching like in a string's characters, for this purpose a distance threshold ε is used; if the distance between two points is smaller than ε, they are considered a match. LCSS is robust to noise; however, it is not a metric and can be inaccurate if the value of ε is not carefully chosen.
  - LCSS: Longest Common Subsequence [153];

Another group of distance measures for trajectories are line-based measures, which compare trajectories by their shapes from lines generated from their sample points, instead comparing the points directly. Some examples of this type of distance measures include DISSIM [56], OWD [91], and LIP [120].

2.3 Map-Matching

Map-matching plays a key role on trajectory preprocessing by improving data quality and reducing uncertainty. Since GPS records can be incomplete, inaccurate and noisy due to connection problems and signal loss, urban canyons, sparse collection rates, and law restrictions, etc., GPS trajectories may not accurately reflect the location of moving objects. Therefore, map-matching is a process to ease the uncertainty and improve the accuracy of trajectory data by matching the GPS records to the logical model of the real world [193], such as the road network graph.

In general, the GPS errors are caused mainly by two reasons [123]: (1) Measurement Error, caused by the inaccuracy of measurements due to device failure or imprecision; and (2) Sampling Error, caused by the uncertainty of representing a vehicle trace. Therefore, map-matching algorithms are introduced to reduce the GPS errors. By integrating the raw GPS data with the road network data, we are able to align the deflected GPS points to the correct road segments.

There is no existing definition for map quality issue. However, a large amount of works addressed some map-related problems, which can be classified into the following map quality issues:

- **Accuracy.** The map is required to be accurate, which means the map should represent the real-world topological structure and geographical location precisely. In another words, for a road map graph G, all the vertices in G should have the correct location information, and all edges in G should have the corresponding real-work links between vertices, and the shape of the links remain the same.

- **Completeness.** The map is required to have accurate and complete attributes. The elements in a map require not only the geographical information, but also other attributes, like speed limit and width of a road segment, turning restriction of a intersection, etc. All these attributes are useful in many map-based applications so that their accuracy and completeness is very important.

- **Recency.** The map is required to be up-to-date. In fact, the change of the roads happen frequently due to rapid constructions of roads, road maintenances and so on.

Map-matching algorithms assume that the road network is stable and precise, and all the running vehicles are essentially constrained to the road network. This constraint holds for most circumstances except when the vehicle drives into off-road areas, like casual parking, private land, etc. Intuitively, this type of unusual traces should account for only a small portion of the entire data.

2.3.1 The Map-Matching Process

Following we formally describe the problem of Map Matching, firstly introducing some background knowledge and definitions.

Definition 10. (Road Network) A road network is a directed graph $G(V,E)$ representing the digital map of streets and roads of a geographic region, where each edge $e \in E$ represents a road segment in

the graph, and each vertex $v \in V$ of the graph represents the intersections and end-points of the road segments.

Definition 11. (Road Edge) A road segment $e \in E$ is a directed edge from a starting vertex $v_i \in V$ to an ending vertex $v_j \in V$ in a road network graph $G(V,E)$, and associated with a list of intermediate points that describes the road polyline.

In digital map representation, both edges and vertexes in the road network graph are associates with an ID, and a set of semantic attributes, such as *speed* and *length*.

Definition 12. (Road Path) A road path P is a set of connected road edges, $P = \{e_i, ..., e_j\} \in G(V,E)$, connecting two locations (vertices) v_p to v_q of $G(V,E)$.

Definition 13. (Map-Matching) Given a trajectory T, and a road network graph $G(V,E)$, map-matching is the problem of how to match T to a path P of $G(V,E)$.

Map matching has applications in satellite navigation, GPS tracking of freight, and transportation engineering, for instance. The overall approach for map-matching is to take recorded serial location points (e.g. GPS coordinates), and relate them to edges in an existing road network graph. However, this approach can quickly became cumbersome for large trajectory and map datasets, since every GPS point record has to be compared with every road edge.

There are two main algorithmic approaches for map matching in the literature, Local [8] [28] [81] [165], and Global [19] [90] [92] [101] [125]. In short, the three main steps followed by map-matching algorithms are: (1) identifying a set of candidate edges in the road graph within a given radius from the location point, then (2) calculating the weight for each candidate edge (e.g. shortest distance between the point and the edge), and finally (3) retrieving the edges that maximize the weight.

Local (or **incremental**) algorithms only consider the trajectory and the road network geometries to relate a trajectory point to its nearest edge (point-to-edge) in the road map. This method is simpler and faster, and more commonly used in on-line map matching, since they rely on the previous trajectory points observations only, which makes it more difficult to use statistical models on the trajectory topology. However, due to measurement errors and GPS inaccuracy, this approach is prone to error (i.e. point mismatch). Wei et al. [164] provided a comparison between local and global map-matching algorithms, and discovered that local algorithms performed poorly, specially due to Y-splits on road networks. For instance, in Figure 2.7 while there are two possible matching candidates, e_2 and e_3, for point p_3, e_3 is the most obvious edge to match, since its next connecting point p_4 is better matched with e_3, and real moving objects are more likely to follow a direct path [92]. Therefore, the best match for trajectory T is the path $P = \{e_1, e_3\}$ connecting v_1 to v_3.

Global algorithms, on the other hand, take into account the geometry and other features of the trajectories and the road network, such as speed, topology, the connectivity between points and edges, and the road network speed limits, in order to find the best match of a trajectory on the road network, thus easing the uncertainty. Global algorithms are mostly used in offline map matching, and use future observations to better match the trajectories correctly. These methods make use of statistical models

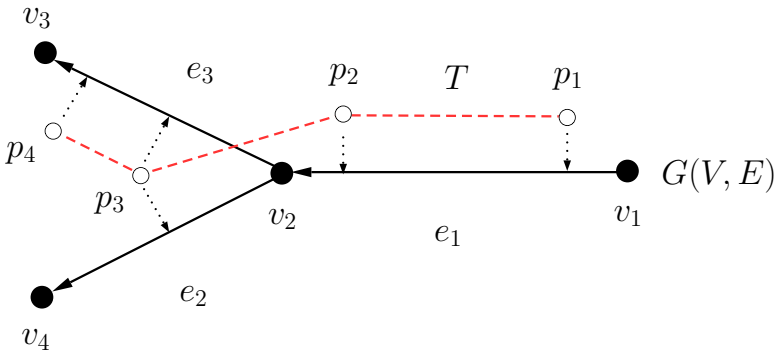


Figure 2.7: Example of road network graph $G(V,E)$, with edges $e_{[1..3]}$ and vertexes $v_{[1..4]}$; and a GPS trajectory T (red dotted) with four coordinate points $p_{[1..4]}$ to be matched with the road network.

(e.g. Hidden Markov Model [101], and spatial-temporal analysis [92]), and sacrifice performance to achieve better accuracy. Offline map-matching plays a key role on trajectory pre-processing by improving data quality and reducing uncertainty when whole new trajectory data, or new and more accurate map data, became available.

2.3.2 Map-Matching Techniques

Related work on map-matching can be divided into two main categories: (1) Serial map-matching algorithms focusing on accuracy and match trajectories in a serial fashion; and (2) Parallel algorithms use spatial partitioning and parallel computation to speed up map-matching. We also briefly discuss related work on spatial data processing using Spark.

Serial Algorithms: Lou et al. [92] presented a spatial-temporal algorithm (ST-Matching) for matching trajectories with low sampling rates (e.g. 2min gap between each point). Firstly, for every trajectory sample point $p_i \in T$ ST-Matching retrieves all candidate points c_j from the road network within a radius r from p_i – any candidate point farther from p_i than r is taken as a impossible match. From the candidates set they compute the projection of p_i on each edge containing c_j; the algorithm chooses the best match first choosing the edges which contain the nearest points from p_i (Spatial Analysis Function), then choosing the edge on which the speed limit is closer to the trajectory speed (Temporal Function). Newson et al. [101] proposed a Hidden Markov Model (HMM) based algorithm to find the most likely road to match a trajectory. The algorithm models the connectivity of the road segments into a HMM where each state is a road segment. The Emission Probability of every state transition is calculated using the Gaussian distribution, where the input is the distance from a trajectory location p_i to a road segment e_j, hence segments farther from p_i are assigned with a lower probability. Finally, the most likely matching road is found using Viterbi algorithm to compute the best path through the HMM lattice. Similar to ST-Matching, the HMM algorithm is robust to noise and temporal sparseness, and only considers matching edges within a $200m$ radius from the trajectory point. OHMM [59] extends the HMM algorithm in [101] for incremental on-line map matching. OHMM uses Support Vector Machines (SVM) classifier instead Gaussian model to calculate the transition probabilities of the

HMM lattice, thus learning the best match based on the current trajectory state. Similarly, [125] uses a Bayesian classifier, including the topological constraints of the road network, to calculate the transition probabilities in the HMM model for local map-matching. IF-Matching [71] uses information fusion to achieve more accurate map-matching. Along with the geometry and topology of the trajectories IF-Matching also uses speed and direction to better describe the moving object. It also uses the speed constraints of the road to find the best match, however, as the speed of the moving object on the road can be limited at different times of the day due to heavy traffic, the IF-Matching algorithm also applies a function to model the speed on the road network during different times of the day. In [19] the authors present an approach using the Frechet Distance to calculate the most similar road to match the trajectory; however, this method only takes into account the geometry of the trajectories, and it is limited to records with very dense points distribution and low sampling error. [75] and [90] introduce a new multi-track map-matching approach, by matching multiple trajectories to the road graph at same time. The idea is to identify the regular patterns of a group of trajectories in order to find their best match, assuming that all trajectories with same pattern belong to a same path in the road network. Although achieving high accuracy, the main drawback of these serial algorithms is that they focus only on the accuracy of the matching; processing one, or only a few, records at a time, in a single process unit, and do not account for scalability.

Parallel Algorithms: Huang et al. [72] presented the MR-based algorithm HOM for map-matching focus on performance rather than accuracy. HOM divides the data space into a grid, and assigns each GPS point and road link to its corresponding grid; each grid partition is sent to a computing node. Finally, map-matching is done in each partition in parallel using MR. HOM, however, used a incremental algorithm to match points to its nearest road link, and do not consider boundary objects, thus it is not robust to noise. Tiwari et al. [150] proposed a framework focus on scalability of map-matching. The framework uses MR computation and Hadoop HBase as distributed storage to achieve horizontal scalability. Similar to HOM, it uses grid partitioning and does not account for load balancing, however, it uses ST-Matching [92] on each partition to compute map-matching rather than provide a new algorithm. The main drawbacks of these MR-based approaches is that they do not consider homogeneous distribution during the partitioning, which is essential in MR computation, and do not account for boundary objects. [156] provides a parallel approach for streaming map-matching in-memory for real-time processing using grid partitioning, however, the authors do not optimize memory consumption and do not handle boundary objects. [167] addresses the problem of load balancing using Quadtree space decomposition, similar to our approach the authors defined a 5km boundary threshold for replication, so that points near partition boundaries are replicated to all nearby partitions; however, it builds the quad-index using the entire dataset dynamically, which is not directly supported by Spark; our work, on the other hand, applies a sampling-based approach to reduce the partitioning cost using Spark. Furthermore, all the works presented to date use disk-based computation, which negatively affects performance for large-scale data due to I/O overhead. Our work is similar to that in [167], except we use a much smaller space boundary limit, and add one more condition to we make sure

our spatial-partitions are no larger than the MR block size in the cluster configuration (i.e. 64MB by default), so that every partition can be processed by a MR task, the MR block size can be adjusted according to the cluster configurations. We also build our quad-index using a sampling-based approach which can be combined with Spark.

Spatial Data on Spark: A number of existing works provide unified systems for spatial queries using Spark, aiming to achieve better performance and scalability for spatial data. Simba [168] uses Spark's RDD to support spatial indexing and spatial operations natively. Simba adds spatial keywords to SparkSQL grammar, so that users can express spatial operations in a SQL-like fashion. Similarly, GeoSpark [175], SpatialSpark [174], and SparkGIS [16] have been proposed to process spatial data on top of Spark. However, although achieving high performance and scalability for spatial queries, none of these systems fully support trajectory data processing nor map-matching. OceanST [178] does provide support for trajectory data on Spark, however, only for selection queries using uniform static data partitioning. Our proposed framework, on the other hand, provides support for map-matching using balanced space partitioning.

2.4 Distributed Parallel Computation

There is vast body of literature on large-scale data processing, wherein distributed-based solutions outperforms centrally-based in aspects such as scalability, fault tolerance and I/O operations [1] [24] [74] [114] [141] [170]. Distributed computation architectures are designed to share the data across many CPU nodes in a cluster to be processed separately by parallel processes running on each node, making the data processing faster, more efficient and scalable.

The two most popular paradigms for distributed computing on shared-nothing[2] architectures are Parallel Database Management Systems (Parallel DBMS) and MapReduce (MR).

2.4.1 Parallel DBMS

Parallel DBMSs extend common database systems by allowing the parallelization of many data-driven tasks, while still supporting standard relational table schemas, SQL operations, and user-defined functions (UDFs). In a parallel DBMS the data schema and query operations are shared across all the data nodes. The systems architecture are generally designed to hide from users lower level system details, such as data indexing options, storage schema, and join operations strategies [1] [114].

2.4.2 MapReduce Framework

MapReduce (MR) [39] is a open-source programming model proposed by Google for massive data processing in a cluster-based environment, usually composed of inexpensive machines, assisting the solution of many real-world problems in a distributed manner. A MR program is composed of a `Map()`

[2]Commodity machines that do not share CPU and other hardware resources, except network.

a and `Reduce()` methods. The `Map()` method performs filtering, sorting, or other preprocessing operations, and map each data input to a $\langle key, value \rangle$ pair. While the `Reduce()` method merge data with same key, applying some operation or summary on the values. The MR library was built so that programmers do not need to worry about hardware and network failures, resources allocation, process communication and synchronization, data transfer between nodes, and other problems inherent of distributed programming. MR is also more extensible and presents better capacity to deal with unstructured and semi-structured data than parallel DBMS [1] [114] [170].

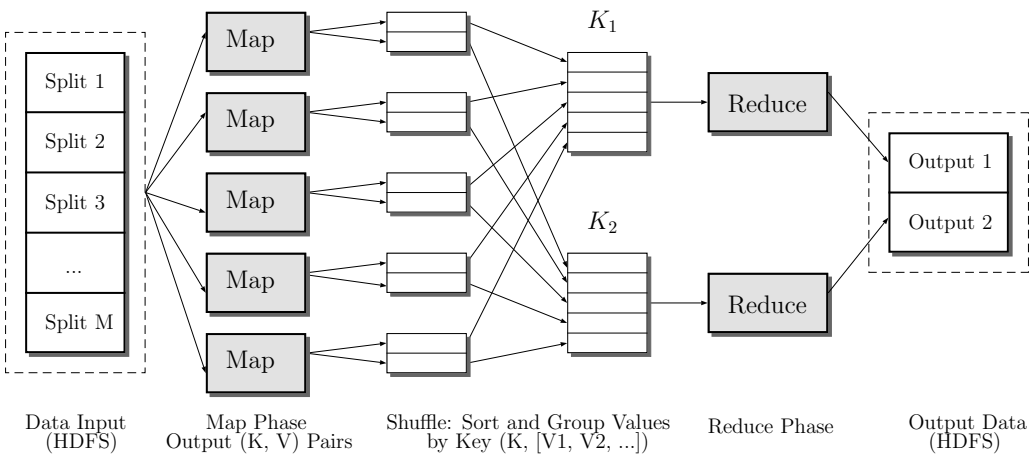


Figure 2.8: MapReduce framework execution overview.

In the MR framework, Figure 2.8, a dataset is partition into M splits across nodes in a cluster. The MR file system (GFS [57]) partition each input split into blocks (64 MB by default), and send each data block to be processed in parallel by *map* tasks. The mapper read each input in the data block and outputs a $\langle key, value \rangle$ pair for the input. When the *map* task is complete, the results are sorted and grouped by *key*, this is the *shuffle* process, so that all the occurrences of the same key are processed by the same reducer. Each *reduce* task receives a $\langle key, list(value) \rangle$, with a list of *values* for a certain *key*, and outputs a $\langle key, result \rangle$ pair after processing the list of *values*. The number of *map* and *reduce* tasks, as well as the data block size, can be controlled by the user; however, some cost-based parameters optimizer exist [70].

Here is a simple example of the word-count problem that can be easily expressed as MR computations from [39]. *"Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write the MR functions similar to the Algorithm 1 pseudo-code."*

The *map* function emits each word plus an associated count of occurrences (just '1' in this simple example). The *reduce* function sums together all counts emitted for a particular word.

There may be hundreds or thousands of machine nodes in a MR cluster. There are Worker nodes, which are responsible to process *map* and *reduce* tasks on each data block; and the Master node, which is responsible to assign *map* and *reduce* tasks to the Workers, as well as allocate resources and manage fault-tolerance. The file system manages fault-tolerance by replicating the data to different nodes, two copies by default, so that if a Worker fails the master resets the failed tasks on another machine that

Algorithm 1 MapReduce Word Count

Input: *key*: document name, *value*: document contents

```
1: function MAP(String key, String value)
2:     for each word w in value do
3:         EmitIntermediate(w, 1)
4:     end for
5: end function
```

Input: *key*: a word, *values*: a list of counts

```
1: function REDUCE(String key, Iterator values)
2:     result ← 0
3:     for each v in values do
4:         result ← result + v
5:     end for
6:     Emit(result)
7: end function
```

contains the data replication.

MR-based works developed strategies to optimize I/O and reduce the network and I/O cost of sorting and grouping the data output from *mappers* to *reducers* (i.e. *shuffle*); and improve data locality and grouping strategies, to keep data with same key fiscally close to one another. Large spatial database applications this can be achieved by means of locality-aware partitioning, which can reduce the number of data objects access, thus reducing network and I/O costs [184]. Surveys about applications and data management using MapReduce can be find at [42] [88].

Hadoop: Apache Hadoop [68] is an open-source framework of tools built on top of MR that enables the processing of massive amounts of data in a distributed parallel fashion; it makes the process of developing solutions for large and distributed data easier for programmers, providing a set of tools to abstract the complexity of writing programs over MR. Hadoop provides its own Distributed File System (HDFS); similar to the GFS [57], the HDFS splits the whole dataset into smaller blocks distributed throughout the cluster (blocks of 64 MB by default), so that each *map* and *reduce* functions can be executed over smaller subsets of the whole dataset, providing the necessary scalability for large-scale data processing.

Apart from that, Hadoop allows the processing of data streams using the framework Storm [143] and can be integrated with Parallel DBMS [1] [170], providing both SQL (e.g. Hive [149]) and NoSQL (e.g. Pig [107]) programming tools.

2.4.3 MapReduce vs. Parallel DBMS

Although both paradigms share many common features, such as high scalability, compressing optimization, and indexing optimization, they are complementary technologies [141] [170]. Parallel DBMS, for instance, shows better performance, more indexing efficiency, and less code to implement queries if compared to MR for large-scale data; whereas MR is more extensible and schema free,

hence presents better capacity to deal with unstructured and semi-structured data; MR is also more fault-tolerant and faster to load data than Parallel DBMS [114]; furthermore, MR scales better for large numbers os nodes [1]. Frameworks like Hadoop [68] and Spark [139] can assist data scientists to achieve a trade off between both technologies [1] [170].

2.5 In-Memory Big-Data Management

In-memory Database: An in-memory database system primarily relies on main memory for data storage, contrasting with database management systems that employ disk storage mechanism. In-memory databases are faster than their disk counterparts because disk access is much slower than memory access, the internal optimization algorithms are simpler and execute fewer CPU instructions. Accessing data in memory eliminates time for search when querying the data, which provides faster and more predictable performance than disk.

A potential drawback with in-memory data storage is the volatility of RAM. Specifically, in the event of a power loss, data stored in volatile RAM is lost. To address the fault-tolerance problem, in-memory database systems uses data replication across multiple machines or maintain a copy of the dataset on disk in case of system or hardware failure. Differently from memory and CPU caching, in-memory datasets are designed to make the data stored in-memory persistent, and available at any time.

Memory Caching: Caching is the process of storing data in a temporary storage area in the main-memory for faster access. Cached data refers to data recently being processed by the CPU or accessed by user applications, thus are readily available since the application can get the data from the cache rather than the original server, saving disk I/O and network traffic. However, main-memory is volatile and cache space is quite limited, and data is released shortly after the task using the data is completed.

In-memory Data Management: In-memory data management aims to store and process data in main-memory to achieve low query latency, and it is more suitable for low-latency services and real-time data analytics, including spatial queries [104] [168], trajectory data processing [27] [115], distributed in-memory data storage [89], and on-line data processing [2]. In-memory storage is also better for iterative MR processes, where it is necessary to apply a function repeatedly on the dataset, since the MR framework reload the data from disk in every iteration, which incurs in a significant performance loss [181], due to the costly load and write operations in a physical partition [187]. In this section we briefly introduce some in-memory storage and analytics systems for Big Data, and the Spark MR framework with more details; a complete survey can be found at [187].

H-Store [78] or its commercial version VoltDB [142], is a distributed main memory OLTP database; H-Store is a row-based relational database, and achieves high performance and throughput by optimizing database operations such as logging and buffer management, which consumes substantial amounts of time, but are unnecessary when storing data in main memory. Each partition in H-Base

is controlled by a *site*, which is a single-threaded entity responsible to execute a transaction without the need of concurrency control in most cases. Hekaton [40] is Microsoft's main memory engine fully integrated with SQL Server; Hekaton was designed for high concurrency, where each thread can access any row in a in-memory table using lock-free data structures [84]. Tables and operations are declared as in regular SQL style, providing effortless SQL Server usage. SAP HANA [54] [136] is a in-memory database developed to integrate both OLTP and OLAP workloads in the same system; SAP HANA support both row-oriented data storage, more beneficial for inserts and updates common in OLTP, and column-oriented storage, more ideal for OLAP transactions which access all values in a column. SAP HANA also provides a temporal indexing to support historical queries, which have been exploited in trajectory databases [158]. MongoDB [100] is an open-source NoSQL database application for documents, MongoDB supports distributed computation by providing atomicity at the document level, allowing operations only within a single data collection, thus join operations are not supported. MongoDB can work as a fully in-memory storage if the data fits in main-memory, or partially in memory otherwise. Piccolo [126] is a in-memory data-centric programming model for parallel data analytics in multiple nodes. Similar to Spark, Piccolo has a control daemon running in a master node, and multiple kernel instances running on slave nodes; all nodes share state via in-memory key-value tables; each kernel sends messages to read and modify table entries using put and get functions.

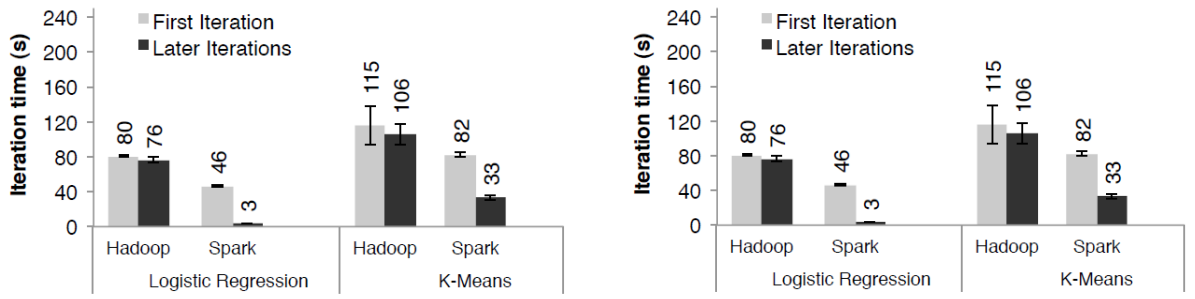
Frameworks like Spark, on the other hand, provides a MapReduce solution for cluster-based computation in main-memory and has been widely used to improve the performance of computing-intensive and Big Data applications.

2.5.1 Spark Framework

Apache Spark [13] [181] is a MR open-source framework for cluster-based data analysis in main-memory environment. Spark is highly scalable and shows faster processing of large-scale data in memory if compared to other MR frameworks like Hadoop, for instance, in a benchmark experiment Spark showed up to 5x faster performance for data-intensive analytics (e.g. page rank, k-Means) than Hadoop [135]. The performance improvements comes from avoiding disk I/O and the smaller cost on objects deserialization by storing the data in main memory, as well as RDD's hash-based aggregation [135] [180]. Spark is particularly suitable for iterative processes, where it is necessary to apply a function repeatedly on the dataset, since the MR framework reload the data from the file system in every iteration, which incurs in a significant performance loss [135] [181].

Figure 2.9shows a comparison of the performance (a) and iteration running time (b) between Spark and Hadoop for the Logistic Regression and k-Means problems [181].

Spark is flexible and can read data from any sort of source and formats, and has being efficiently applied for a range of machine learning problems and faster data analytics, for instance, graph processing [169], data streams computation [182], relational data processing and SQL queries [14] [51], Online Analytical Processing (OLAP) [185], statistical computing based on R language [151], and



(a) Logistic regression performance in Hadoop and Spark.

(b) Duration of iteration processes in Spark and Hadoop for Logistic Regression and k-Means.

Figure 2.9: Hadoop vs Spark comparison. Source [181].

astronomy and bioinformatics applications [105].

Resilient Distributed Datasets (RDD): Spark provides its own fault-tolerant data structure, named Resilient Distributed Datasets (RDD) [180], to organize data collections to be processed in parallel. RDD is a read-only collection of objects partitioned across the cluster nodes, where the working set of data can be reused across multiple parallel tasks [181]. Once RDDs are kept in main-memory, query tasks can iterate over a RDD many times very efficiently. RDDs support transformation and action operations over the dataset. Transformation operations create a new RDD from an existing one, or from the input dataset, such as *map*, *filter* and *groupBy* functions, which return pointers to new RDDs. Operations such as *reduce*, on the other hand, performs an Action on the existing RDD and return values. RDDs allow the programmer to perform as many transformation operations as needed before an action is executed. RDDs can be persisted at both main memory, for a faster access and data processing, or on disk, which still performs up to 10x better than other MR frameworks.

The following script demonstrates how to compute the word-count problem on Spark, in Scala language, as an example of use of Spark's RDD data structure for distributed computation:

```
val textFileRDD = sc.textFile("data.txt").cache()
val countsRDD = textFileRDD
                .flatMap(line => line.split(" "))
                .map(word => (word, 1))
                .reduceByKey((a,b) => a+b)
```

The **sc** variable represents the Spark Context, which tells Spark how to access the cluster. For more information on Spark cluster configuration visit the Spark configuration guide[3]. The *textFile* function read the data from the Spark Context environment into a RDD object and cache *cache* it into main memory; a *flatMap* transformation splits each line of the file into words, which are further mapped to $\langle key, value \rangle$ pairs as $\langle$"*word*", 1$\rangle$. A *reduceByKey* action is finally performed on the new RDD from

[3]Spark Configuration Guide:
https://spark.apache.org/docs/latest/configuration.html

the map phase, to sum the occurrences (1 values) of each word (key). The *reduceByKey* returns at the end of the process a RDD of pairs $\langle$"*word*", *numberOfOccurrences*$\rangle$.

Some more few examples of functions that can be performed on the new RDD are *count*, *collect* and *filter*:

```
// Number of items in this RDD
val numItems = countsRDD.count()
// Return all elements in this RDD
val elements = countsRDD.collect()
// Filter words counted more than 10 times
val filterRDD = countsRDD.filter(word => word.value > 10)
```

Another useful functionality of Spark is Broadcasting. Spark broadcasting allows the application to keep a read-only variable cached on each machine's memory. They can be used, for example, to give every node in the cluster a copy of a variable or dataset that will be used by all nodes in the Spark context. An example in Scala language of Broadcast variable for an array of numbers is shown as follows:

```
val broadcastVar = sc.broadcast(Array(1,2,3,4))
```

For a decent programming guide on how to use other Spark functions with Scala, Python and Java, please visit the Spark programming guide[4].

Spark SQL: To ease the process of writing relational data-based queries on Spark, Armbrust et al. [14] developed SparkSQL, a new SQL-based module on Spark to perform relational operations and optimizes query processing. SparkSQL integrates relational and procedural data processing on Spark framework providing a new data structure (i.e. *DataFrame*), similar to a relational table in a DBMS, to support relational operations (e.g. `select, filter, join, groupby`), and a relational query optimizer (i.e. *Catalyst*), to speed up query processing using *DataFrames*, *Catalyst* supports both rule-based and cost-based optimization. Representing a query plan as a tree and applying rules to manipulate them. SparkSQL provides a new solution for a wide range of data-driven problems that relies on relational data and relational query processing.

SparkSQL is built as a library on top of Spark, as shown in Figure 2.10, the DataFrame API integrates relational SQL commands with procedural code within Spark, for instance, it is possible to combine declarative SQL queries (from external data sources using JDBC/ODBC, or from existing RDDs) with analytics methods in Spark (e.g. machine learning library). The following example shows how to express a 10-nearest-neighbors query for the point $p = (3.0, 2.0)$ in a dataset named *points* (containing 2D point objects), using the SparkSQL programing interface, source [168].

[4]Spark Programming Guide:
https://spark.apache.org/docs/latest/programming-guide.html

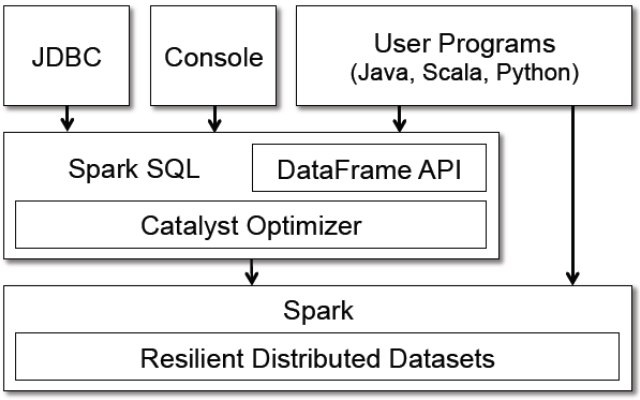


Figure 2.10: SparkSQL Overview. Source [14].

```
SELECT * FROM points
ORDERED BY    (points.x - 3.0)*(points.x - 3.0) +
              (points.y - 2.0)*(points.y - 2.0)
LIMIT 10
```

Discretized Streams (D-Streams): Discretized streams (D-Streams) [183] [182] is a fault-tolerant programming model build on top of Spark framework for data streams processing. D-Streams unifies batch and streaming processing by treating data streams as a series of batch computation within a given time interval. D-Streams store results for each batch process in groups of RDD data structures, so users can manipulate RDDs using transformation and actions functions of Spark.

DataFrame: A DataFrame is a Spark's Dataset (distributed collection of data) organized into named columns. It is equivalent to a table in a relational database, but with richer optimizations through the Spark API. DataFrames can be constructed from a variety of sources such as: structured data files, tables in Hive, external databases, or existing Spark RDDs. A DataFrame is represented by a set of data row. As in relational databases, DataFrames were build to be used with structured and relational data; trajectory data, however, is sequential and unstructured, each trajectory is different in length, number of data records, time interval, sampling rate, and scale to name a few.

2.6 Spatial Data Processing in MapReduce

MapReduce can cope remarkably well with large amounts of data; however, spatial-temporal objects are generally more complex than words and URL strings in common MR applications; thus, it is more difficult to fit spatial and spatial-temporal data into the MR model due its nested and multi-dimensional nature [4] [200]. Nevertheless, several scalable solutions have been proposed to support spatial operations in MR. Existing research utilize either a multi-core *divide-and-conquer* strategy, where each *mapper* is responsible to process a sub-query over a subset of the dataset, while the intermediate results from the *map* are refined by the *reducers*; or utilize spatially-aware partitioning techniques (see

Section 2.1.2), in order to organize the space into disjoint groups of spatially close objects, providing both process decentralization and efficient space pruning; hence reducing I/O and minimizing data transfer across nodes.

The main drawback of the plain *divide-and-conquer* approach is that computational resources may be wasted by processing data blocks that does not contribute for the query result. On the other hand, spatial-aware partitioning strategies in MR can achieve up to 10x faster performance than *divide-and-conquer* by maintaining data locality [44] [48] [200], since only a smaller number of partitions containing query candidates are selected for processing, reducing query latency and avoiding unnecessary I/O. Therefore, most works propose MR algorithms for data partitioning and query processing briefly described as follows:

Partitioning: the *mapper* reads a data record and outputs a key/value pair with a spatial object as *value* and the partition containing the object as *key*, this step is done according to a previously chosen partitioning techniques (e.g. Quadtree, Rtree). The *reducers* group data from the same partition *key* into a final data structure to be stored on disk (e.g. Hadoop's HDFS), or in-memory (e.g. Spark's RDD).

Query Processing: spatial queries are processed in a "filter-and-refinement" fashion, where in the *filter* step candidate partitions are selected using a given spatial index during the *map*, and the *refinement* is done by the *reducers*. For spatial operations that demand scanning the whole dataset (e.g. join), a sub-set of the problem is computed by the *mappers*, while the global result is done by the *reducers*.

In this section we discuss the related work on spatial data and trajectory data using MR. Related work can be divided into four main categories: (1) MR-based solutions for a specific spatial query/problem, (2) unified frameworks/systems for spatial data in MapReduce (Hadoop-based), (3) unified frameworks/systems for spatial data in Spark, and (4) MR-based solutions for trajectory data.

2.6.1 Spatial Queries and Operations in MapReduce

Zhang et at. [186] proposed a nested-loop-based algorithm for k-NN join in two MR phases; the algorithm partitions the two input datasets to join into N equal-sized blocks during the *map* phase; then the N^2 block combinations are send to the *reducers* which perform a local nested loop k-NN join; the second MR phase computes the global k-NN join result. Lu et al. [95] and Akdogan et al. [7] use a Voronoi diagram-based approach to partition the space and index spatial objects based on its closest pivots during the *map* phase, and processing k-NN and RNN queries [7], and k-NN join [95] in iterative *MR* tasks; and outperforms similar MR works based on grid-based partitioning for k-NN query [160] [200] and k-NN join [186]. Overall, Voronoi-based partitioning has been shown to outperform other methods for nearest neighbors search [83] [86] [134].

Zhang et al. proposed a MR algorithm for selection queries using a "divide-and-conquer" approach [188], and spatial join using a grid-based space partitioning and Z-order curve [189], however,

their work use spatial distance only and do not handle load balance. Similarly, VegaGiStore [200] use grid partitioning and Hilbert-order curve in selection queries for better data locality preserving than Z-order curves.

Gupta et al. [63] presents a grid-based approach to process overlap and range joins on MR, but only for the spatial dimension of rectangular objects. Puri et al. [127] proposed a MR-based algorithm for GIS polygonal overlay, with an improvement of the naive "divide-and-conquer" method by creating a local R-Tree in each *map* task, in order to reduce the number of join candidates which are sent to the final *reduce* phase. In [23] an approach is presented for R-tree construction using MR tasks; nevertheless the work construct a separate R-tree for each data partition, without addressing any type of query. CG_Hadoop [44] implements various computational geometry operations in Hadoop, i.e. polygon union, skyline, convex hull, farthest pair, and closest pair. CG_Hadoop uses a three-phase "divide-and-conquer" approach in MR; firtly, CG_Hadoop uses a spatially-aware partitioning to group spatial objects; secondly each partition is processed in parallel by MR tasks and a local result is calculated; finally the local results are sent to a final MR phase to compute the global result; Figure 2.11 illustrates the operations covered in CG_Hadoop.

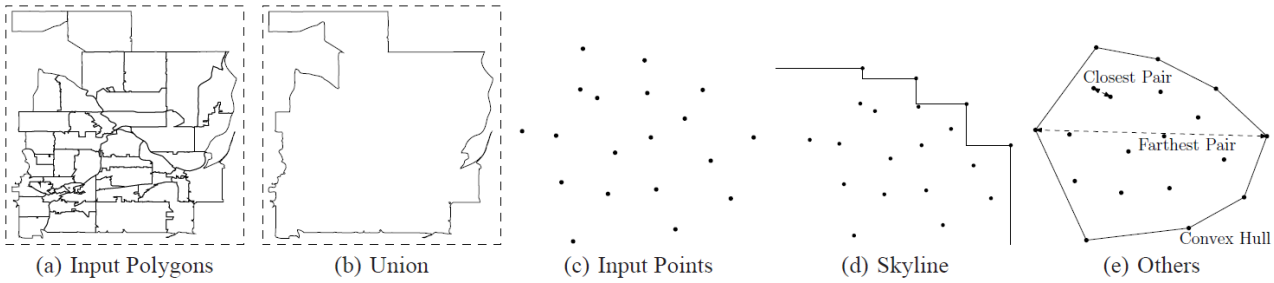


Figure 2.11: Computational Geometry operations covered by CG_Hadoop. Source [44].

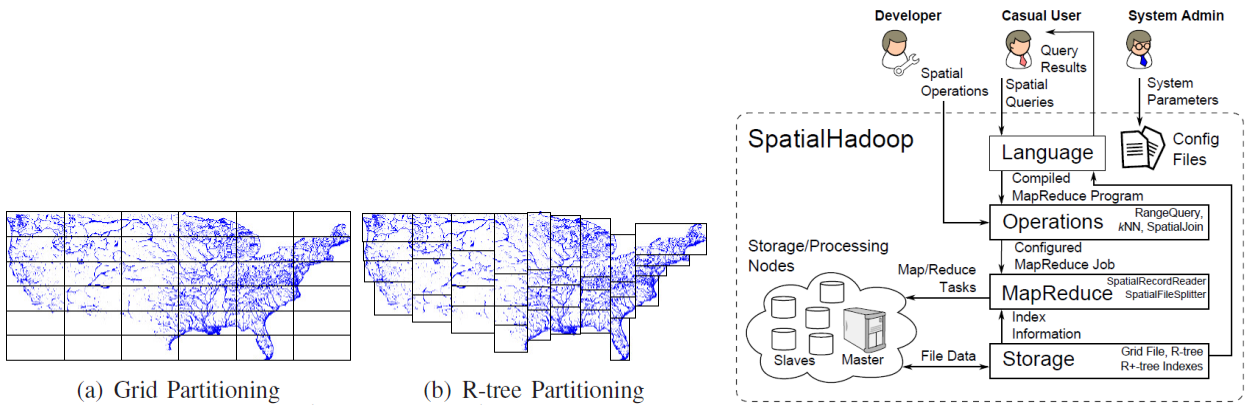
Park et al. [111] proposed a three-phase algorithms for Skyline and Reverse Skyline using MR; in the first phase the algorithm builds a quad-tree from a sample of the dataset to filter out non skyline points; in the second phase the dataset is partitioned according with the previously built quad-tree and computes the local skyline for every candidate partition; the third phase calculates the global skyline from the local results. In [112] the authors extended this approach for uncertain data using a probabilistic function in the second phase.

However, all the aforementioned approaches only focus on the spatial dimension of the data, for points and polygons on disk-based computation only, and do not apply for trajectory queries.

2.6.2 Unified Frameworks for Spatial Data in MapReduce

The second group of existing works provide unified frameworks for spatial queries using MR on top of the Hadoop framework.

SpatialHadoop: SpatialHadoop [48], an extension of Hadoop [68], was built in the core of the Hadoop framework and extends its data types and query language to support and simplify spatial queries [47], and adding new functions to support spatial analysis [46]. Algorithms to cope with computational geometry operations in MR have been proposed and evaluated against traditional algorithms and SpatialHadoop [44], wherein SpatialHadoop outperformed Hadoop and traditional approaches for spatial queries and computational geometry problems by using an extensive set of spatial partitioning structures to improve spatial queries throughput [43], as shown in Figure 2.12(a). Figure 2.12(b) shows the main architecture of SpatialHadoop, the lowest level SpatialHadoop provides data storage in the HDFS using spatial indexing; data partitioning and query processing are done using MR in the MapReduce layer and Operations layers repectively. SpatialHadoop also provides its own query language named Pigeon [47] based on Pig Latin [107] and in compliant with OGC standards; Pigeon provides declarative data types (e.g. `POINT, POLYGON, LINE`) and spatial functions (e.g. `Overlaps(), Distance(), Centroid()`). The open-source HadoopViz [45] [50] is integrated with SpatialHadoop, and provides a framework for big spatial data visualization using MapReduce. HadoopViz can generate images with giga-pixel resolution in various image formats.



(a) Space Patitioning in SpatialHadoop. (b) SpatialHadoop Architecture.

Figure 2.12: SpatialHadoop Overview. Source [48].

Similar to SpatialHadoop, Aji et al. proposed Hadoop-GIS [6] as another MR solution for spatial query analysis and cost-efficient data indexing. Hadoop-GIS provides a real-time spatial query engine (RESQUE), which index data on-demand during query processing. Hadoop-GIS is integrated into Hive [149] data warehouse, hence it allows declarative spatial queries using the HiveQL language (e.g. ST_INTERSECT, ST_UNION). Hadoop-GIS implements the framework SATO [155], which provides heat map visualization and a set of skew-aware partitioning approaches (e.g. R-tree, Hillbert curve, binary split) for load-balanced partitioning and scalable query processing. SATO finds the optimal partitioning strategy by analyzing a small sample of the entire dataset, and then performing a complete partitioning from the sample analysis. Hadoop-GIS have been more specifically used for spatial analysis in medical systems [4] [5] [16], and GPU processing [3].

MD-HBase [103] is a storage system built on top of Hadoop's HBase [69]. Both HBase and MD-HBase are scalable and fault-tolerant key-value-based storage systems built based on Google's

BigTable [26], providing high insert and update throughputs and high availability. Furthermore, MD-HBase also supports kd-Tree and Quadtree partitioning for spatial data, and range selection and nearest-neighbor queries for location-based services.

SHAHED [49] is a MR-based framework for query processing and visualization of large-scale satellite data. SHAHED deploys SpatialHadoop [48] to manage large-scale spatial data, and provides four main functionalities: (1) data cleaning using map data interpolation, (2) quad-tree based partitioning for spatial-temporal satellite data, (3) support for selection and aggregate map queries, and (4) data visualization.

However, all the aforementioned works are disk-based optimized for I/O efficiency, and only support off-line static data partitioning, and does not consider the query-workload. Moreover, they do not support spatial-temporal trajectory queries, since all systems were developed for operations on the spatial dimension of points and polygonal objects only.

AQWA: Similarly, Aly et al. proposed AQWA [12], an adaptive spatial data partitioning based on k-d Tree to support range selection and k-NN search in MR; unlike SpatialHadoop, AQWA provides a dynamic space partitioning which reacts to changes on both the query workload and new incoming data; in AQWA data repartitioning is done so that the cost $Cost(L)$ of processing queries over a set of spatial partitions L is reduced, the partitions may be updated as data is consumed by queries, or when new batch of spatial points are append to the dataset. AQWA uses the cost function $Cost(L)$ to decide whether or not to re-partition the dataset. The cost model is based on the number of points a query has to read $N(p)$ and the number of queries $O_q(p)$ executed over a given spatial partition p. AQWA also provides a time-fading mechanism, giving lower weights to older queries in the cost model, this allows AQWA to alleviate the cost of re-partitioning overhead corresponding to old queries.

$$Cost(L) = \sum_{p \in L} O_q(p) * N(p)$$

In [11], the authors presented Kangaroo, a query-workload-aware system built using AQWA's cost model for range queries. Similar to AQWA, ScalaGiST [94] presented a general purpose solution for query processing and data indexing in MR. ScalaGiST supports B+-Tree and R-Tree like indexes for different types of application, including multidimensional range queries and k-NN which can be applied for spatial data. Although they provide on-line dynamic data partitioning, both AQWA and ScalaGist are disk-based and do not provide support for trajectory data storage and query operations.

2.6.3 Unified Frameworks for Spatial Data in Spark

The third group of existing works provide unified in-memory-based frameworks for spatial queries using Spark framework.

Simba: Simba [168] is a in-memory MR-based system build on top of SparkSQL [14] for spatial Big Data analytics. Unlike disk-based systems such as SpatialHadoop, Simba focus on optimizing per-

formance by extending Spark's RDD to support spatial indexing and spatial operations natively. Simba adds new features to SparkSQL's architecture, as shown in Figure 2.13. Fistly, Simba implements a wide range of popular spatial queries (e.g. range query, k-NN, distance join, and k-NN join), and adds spatial keywords to SparkSQL grammar (e.g. POINT, RANGE, KNN, DISTANCE JOIN), so that users can express spatial operations in a SQL-like fashion, or using the SparkSQL's DataFrame API language. Following, the authors exemplify how to write a 3-NN query of point (4,5) in the *point_data* dataset using Simba.

```
SELECT * FROM point_data
WHERE POINT(x,y) IN KNN (POINT(4,5), 3)
```

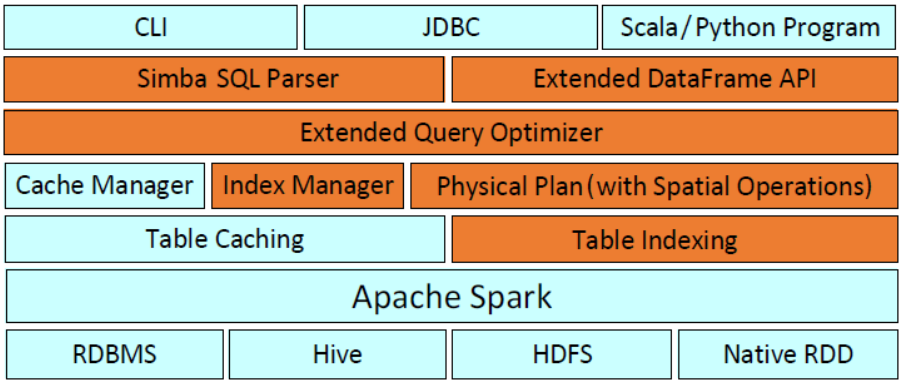


Figure 2.13: Simba Architecture. Source [168].

Simba extends Spark's RDD to support spatial indexing, it introduces a new abstraction, named `IndexRDD` which is essentially an RDD of rows `RDD[row]` (like a table of records), to support user specified indexing (i.e. R-Tree, hash tree, hash map) in order to reduce query latency and increase throughput. `IndexRDD` supports all native RDD operations plus spatial ones.

Partitioning Optimization in Simba: Simba also uses a cost-model, to partition the input dataset so that the data can fit into main-memory, and each partition has roughly same size, in order to provide load-balancing. Simba determines the number of partitions as well as the partition size for different dataset using a cost-function based on the available resources in the cluster. The cost-model is shown as follows, where β is the partition size, λ is a system parameter (i.e. the percentage of available memory allocated for storage, 80% by default), α is the memory reserved for Spark caching, M is the total memory reserved for Spark on each slave node, and c is the number of cores in the cluster.

$$\beta = \lambda((1-\alpha))M/c)$$

Query Optimization in Simba: Furthermore, Simba extends the Spark SQL's *Catalyst* optimizer for spatial queries; when a select condition is submitted to the application, Simba transform the original select condition to the Disjunctive Normal Form (DNF), for instance $(A \wedge B) \vee C \vee (D \wedge E \wedge F)$, and selects the predicates that can be optimized by index to form a new select condition θ; then Simba filters data with θ first using index-based operators, and finally applies the original condition to get the

final answer. Index optimization may improve performance when the predicate is selective,however, it may cause overhead if the selectivity is high. Therefore, if the selectivity of the predicate is higher than a given threshold (80% default), Simba will scan the whole partition instead leverage indexing. Simba makes an estimative of the predicate selectivity based on the partitions MBR whose boundary intersects the query area, and the estimate number of records in the partitions.

However, despite its great contributions, Simba still not provides any support for spatial-temporal trajectories.

Similar to Simba, GeoSpark [175] is another system build on top of Spark to process spatial queries. GeoSpark provides an extension of RDDs for two-dimensional spatial objects, named Spatial RDDs (SRDDS), and uses quad-tree and R-Tree indexing in the SRDD partitioning phase; GeoSpark supports range selection, *k*-NN, and spatial join over its SRDDs of points and polygons (i.e. PointRDD, PolygonRDD). Similarly, SpatialSpark [174] is a system to support range queries and spatial join over points and polygons using Spark; SpatialPark uses uniform grid and k-d tree partitioning to improve query performance over large datasets. Unlike Simba, GeoSpark and SpatialSpark implement few spatial operations, and does not provides a query engine. SparkGIS [16] is another framework build on top of Spark for in-memory distributed spatial data processing. SparkGIS combines Spark with the RESQUE query engine of Hadoop-GIS [6] to support pathology image analysis.

However, none of the aforementioned frameworks consider the query-workload when storing the data in-memory, thus they are not memory-wise, and they do not provide support for spatial-temporal trajectory data.

2.6.4 Trajectory Data in MapReduce

Despite many efforts to process large-scale trajectory and time-series data in parallel, e.g. [27] [41] [53] [73] [93] [121], trajectories of moving objects are difficult to fit into the MR model due to their multi-dimensional and sequential nature. Furthermore, even for single thread processing applications, trajectory management still pose a great challenge (recall Section 2.2). However, some efforts have been done to deal with trajectories using MR.

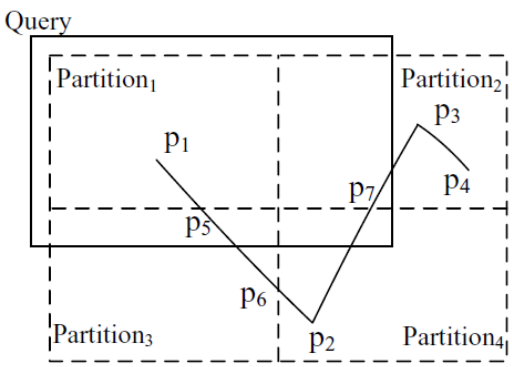


Figure 2.14: Trajectory, partitioning and query in TRUSTER. Source [172].

TRUSTER [172] is a system for trajectory data processing in MR. TRUSTER uses uniform grid

cells for space partitioning, and a 1D tree to index time within each cell. During the partitioning phase in MR, trajectories are split into segments and every segment is assigned to the partition it overlaps with, if a segment spans for more than one grid cell the segment is split according to the cells it overlaps with. During query processing the segments in the partitions containing the query range are selected. However, TRUSTER uses a uniform grid cells partitioning, hence does not handle load balancing. Figure 2.14 shows an example of partitioning and query in TRUSTER. In [98] the authors presented PRADASE, an improvement of TRUSTER. PRADASE index both space and time using a quad-tree-based structure for better load-balancing. PRADASE uses GFS [57] based data storage for replication and dynamic partitioning of temporal dimension. Trajectories are indexed using two spatial indexes to optimize trajectory queries, i.e. PMI and OOI. PMI provides a quad-tree-based space partition with multiple assignment strategy for boundary objects, wheres OOI is used to associate moving objects with their respective trajectories. However, both TRUSTER and PRADASE are disk-based, and do not consider trajectory data preprocessing nor the query-workload.

CloST [146] is a Hadoop-based storage system for spatial-temporal range queries. CloST proposes a new data model and file format to store trajectory data in HDFS. CloST uses a three-level hierarchical partitioning in MR, where in the first level trajectories are grouped into coarse *buckets* according to the moving objects OID; in the second level each *bucket* is partitioned into spatial *regions* using quad-tree; in the third level each *region* is divided into fine-grained 1-D *blocks* of time. Figure 2.15 illustrates the hierarchical partitioning in CloST. Input records from same moving object are grouped together and stored in a table format into the HDFS using *delta* and *running-length* compression. The goal of CloST is to support efficient *single-object queries* (i.e. spatial-temporal selection) and *all-object queries* (i.e. selection by object OID). Although CloST also provides a dynamic partitioning according with the data utilization ratio, it is a disk-based approach, and does not consider trajectory data preparation nor preprocessing.

OceanST [178] is a Spark-based system designed for spatial-temporal Mobile Broadband (MBB) data. OceanST adopts the same hierarchical partitioning of CloST, and provides an additional set of inverted indexes to attributes associated with MBB data (e.g. textual information). OceanST uses Spark to speed up *exact* and *sampling-based* aggregate queries over distributed data (e.g. `count, distinct, max, min, sum, avg`). OceanST also includes an API with some basic spatial-temporal analytics, such as frequent path identification, transportation mode prediction, and activities prediction, for instance. However, OceanST only provides a static off-line data partitioning, hence does not consider the query-workload and it's not memory-wise, moreover, OceanST aims for MBB data, and does not support indexing for similarity search over GPS trajectories, and does not consider data preprocessing.

Another work by Li et. al [87] uses MR to calibrate bus trajectories and identify bus routes directions using a k-NN query; however, they simply run a k-NN on the whole dataset using MR, and do not use any index structure or data partitioning, which negatively affects the performance. Jinno et al. [76] proposed a grid representation of trajectories, and a quad-tree-based search with MapReduce for frequent movement pattern mining; the grid resolution can be modified to identify different types

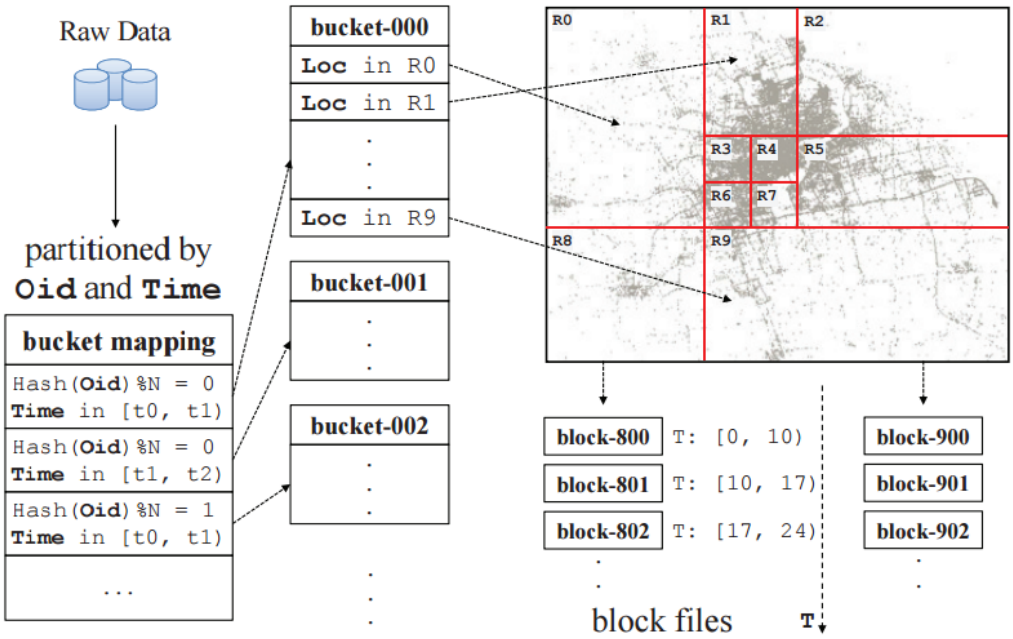


Figure 2.15: Hierarchical partitioning in CloST. Source [146].

of patterns, however they use a lossy representation of trajectories, and can only be applied for few movement patterns identification.

Chapter 3

Trajectory Data Preparation and Preprocessing

In this chapter we present our contribution on trajectory data preparation and preprocessing. In Section 3.1 we introduce a novel script model for trajectory data representation, and designed a system for trajectory data integration and compression. In Section 3.2 we introduce a framework for trajectory data preprocessing using map-matching on top of Spark, in order to achieve data quality with performance and scalability.

3.1 Trajectory Data Integration and Representation

Raw trajectories should go through a series of preprocessing steps before they become suitable for indexing and querying. This chapter introduces a novel parallel system for trajectory data integration and representation, with support for lossless trajectory *Delta* compression, and synthetic trajectory data generation. This system also provides templates for trajectory data representation (e.g. spatial-temporal attributes, textual attributes) providing a single data model for integration of different input datasets. Moreover, this application is responsible to collect statistics of the input dataset (i.e. metadata).

3.1.1 Trajectory Data Loader and Parser

Different GPS devices and transportation companies record and store their data using various formats. Even though GPS data often contains the same spatial-temporal and semantic attributes, describing the moving object's trajectory, the integration of these datasets into a single format and storage platform is yet an issue. Therefore, we deliver a data integration system for simplified loading and preprocessing of trajectory data into a standard text platform; this facilitates data access and processing by any trajectory application using multiple and heterogeneous datasets.

With the increasing of GPS trajectory data volume and sources, large amount of spatial-temporal trajectory data formats have emerged. Therefore, spatial-temporal trajectory data integration is significant to combine data from different sources into a unified format and platform for trajectory data-based applications [195] [199]. We introduce a novel system to represent and integrate spatial-temporal trajectory data from different sources and formats. This system targets researchers and professionals working on trajectory data-driven systems and applications, which often demands the collection of data from several sources in order to perform experiments and trajectory-based analytics. The application parses the input data to a predefined output and compressed CSV format, and stores the formatted data into any of the provided primary storage platforms, i.e., MongoDB [100], HDFS [68], or Local directory. This allows any trajectory-based system to process data from multiple heterogeneous datasets in a user-provided storage platform, without the need of re-implementation.

Current spatial-temporal trajectory data sources generate and store data in a semi-structured textual format, containing the latitude, longitude, and time-stamp of the trajectory coordinates points, along with additional semantic information, which varies from one dataset to another. Furthermore, several independent sensors may be used in different circumstances to collect data [77]. However, it is challenging to interpret and integrate trajectory data from the multitude of textual formats and sensors available, and it is still an issue [138]. Therefore, in order to represent and integrate data from different formats, we firstly introduce the *Trajectory Data Description Format (TDDF)*, a data description format for spatial-temporal trajectory data representation. The TDDF was designed based on a survey on several real GPS trajectory datasets, both public and private, accessible by our research group. Then, based on the user-provided TDDF, our application loads and parses the input data into the selected output data format using lossless Delta compression, in order to reduce the size of the stored data. Our system also generates statistical information (Metadata) about the input datasets, which are used in the Spark algorithms, and can also be used to generate synthetic data for experimental purposes. A data parser was built to convert each data record from the input datasets to the output format provided.

Briefly, the main functionalities of this application are:

- **Load** GPS trajectory data from any traditional textual format, by means of a user-specified Trajectory Data Description Format (TDDF).
- **Parse** the raw data based on the input *TDDF* to one of the system-provided *Output Data Formats* (i.e. output TDDF).
- **Generate** synthetic trajectory data in the default *Output Data Formats* (i.e. output TDDF).
- **Store** the parsed data in a compressed format into any of the primary storage platforms provided, i.e. Local directory, MongoDB, and HDFS.

A data parser was built to convert each data record from the input dataset to the output format provided. Finally, we provide a platform independent GUI for data parsing. Figure 3.1 shows the application GUI. The system can be accessed and download from the project repository [1]

[1] https://github.com/douglasapeixoto/trajectory-data-loader

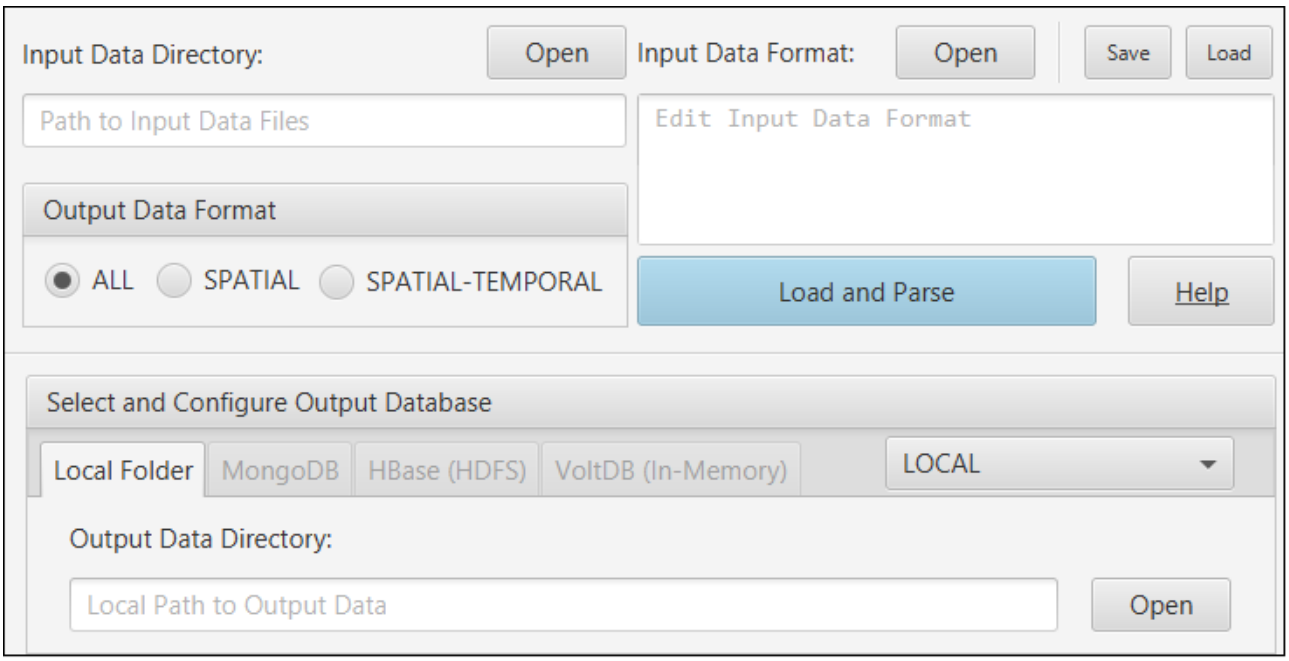


Figure 3.1: Trajectory Data Loader application GUI.

3.1.2 Problem Statement

Due to the myriad of trajectory data formats available, our first goal is data preparation, by formating and integrating a set of input trajectory dataset into a common and simplified format, that is.

Given a set of input trajectory datasets $\mathbb{D} = \{\mathbb{T}_1, \mathbb{T}_2, ..., \mathbb{T}_N\}$, and a set of input trajectory data formats $\{f_1, f_2, ..., f_N\}$, where the format of $\mathbb{T}_i$ is f_i, that is $T_i \rightarrow f_i$. We want to represent and integrate $\mathbb{D}$ using a predefined trajectory data format f_t, such that, for every $\mathbb{T}_i \in \mathbb{D}$, $\mathbb{T}_i \rightarrow f_t$.

To solve this problem, first we need to define the data format f_t that best represent the trajectories in the datasets of $\mathbb{D}$, then for every $\mathbb{T}_i \in \mathbb{D}$ we must convert every trajectory in $\mathbb{T}_i$ from the format f_i into f_t.

3.1.3 System Design

Figure 3.2 introduces the system workflow. Briefly, raw trajectory data is read and parsed based on a user provided input data format, i.e. the *Input TDDF*. The parser identifies trajectory records and attributes from the raw data, and parse the raw data to any of the system provided output data formats, along with a metadata file and the description of the output data format, i.e.e the *Output TDDF* file. The output data can be stored into any of the primary storage platforms provided.

Spatial-temporal trajectory datasets available are organized in basically three manners, (1) each document in the dataset contains one trajectory record, (2) each document contains several records, one per line, (3) each document contains several records in multiple lines separated by a delimiter. Attribute values in a record are separated by a delimiter (such as a comma or semicolon). Attributes are either atomic or multi-valued (i.e., list). We overcome the problem of reading different formats by telling the parser how the records are organized in the dataset, that is, the format, type, and order of

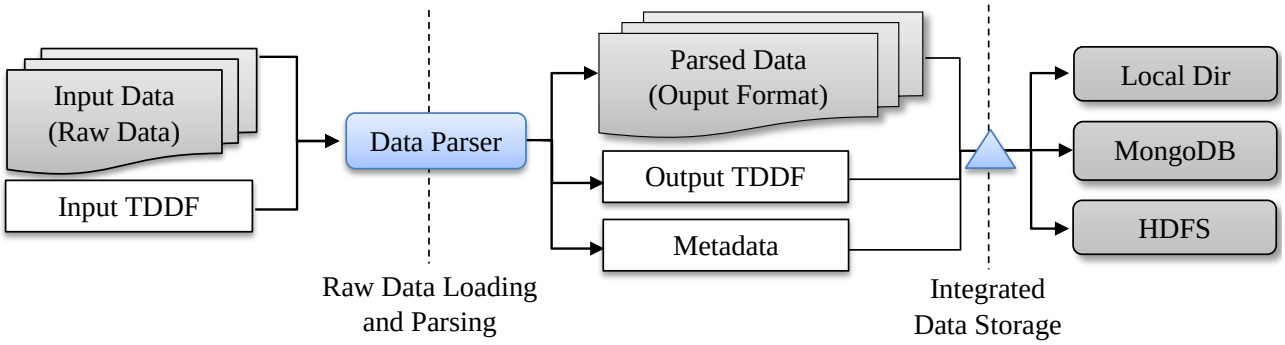


Figure 3.2: Trajectory Data Loader workflow.

each attribute in the trajectory records. In the next sections we describe the architecture, features, and implementation of the proposed data loader and parser system.

3.1.4 Trajectory Data Description Format: TDDF

This application reads trajectory files from any source format; however, different GPS data sources provide different data formats. Therefore, the core of our application implements a file interpreter to extract attributes from the input data files containing GPS trajectory records. In order to do that, the user must specify the format (fields/attributes) of the input data as they appear in the source files, these specifications are provided through the TDDF.

We introduce a set of data description keywords to describe the input data format. The format (fields/attributes) of the input data must be provided as they appear in the source files. The *TDDF* is a user-specified script containing the descriptions of the input data files, similar to a *Data Description Language (DDL)*, assisting the parser to identify trajectory records and attributes. The TDDF scope contains both *attribute* declarations, and *commands* to be executed while parsing the data. We introduce a set of declarative keywords to the TDDF, for both attributes' (*Data Definition Keywords*) and command's (*Data Control Keywords*) declarations. Identifiers and spatial-temporal attributes have a special tag since they represent the core of trajectory data. The scope of the TDDF was designed based on a survey of existing spatial-temporal trajectory formats, in order to cover a wide range of trajectory datasets. Following we describe the *Data Definition Keywords* for attributes' declaration, and *Data Control Keywords* for command's declaration, and give some examples.

TDDF Grammar

Predefined Keywords: Predefined keywords aid the parser to identify important parameters and commands in the input data. Tables 3.1 and 3.2 introduce the list of predefined keywords and their meanings.

Default Command Values: Although necessary for the data interpreter, some commands are provided with a default parameter/value in case they are not provided by the user. Table 3.2 shows the

Keyword	Type	Description
_ID	Attribute Name	Trajectory Identifier
_COORDINATES	Attribute Name	List of Trajectory Coordinates
_X	Attribute Name	Coordinate X (or Longitude) value
_Y	Attribute Name	Coordinate Y (or Latitude) value
_LON	Attribute Name	Coordinate Longitude value
_LAT	Attribute Name	Coordinate Latitude value
_TIME	Attribute Name	Coordinate Time-Stamp
INTEGER	Attribute Type	Integer number
DECIMAL	Attribute Type	Decimal number
STRING	Attribute Type	String character
BOOLEAN	Attribute Type	Logic type (True/False)
CHAR	Attribute Type	Single character
DATETIME	Attribute Type	Date and time (Java DateTimeFormat)
DELTAINTEGER	Attribute Type	Integer delta compressed number
DELTADECIMAL	Attribute Type	Decimal delta compressed number
ARRAY	Attribute Type	Array type (List)
CARTESIAN	Command Value	Cartesian coordinates (x,y)
GEOGRAPHIC	Command Value	Geographic coordinates (longitude,latitude)
LN	Command Value	Line-break
LS	Command Value	Line-space
EOF	Command Value	End-of-File
SPATIAL_TEMPORAL	Output Format	Outputs spatial-temporal attributes only
SPATIAL	Output Format	Outputs spatial attributes only
ALL	Output Format	Outputs all attributes
#	Comment Marker	Line comment symbol

Table 3.1: TDDF Data Definition Keywords.

Keyword	Type	Description	Default Value
_RECORDS_DELIM	Command Name	Data Records Delimiter	LN (Line-break)
_IGNORE_ATTR	Command Name	Ignore Input Attribute	–
_IGNORE_LINES	Command Name	Ignore Input File Line(s)	–
_AUTO_ID	Command Name	Auto generate ID attribute	–
_COORD_SYSTEM	Command Name	Spatial coordinates system	GEOGRAPHIC
_DECIMAL_PREC	Command Name	Precision for decimal numbers	5
_SAMPLE	Command Name	Load a sample of the dataset	1.0 (100%)
_OUTPUT_FORMAT	Command Name	User-specified output format	ALL

Table 3.2: TDDF Data Control Keywords.

command keywords and their respective default values. All keywords are case-sensitive.

TDDF Syntax and Semantic

For each *attribute* of the data record, one must provide the attributes' `NAME`, `TYPE` and `DELIMITER`, separated by space or tab.

NAME: Name of the field/attribute.
TYPE: Type of the field/attribute to read.
DELIMITER: Field delimiter in the input file.

When providing the TDDF script, the user must declare one attribute per line in the exact order they appear in the input files. The parser will read the attributes' value until the given field `DELIMITER` is reached. Attributes' name must be unique in the TDDF.

Commands, on the other hand, are declared in the form `NAME`, and `VALUE`.

NAME: Name of the field/attribute.
VALUE: The command's input parameter/value.

The attribute keyword `_ID` describes the identifier field of each trajectory record. Since in our research not all input datasets provide an ID for the trajectory records, the command `_AUTO_ID` to generate the records' IDs automatically. An example of the `_AUTO_ID` command syntax is given as follows:

_AUTO_ID prefix
Output the ID attribute as "String": "prefix_1", "prefix_2", ...

_AUTO_ID 10
Outputs the ID as attribute "Integer", starting from the given number: 10, 11, 12, ...

Either the trajectory `_ID` attribute field, or `_AUTO_ID`, should be provided in the input TDDF. If both are omitted, the application will use "`_AUTO_ID 1`" by default.

The attribute keyword `_COORDINATES` is a mandatory field, and describes the list of coordinate points of the trajectory records. The `_COORDINATES` must be declared as an `ARRAY` type, followed by the description of the spatial-temporal attributes – i.e. `_X, _Y, _TIME` in `CARTESIAN` system, or `_LON, _LON, _TIME` in `GEOGRAPHIC` system – and any semantic attributes of the coordinate points, in the same order they appear in the input data files. The spatial-temporal fields `_X, _Y, _TIME`, or `_LON, _LAT, _TIME`, in a `_COORDINATES` attribute declaration are mandatory.

The command `_RECORDS_DELIM` tells the parser the final of a data record. In most GPS trajectory datasets in our research, data records are organized by either one trajectory record per file line, that is `_RECORDS_DELIM LN`, one trajectory record per file, that is `_RECORDS_DELIM EOF`, or many records per file separated by a delimiter character or word c, that is `_RECORDS_DELIM c`. The parser will read a data record until the given delimiter is found.

The command `_IGNORE_LINES` tells the parser to ignore the given lines in all input data files. For instance, the following command will ignore the lines 1 to 5 and 7 in the input data files.

_IGNORE_LINES [1-5,7]

The command `_IGNORE_ATTR`, on the other hand, ignores the attribute in the position of its declaration in all data records, and it is followed by the attributer's delimiter. Both `_IGNORE_LINES`

and _IGNORE_ATTR commands are useful, for instance, when not all data records, file lines, or attributes from the input dataset are necessary for the user application.

The command _DECIMAL_PREC tells the parser the number of decimal points d to consider in decimal values, the default value is $d = 5$. Attributes declared as DECIMAL will be converted to a integer number in the format $value * 10^d$, and compressed using a lossless delta-compression to reduce storage space.

The command _SAMPLE tells the data loader to randomly select a sample the input dataset for reading and parsing. The value for sampling must be in the range $]0.0, 1.0]$ which specifies the percentage of data records to read. The _SAMPLE command is particularly useful for large datasets and debugging purposes.

DATETIME values are declared and parsed using Java's *DateTimeFormatter*[2]. DATETIME types must be declared as DATETIME[``pattern''], where "pattern" describes the attribute using the DateTimeFormatter format.

Array Type Syntax: Arrays (or lists) types are declared by specifying the attributes in the array, i.e. attributes' NAME, TYPE and DELIMITER, the general syntax Array declaration is:

ARRAY(NAME TYPE DELIMITER ...)

Arrays can be single-valued or multi-valued (e.g. objects) of any of the pre-defined data types, the parser will read the parameters until the given field delimiter is reached. Attributes in the array are specified in the exact order they appear in the source file, similar to any other attribute declaration. Following are some examples of array type declaration for _COORDINATES field.

Example 1: Trajectory coordinates as an array/list of spatial-temporal points, comma separated.

```
ARRAY(  _X      DECIMAL  ,
        _Y      DECIMAL  ,
        _TIME   INTEGER  , )
```

Example 2: Trajectory coordinates as an array/list of spatial-temporal points, with *weight* and *type* attributes, one coordinate per file line, separated by semicolon.

```
ARRAY(  _X      DECIMAL  ;
        _Y      DECIMAL  ;
        _TIME   INTEGER  ;
        weight  DECIMAL  ;
        type    STRING   LN )
```

[2]https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html

3.1.5 Output Data Format

After the input data is parsed, the data in the new format is stored into any of our primary storage platforms (i.e. Local directory, MongoDB [100], HDFS [68]), in the output format of choice, along with the *Output TDDF* and a Metadata file, containing information and statistics about the input trajectory dataset, such as number of records, statics about the speed, length, duration, sampling rate, and coverage of the trajectory records. The system generated *Output TDDF* file, on the other hand, contains the specifications of the output data, that is, the `NAME` and `TYPE` of all attributes in the output data.

Three different output formats are provided, using a *Feature Selection* approach, namely `SPATIAL`, `SPATIAL-TEMPORAL`, and `ALL`. The output formats follow a CSV (comma separated values) style. Attribute values are separated by semicolon, and array items are separated by comma. The output documents contain one trajectory record per line. Documents can also be output as BSON [3] documents in MongoDB. Furthermore, to reduce storage consumption, the spatial-temporal attributes in the list of coordinates are delta-compressed. The records' attributes are always in the order:

_ID;_COORDINATES;_OTHER_ATTRIBUTES

Following we describe the system provided output data formats.

SPATIAL: In this output format, records contain the trajectory `_ID` and the list of spatial attributes of the `_COORDINATES` only. This format is useful for applications that does not demand processing over the temporal attributes of the trajectories.

SPATIAL-TEMPORAL: In this output format, records contain the trajectory `_ID` and the list of spatial-temporal attributes of the coordinates only. This output format contains the most basic information of trajectories, commonly used in spatial-temporal queries and mining applications.

ALL: In this output format, records contain the complete set of attributes specified in the *Input TDDF*, that is, the trajectory ID, the list of trajectory coordinate points (with all provides coordinate attributes), and the list of semantic attributes of the trajectory. This is the default output format.

3.1.6 Primary Storage Platforms

Since our goal is to work over large-scale trajectory data, mainly on top of Spark, we employ two widely known storage platforms for large-scale data in our application to store the parsed data, i.e. MongoDB, and HDFS. All storage platforms are scalable and open-source, and provide easy data access to Spark applications. Our system is can also output the parsed data to a local directory, if no storage system is required by the user.

[3] https://www.mongodb.com/json-and-bson

MongoDB: MongoDB [100] is a document-oriented database program for large-scale data. MongoDB uses JSON-like documents with schemas. The document model maps to the objects in the application code, making data easy to query. Furthermore, MongoDB is a distributed database at its core, thus it provides high availability and horizontal scaling. Our application stores the parsed data in MongoDB using CSV document format.

HDFS: HDFS is the Hadoop's File System [68], it stores a distributed way to store data in blocks shared across the cluster, and provided very high data read and write. HDFS makes it easy to store and maintain large amounts of data in a distributed way, and provides fault-tolerance using replication, and I/O in batches.

3.1.7 Case Study

We present a set of case studies using real GPS trajectory datasets. We demonstrate how our application can be used to integrate data from different sources and formats into a single format. For each case, we provide an overview of the input raw data, as well as the Input and Output TDDF scripts, and the parsed data. The sources of the data files, as well as some attribute values, will be omitted for privacy reasons. For the sake of simplicity, and to demonstrate how our system can be used to integrate datasets into a common format, we output all datasets using the `SPATIAL_TEMPORAL` output format.

CASE 1: This dataset contains one GPS trajectory record per file. The first six lines of each file contains some descriptions about the source dataset, and can be ignored. The remainder lines contains the list of trajectory coordinates, one coordinate per line. Coordinates contain both spatial-temporal and semantic attributes. An overview of the dataset records and its corresponding TDDF are given below.

Input Trajectory Data 1:

```
40.008304,116.319876,0,492,39745.0902662037,2008-10-24
40.008413,116.319962,0,491,39745.0903240741,2008-10-24
.  .  .
40.009209,116.321162,0,84,39745.1160416667,2008-10-24
```

Input TDDF Script 1:

```
# One record per file
_RECORDS_DELIM   EOF
# Coordinates in Long/Lat
_COORD_SYSTEM    GEOGRAPHIC
# Lines 1 to 6 of each file can be ignored
```

```
_IGNORE_LINES     [1-6]
# Auto generate ID with prefix 'db1_t'
_AUTO_ID          db1_t
# The list of coordinates in each trajectory record.
# Field 1:  Latitude in decimal degrees.
# Field 2:  Longitude in decimal degrees.
# Field 3:  All set to 0 for this dataset.
# Field 4:  Altitude in feet.
# Field 5:  Number of days since 12/30/1899, with fractional part.
# Field 6:  Date as a string.
# Field 7:  Time as a string.
_COORDINATES      ARRAY(_LAT          DECIMAL    ,
                        _LON          DECIMAL    ,
                        zeroVal       INTEGER    ,
                        alt           INTEGER    ,
                        timeFrac      DECIMAL    ,
                        _TIME         DATETIME("yyyy-MM-dd") LN) EOF
```

CASE 2: This dataset contains several GPS trajectory record per file, every record is delimited by the character #. The first line of each record contains a set of semantic attributes of the trajectory, followed by the list of trajectory coordinates, one coordinate per line. Coordinates contain both spatial-temporal and semantic attributes. An overview of the dataset records and its corresponding TDDF are given below.

Input Trajectory Data 2:

```
#,1,3/2/2009 9:23:12 AM,3/2/2009 10:02:17 AM,10.4217737338017 km
3/2/2009 9:23:12 AM,39.929961,116.355872,23570
3/2/2009 9:23:42 AM,39.926785,116.356007,23526
.  .  .
3/2/2009 10:02:17 AM,39.950725,116.295991,27942
#,2,3/2/2009 10:04:14 AM,3/2/2009 10:56:23 AM,13.1721183785493 km
3/2/2009 10:04:14 AM,39.969738,116.288209,32482
3/2/2009 10:04:44 AM,39.973138,116.288661,13208
.  .  .
3/2/2009 10:56:23 AM,39.99992,116.352966,37268
```

Input TDDF Script 2:

```
_RECORDS_DELIM    #
```

```
_COORD_SYSTEM    GEOGRAPHIC
# Creates new IDs with prefix 'db2_t'
_AUTO_ID         db2_t
# Ignore the first empty attribute, and the integer ID
_IGNORE_ATTR     ,
_IGNORE_ATTR     ,
timeIni          STRING   ,
timeEnd          STRING   ,
length           STRING   LN
_COORDINATES     ARRAY(_TIME   DATETIME("M/d/yyyy HH:mm:ss a")  ,
                       _LAT    DECIMAL                         ,
                       _LON    DECIMAL                         ,
                       alt     INTEGER                         LN)  #
```

CASE 3: This dataset contains several GPS trajectory records per file, one record per file line. The dataset contains trajectories from cars, with the list of trajectory coordinates, and a set of semantic attributes. This dataset had been used for map-matching, hence the coordinate points also contain semantic attributes regarding map-matching. A record in this dataset, corresponding to a single line in the input file, and its corresponding TDDF are given below.

Input Trajectory Data:

```
10000018_1427933750,10000018,1,27|27|27|19,3639865:0:57:114.33708:
30.50130:1427933750|3639862:6:59:114.33715:30.50128:1427933759|3624382:
46:33:114.33728:30.50168:1427933759|3624382:98:12:114.33742:30.50213:
1427933772|3624382:131:17:114.33752:30.50242:1427933778|3630066:0:35:
114.33752:30.50242:1427933772|3630066:0:17:114.33752:30.50242:1427933778
```

Input TDDF Script:

```
_RECORDS_DELIM   LN
_COORD_SYSTEM    GEOGRAPHIC
_ID              STRING   ,
# The moving object which generated this trajectory
sourceId         INTEGER  ,
# Car type:  personal car=1, taxis=2, others=0
carType          INTEGER  ,
citySequence     ARRAY(cityId   INTEGER   |)   ,
# Information of each mapped points to each link, including linkID,
```

```
# the distance between each mapped point, the distance of mapping,
# longitude, latitude, time
_COORDINATES     ARRAY(linkID       INTEGER   :
                       oDistance    INTEGER   :
                       mDistance    INTEGER   :
                       _LON         DECIMAL   :
                       _LAT         DECIMAL   :
                       _TIME        INTEGER   |)    LN
```

For all three cases, the output TDDF is the following, since in all cases the input datasets have been parsed to the same output format.

Output TDDF Script:

```
_OUTPUT_FORMAT    SPATIAL_TEMPORAL
_COORD_SYSTEM     GEOGRAPHIC
_DECIMAL_PREC     5
_ID               STRING
_COORDINATES      ARRAY(_LON DECIMAL _LAT DECIMAL _TIME INTEGER)
```

The output formated data, in CSV and BSON documents, for the three datasets is the following. Notice that now all datasets are in the same format `SPATIAL_TEMPORAL`.

Output Trajectory Data (.csv):

```
db1_t_1;11631987,4000830,1224814199000,8,10,5000
db2_t_1;11635587,3992996,1235985792000,13,-318,30000
db2_t_2;11628820,3996973,1235988254000,46,340,30000
1018_1450,11433708,3050130,1427933750,7,-2,9
```

Output Trajectory Data (.bson):

```
{_id : "db1_t_1", _coordinates : [11631987,4000830,1224814199000,
8,10,5000]}
{_id : "db2_t_1", _coordinates : [11635587,3992996,1235985792000,
13,-317,30000]}
{_id : "db2_t_2", _coordinates : [11628820,3996973,1235988254000,
46,340,30000]}
{_id : "1018_1450", _coordinates : [11433708,3050130,1427933750,
7,-2,9]}
```

Notice that the output format for attributes is slightly different than that in the input format. Since the output data is always in a CSV-like style, where all attributes are semicolon-separated, and array items as comma-separated by default, the delimiters in the output TDDF are therefore omitted. The output TDDF can be used by any trajectory data reader implementation as the common format of all parsed datasets.

CASE 4 (Unsupported formats): Despite our efforts to provide a universal parser, we understand that some data formats may still not fit perfectly in our parser. However, some painless data preprocessing can be done in the raw data to make it fits our model, and thus be integrated into a common format. For instance, our research group had access to a dataset collected by a private bus company, which collected the GPS locations of all their buses after certain time interval, and stored all GPS coordinates collected at the same time together in a text file. Consequently, the GPS coordinates for a given bus trip were spread across multiple files. Since the GPS records also contained the buses IDs and trip IDs, we simply had to perform a quick sort-and-aggregate algorithm to group coordinates of a same bus and trip into the same file sorted by time-stamp. After that, the trajectory records could be easily parsed by our application.

3.1.8 Metadata

While parsing the data, the application generate statistical information about the dataset. The complete set of dataset attributes in the output Metadata file is shown in Table 3.3:

3.1.9 Summary

In this section we introduced a novel model for spatial-temporal trajectory data integration and representation; in addition, a system using our proposed model was developed. Our application interprets and integrates trajectory data from several textual formats into a standard format, using a novel Trajectory Data Description Format (TDDF) designed from a research on real-world datasets; our system outputs the integrated data into a user-specified storage platform, in order to assist researchers and developers working on trajectory data-driven applications.

Metadata	**Description**
NUM_FILES	Number of files in the input dataset
NUM_ATTRIBUTES	Number of attributes in the input trajectories
NUM_COORD_ATTRIBUTES	Number of coordinates' attributes
NUM_TRAJECTORIES	Total number of trajectories in the input dataset
NUM_POINTS	Total number of points/coordinates in the input dataset
MIN_PTS_PER_TRAJECTORY	Minimum number of points per trajectory
MAX_PTS_PER_TRAJECTORY	Maximum number of points per trajectory
AVG_PTS_PER_TRAJECTORY	Average number of points per trajectory
STD_PTS_PER_TRAJECTORY	Standard Deviation of the number of points per trajectory
MIN_TRAJECTORY_LENGTH	Minimum trajectory length in the dataset
MAX_TRAJECTORY_LENGTH	Maximum trajectory length in the dataset
AVG_TRAJECTORY_LENGTH	Average trajectories length in the dataset
STD_TRAJECTORY_LENGTH	Standard Deviation of the trajectories length
MIN_TRAJECTORY_DURATION	Minimum trajectory duration in the dataset
MAX_TRAJECTORY_DURATION	Maximum trajectory duration in the dataset
AVG_TRAJECTORY_DURATION	Average trajectories duration in the dataset
STD_TRAJECTORY_DURATION	Standard Deviation of the trajectories duration
MIN_TRAJECTORY_SPEED	Minimum trajectory speed in the dataset
MAX_TRAJECTORY_SPEED	Maximum trajectory speed in the dataset
AVG_TRAJECTORY_SPEED	Average trajectories speed in the dataset
STD_TRAJECTORY_SPEED	Standard Deviation of the trajectories speed
MIN_SAMPLING_RATE	Minimum trajectory sampling rate in the dataset
MAX_SAMPLING_RATE	Maximum trajectory sampling rate in the dataset
AVG_SAMPLING_RATE	Average trajectories sampling rate in the dataset
STD_SAMPLING_RATE	Standard Deviation of the trajectories sampling rate
MIN_X	Minimum value of X/Longitude in the dataset (coverage)
MIN_Y	Minimum value of Y/Latitude in the dataset (coverage)
MIN_T	Minimum value of Time-Stamp in the dataset (coverage)
MAX_X	Maximum value of X/Longitude in the dataset (coverage)
MAX_Y	Maximum value of Y/Latitude in the dataset (coverage)
MAX_T	Maximum value of X/Longitude in the dataset (coverage)

Table 3.3: Metadata Descripion.

3.2 Parallel Map-Matching at Scale

Map-matching is a problem of matching recorded GPS trajectories to a digital representation of the road network. GPS data may be inaccurate and heterogeneous, due to limitations or error on electronic sensors., as well as law restrictions. How to accurately match trajectories to the road map is an important preprocessing step for many real-world applications, such as trajectory data mining, traffic analysis, Smart cities and rout,es prediction. However, the high availability of GPS trajectories and map data challenges the scalability of current map-matching algorithms, which are limited for small datasets since they focus only on the accuracy of the matching rather than scalability. Therefore, we propose a distributed parallel framework for efficient and scalable offline map-matching on top of the Spark framework. Spark uses distributed in-memory data storage and the MapReduce paradigm to achieve horizontal scaling and fast computation of large datasets. Spark, however, is still limited for dynamic map-matching, and memory consumption in Spark can be an issue for very large datasets. We develop a framework to allow map-matching on top os Spark, while achieving horizontal scalability, memory-wise usage, and maintaining the accuracy of state-of-the-art matching algorithms by: (1) We combine a sampling-based Quadtree spatial partitioning construction and batch-based computation to achieve horizontal scalability of map-matching, as well as reduce cluster memory usage. (2) We employ a safe spatial-boundary approach to preserve matching accuracy of boundary objects. (3) In addition, a cost function for the distributed map-matching workload is provided in order to tune the framework parameters. Our extensive experiments demonstrate that our framework is efficient and scalable to process map-matching on large-scale data, while keeping matching accuracy and low memory usage.

3.2.1 Introduction

Map-matching is the process of matching recorded GPS trajectory observations to road segments on a digital map. This process is useful in applications such as intelligent transportation systems (ITS), traffic analysis, smart cities, and routes recommendation, to name a few. Since GPS records can be incomplete, inaccurate and noisy due to connection problems and signal loss, urban canyons, sparse collection rates, and law restrictions, etc., GPS trajectories may not accurately reflect the location of moving objects. Therefore, map-matching is a process to ease the uncertainty and improve the accuracy of trajectory data analysis by matching the GPS records to the logical model of the real world [193]. The large amount of GPS trajectory data available, however, has introduced a new problem of how to match massive amounts of both map and trajectory data in an efficient manner, since traditional map-matching algorithms focus on the accuracy of the matching rather than performance and scalability. Moreover, map data availability has also increased, for example, the OpenStreetMap (OSM) [108] releases a weekly version of the Map of the World, currently with over 700GB of uncompressed data.

Offline map-matching methods use the knowledge of the complete trajectory geometry and its semantic attributes (e.g. speed, direction) to find its best match on the road network, and commonly needs to be performed only once for the entire dataset [193]; unless a whole new set of trajectory data

is acquired for elsewhere; or there is a need for re-processing the original data when more accurate algorithms become available, or most commonly, when a new and updated version of the road map is available. Therefore, offline map-matching plays a key role on trajectory pre-processing by improving data quality and reducing uncertainty.

The overall approach for map-matching is to take recorded serial location points (e.g. GPS coordinates), and relate them to edges in an existing road network graph. However, this approach can quickly became cumbersome for large trajectory and map datasets, since every GPS point record has to be compared with every road edge. Nevertheless, map-matching computation is intrinsically parallelizable. For instance, [72] [147] [150] [167] decompose the data space using spatial data structures (i.e. Grid, Quadtree), then co-group both map and trajectory data by containing spatial partitions, and perform the matching in each spatial partition in a parallel fashion, achieving orders of magnitude speed up. Besides, with the increasing demand for low-latency services over large scale data, a trajectory-based system should provide good scalability and fast response for map-matching. To address this important issue, an alternative is to partition both trajectory and map data into self-contained partitions that can be processed in a fault-tolerant distributed manner, while storing data in main-memory to reduce I/O cost, therefore improving map-matching performance and scalability [156]. In this work, we leverage the parallelizable property of map-matching computation with Spark and Quadtree space partitioning to achieve both scalability and performance speed up. The proposed framework was built to achieve both speed up, by means of parallel computation and in-memory data storage, and scalability using spatial-aware partitioning and distributed data storage and computation. We also focus on memory usage, since the framework is developed on top of an in-memory data structure (i.e. Spark RDD).

Existing works focus either on the matching accuracy or its performance. Accuracy-driven algorithms such as [71] [92] [101] [164], can achieve high accuracy, but are limited to small datasets, since they focus on the accuracy of the matching rather than its performance and scalability, iterating through the entire dataset to find the best match, thus facing performance deterioration as the dataset grows. Performance-based algorithms such as [72] [147] [150] [167], on the other hand, consider spatial partitioning and parallel processing to speed up map matching computation, but do not account for load balancing and memory usage, and are limited for disk-based computation. Furthermore, due to different density of trajectory data distribution in urban areas, we must account for load-balancing when partitioning the data space for parallel processing.

Frameworks like Spark [181] can fill the gap between performance and scalability, since Spark is an in-memory based framework, and supports distributed parallel computation. Spark have been used in a handful number of data-intensive analytics, including large-scale spatial databases [168] [175]. We implement our solution on top of Spark's RDD, which provides a robust distributed data structure for MapReduce tasks in main-memory. However, since Spark is a distributed and in-memory storage based framework, we must account for workload balancing and main-memory usage. Existing systems for spatial data using Spark and MapReduce [6] [48] [168] employ balanced partitioning structures, such as Quadtree, k-d Tree, and STR-Tree, to provide workload balancing. Optimizing load-balancing

and memory usage are essential to a good Spark algorithm. The main limitations and challenges of large-scale map-matching using Spark include:

- **Load Balancing and Dynamic Spatial Data Partitioning:** Since Spark is designed for parallel distributed computation, we must account for data partitioning and load balancing, which are not directly supported for spatial data in Spark. Existing works for map-matching using MapReduce [72] [150] employ uniform Grid space partitioning to organize the dataset into self contained partitions.However, they do not account for load balancing, which is essential in Spark to reduce contention and communication cost. Workload balancing can greatly reduce the cost of map-matching in Spark, as we demonstrate in our experiments. However, balanced space partitioning structures should be built in a dynamic fashion as the data is consumed. In a parallel distributed environment, such as Hadoop, the processes need to exchange data through the network after the shuffle phase, to aggregate data in the same partition, which increases network cost. With spark the problem is even more challenging, since the Spark's RDD data structure is read-only, which means that to create a dynamic data structure with spark using the entire dataset would demand to build a new RDD on every iteration of the dynamic process, which is both memory and computationally expensive. Thus, related works based on dynamic spatial partitioning such as [167] cannot be applied directly, since we need to find the best partitioning schema beforehand.

- **Memory Consumption and Replication of Boundary Objects:** Since Spark is an in-memory storage framework, mainly for commodity hardwares, the amount of memory available in the cluster may be limited, and hence does not comfortably fit the entire map and trajectory datasets. For instance, the fastest storage level of Spark stores the dataset in memory with replication to speed up data recovery in case of node failure; this can quickly exhaust the cluster memory available. Hence, memory consumption must be taken into account. Furthermore, both trajectory and road map segments can extend for multiple spatial regions, thus we must account for boundary objects when partitioning the datasets. The easiest solution is to replicate boundary objects to all intersecting partition; for in-memory frameworks like Spark, however, this is undesirable, since replication will increase in-memory storage. To avoid replication, existing work split line segments according to their intersecting partitions, however, due to temporal sparseness of record points and GPS noise, this approach is prone to boundary points mismatch.

Contributions and Novelty: In this chapter we propose a Spark-based framework for large-scale map-matching. We leverage the distributed in-memory nature of Spark for scalable and fast processing of offline map-matching. We provide a sampling-based Quadtree partitioning for load-balancing using a cost-model to allow Spark to use a dynamic spatial data structure. Furthermore, we apply a batch-based data loading and processing to reduce memory consumption. In addition, we employ boundary extension using an empirical evaluation for accuracy maintenance as well as replication

reduction. Finally, we provide experimental evaluation and study of parameters of the proposed framework. The key contributions can be summarize as follows:

1. **Cost-Function:** We provide an estimative of the distributed map-matching workload cost, in order to tune the system parameters and optimize the sampling-based data partitioning.

2. **Cost-based Spatial Partitioning:** We employ Quadtree partitioning for trajectory and map data. Quadtree has been previously applied for parallel map-matching with good performance outcomes [167]; moreover, our experiments demonstrates that Quadtree provides an efficient and fairly uniform space partitioning when compared with other commonly used dynamic structures, such as k-d Tree and STR-Tree, achieving better performance and scalability. Since building a dynamic spatial index model from a large dataset can be cumbersome, and a data partitioning model must be provided to Spark beforehand, we address this limitation by providing a sampling-based quad-index construction using a cost-based model. Finally we co-partition both map and trajectory data using the quad-index model into the Spark's RDD.

3. **Batch Loading and Map-Matching:** Once the map data is loaded, trajectory records can be matched independently, thus we provide a batch-based loading and processing of the input trajectory dataset to reduce distributed memory consumption, specially in situations where the cluster memory size is a constraint.

4. **Empirical Boundary Replication:** In addition, we employ a safe boundary threshold for segmentation and replication as proposed in [167] to reduce uncertainty. However, previous works did not evaluate the choice of the threshold value. Therefore, we conduct a set of experiments to find the appropriate boundary threshold which does not affect map-matching accuracy, yet reduce the number of replication, hence memory consumption.

5. **Experimental Evaluation:** Finally we provide an evaluation study on the accuracy of our approach, and the performance and scalability of spatial-aware map-matching using different spatial data structures on top of Spark, and comparing our work with a state-of-the-art technique. Our experiments demonstrate that our approach can achieve efficient and scalable map-matching processing.

3.2.2 Map-Matching Algorithms

There are two main algorithmic approaches for map matching in the literature, Local [8] [28] [81] [165], and Global [19] [90] [92] [101] [125]. In short, the three main steps followed by map-matching algorithms are: (1) identifying a set of candidate edges in the road graph within a given radius from the location point, then (2) calculating the weight for each candidate edge (e.g. shortest distance between the point and the edge), and finally (3) retrieving the edges that maximize the weight.

Local (or *incremental*) algorithms only consider the trajectory and the road network geometries to relate a trajectory point to its nearest edge (point-to-edge) in the road map. This method is simpler and

faster, and more commonly used in on-line map matching, since they rely on the previous trajectory points observations only, which makes it more difficult to use statistical models on the trajectory topology. However, due to measurement errors and GPS inaccuracy, this approach is prone to error (i.e. point mismatch). Wei et al. [164] provided a comparison between local and global map-matching algorithms, and discovered that local algorithms performed poorly, specially due to Y-splits on road networks. For instance, in Figure 3.3 while there are two possible matching candidates, e_2 and e_3, for point p_3, e_3 is the most obvious edge to match, since its next connecting point p_4 is better matched with e_3, and real moving objects are more likely to follow a direct path [92]. Therefore, the best match for trajectory T is the path $P = \{e_1, e_3\}$ connecting v_1 to v_3.

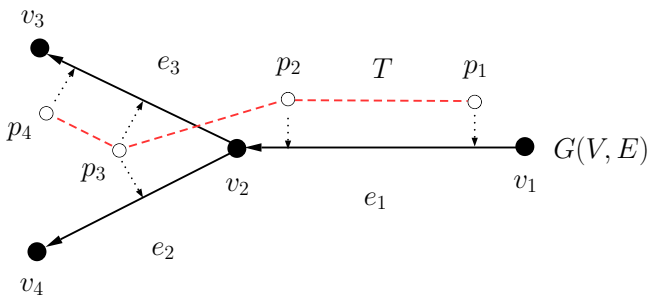


Figure 3.3: Example of road network graph $G(V,E)$, with edges $e_{[1..3]}$ and vertexes $v_{[1..4]}$; and a GPS trajectory T (red dotted) with four coordinate points $p_{[1..4]}$ to be matched with the road network.

Global algorithms, on the other hand, take into account the geometry and other features of the trajectories and the road network, such as speed, topology, the connectivity between points and edges, and the road network speed limits, in order to find the best match of a trajectory on the road network, thus easing the uncertainty. Global algorithms are mostly used in offline map matching, and use future observations to better match the trajectories correctly. . These methods make use of statistical models (e.g. Hidden Markov Model [101], and spatial-temporal analysis [92]), and sacrifice performance to achieve better accuracy. Offline map-matching plays a key role on trajectory pre-processing by improving data quality and reducing uncertainty when whole new trajectory data, or new and more accurate map data, became available.

3.2.3 Problem Statement

In this section we formally describe the problem of Map Matching, firstly introducing some background knowledge.

(Trajectory) A trajectory T of a moving object is a sequence of spatial-temporal points, where each point is described as a triple (x,y,t), where (x,y) are the spatial location of the moving object, such as its *latitude* and *longitude* coordinates, at a time t, that is, $T = [(x_1,y_1,t_1),(x_2,y_2,t_2),...,(x_n,y_n,t_n)]$ in a two-dimensional space, where n is the number of sample points, and $t_1 < t_2 < ... < t_n$.

A trajectory describes the motion history of any kind of moving object, such as people, animals and natural phenomena. Trajectories of moving objects are continuous in nature, but captured and

stored as a collection of spatial-temporal points by GPS devices. A discrete representation of trajectory is shown in Figure 3.4.

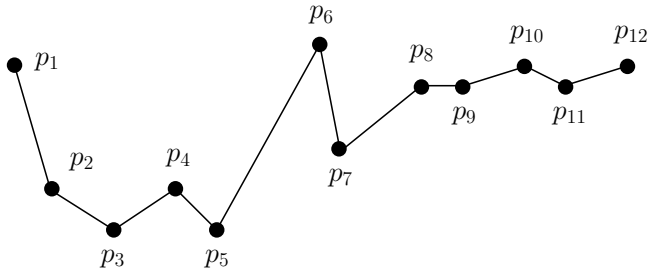


Figure 3.4: Example of trajectory as a discrete sequence of spatial-temporal points.

(Road Network) A road network is a directed graph $G(V,E)$ representing the digital map of streets and roads of a geographic region, where each edge $e \in E$ represents a road segment in the graph, and each vertex $v \in V$ of the graph represents the intersections and end-points of the road segments.

(Road Edge) A road segment $e \in E$ is a directed edge from a starting vertex $v_i \in V$ to an ending vertex $v_j \in V$ in a road network graph $G(V,E)$, and associated with a list of intermediate points that describes the road polyline.

In digital map representation, both edges and vertexes in the road network graph are associates with an ID, and a set of semantic attributes, such as *speed* and *length*.

(Road Path) A road path P is a set of connected road edges, $P = \{e_i, ..., e_j\} \in G(V,E)$, connecting two locations v_p to v_q of $G(V,E)$.

(Map-Matching) Given a trajectory T, and a road network graph $G(V,E)$, map-matching is the problem of how to match T to a path P of $G(V,E)$.

In this work we focus on large scale map-matching.

Large-Scale Map-Matching: Given a large set of GPS trajectories $\mathbb{T}$, and large road network graph $G(V,E)$, our goal is to match every trajectory $T_i \in \mathbb{T}$ to a path P of $G(V,E)$ in an efficient, scalable, and memory-wise manner; that is, we want to maximize performance and scalability of large-scale map-matching, and minimize memory consumption at the same time.

We perform large-scale map-matching on top of Spark in order to achieve high performance and scalability. However, since Spark is an in-memory-based framework, performing any operation on top of Spark should be memory-wise, for instance, for map-matching both datasets may be too large to comfortably fit in the cluster memory. Furthermore, we should account for load balancing and communication cost, since a good data partitioning is a key point in distributed computation. Moreover, both trajectory and map data are difficult to fit into the Spark MapReduce computation model, since Spark does not natively support spatial data indexing and partitioning.

Map-Matching in Spark: Most of existing Spark-based systems, such as [168] [175], only deal with points and polygons queries; furthermore, those works do not focus on memory usage. In this work we exploit the in-memory nature and distributed parallel properties of Spark for scalable offline

map-matching. In addition, we employ a sampling-based partitioning using on a cost model to cover the Spark limitation on dynamic partitioning.

3.2.4 Map-Matching Workload

We provide an estimative of the workload for the distributed map-matching problem based on the following observations:

Execution time and Partitioning: In a nutshell, the baseline cost C_M of map-matching for a given matching algorithm of function f, can be estimated as the number of trajectory points m against the number of road edges n to match, i.e. $C_M = f(m * n)$, since for every trajectory point one must find the best match node. Notice that the function f refers to the map-matching algorithm employed; our framework was built to use state-of-the-art map-matching algorithms as "blackbox", however, every map-matching algorithm has it own computational cost/complexity based on the number of data records to process, which is expressed by the function f.

Nevertheless, since map-matching computation is intrinsically paralellizable, we can greatly decrease the computational cost C_M and improve scalability by co-partitioning the input map and trajectory dataset using some spatial partitioning method, and perform map-matching in each partition in parallel. Equation (3.1) depicts the estimate cost for spatial-aware map-matching C_M^P in parallel,

$$C_M^P = C_{index} + \frac{f(m * n)}{(B * U)} + C_{pos} \tag{3.1}$$

where B is the number of data partitions/blocks, and U is the number of processing units (supposing all units with same computational power).

C_{index} is the cost of building the spatial index model, and partitioning the input datasets. The cost of building the spatial index depends on both the index strategy employed (e.g. balanced or static index), and the number of data records used to build the index model. The data partitioning accounts for the cost to partition the entire datasets, both map and trajectory, using the spatial index model. For instance, in the case of our Spark framework, a quad-index is constructed in the master node from a sample S of the input trajectory dataset; the data partitioning, on the other hand, is done for the entire datasets in the Spark cluster. Therefore, in this scenario C_{index} is the cost to build a quad-index with B spatial partitions from S in the master node, plus the cost to partition the map and trajectory datasets into B spatial partitions in the Spark cluster.

Furthermore, there is a post-processing step to merge records from trajectories that have been split across multiple partitions; this step also adds a cost C_{pos} to the final workload. Notice that the cost C_{pos} depends on how we handle boundary records, as well as the partitioning granularity. For instance, Figure 3.5 shows an example of space decomposition using Quadtree. If we decide to assign boundary crossing trajectories to all intersecting partitions (i.e. multiple assignments), then C_{pos} is simply the cost of choosing the best match from the result copies. If we decide, however, to split boundary trajectories into sub-trajectories according to their containing spatial partitions (i.e. single assignment), then C_{pos} is the cost of merging the resultant sub-paths at the end of the processing. For both strategies

C_{pos} also depends on the partitioning granularity, for instance, in Figure 3.5 increasing the partitioning granularity would either increase the number of replications for multiple assignments, or increase the number of splits for single assignment, thus increasing the post-processing cost. Overall, we can estimate C_{pos} on either the number of replicated trajectories on multiple assignment policy, or the number of sub-trajectories to merge in single assignment.

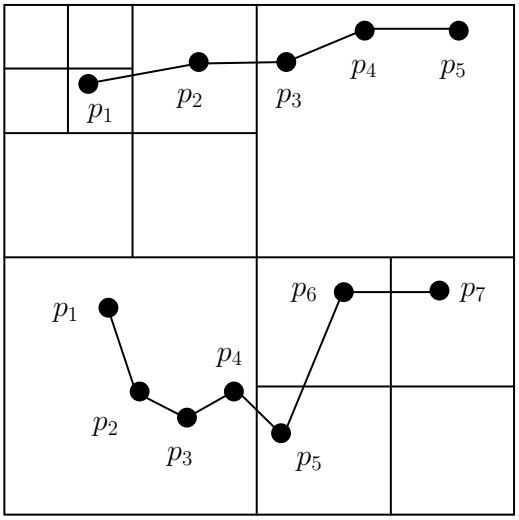


Figure 3.5: Example of Quadtree space partitioning for trajectories.

Load balancing and Communication cost: In Equation (3.1) we suppose B as a set of disjoint and homogeneous spatial partitions. However, real life spatial datasets are not uniformly distributed, for instance, the density of data records in a city center is much larger than in the suburbs. In distributed parallel applications, a poorly partitioned dataset can lead to contention, and increase communication and data transfer between the computing nodes. Therefore, we must make sure we employ a partitioning strategy that takes the data distribution into account for better load balancing. Although dynamic partitioning structures, such as Quadtree and k-d Tree, have a higher partitioning cost C_{index} compared to static structures such as Grids, this cost is small compared to the gains in load balancing (see Section 3.2.6).

Furthermore, global map-matching algorithms use a distance threshold to select candidate edges/nodes for matching. Therefore we must ensure that records within the candidates threshold are assigned to the correct partition. Given that a matching algorithm uses a candidate's distance threshold of size β, if the distance from a record to the partition boundary is smaller than β this will cause the matching algorithm miss some candidates in adjacent partitions. The simplest solution is to replicate points and edges within a distance β from the partitions boundary; however, for regions with high density of boundary records, replication will negatively affect the computation cost by increasing the number of data records (i.e. m and n in Equation (3.1)) (see Section 3.2.6); moreover, for in-memory storage frameworks, such as Spark, replication will also increase memory usage. In Equation (3.2) we estimate C_M^D the cost of distributed map-matching,

$$C_M^D = C_{index} + \frac{f((m+r_m)*(n+r_n))}{(B*U)} + C_{net} + C_{pos} \tag{3.2}$$

where r_m and r_n are respectively the total number of replication of trajectory points and road nodes, and C_{net} is the network cost of communications and data transfers between the nodes. In MR-based systems, such as Spark, C_{net} is basically the cost of the distributed shuffle operation to send data from *mappers* to *reducers* [39], and it is dependent of the network configurations, such as bandwidth, and both data locality and load balancing. For instance, in the MR model the slave nodes redistribute data based on the output keys (e.g. partition ID), such that all data belonging to one key (partition) is located on the same slave node, furthermore, MR always try to assign work to idle notes to best use the cluster resources, which means that a poorly distributed dataset would cause the slave nodes to shuffle more data, increasing networking cost. Therefore, for Spark map-matching C_{net} is highly dependent on the space partitioning strategy.

Bearing that in mind, our goal is to propose a framework for efficient and scalable map-matching in a distributed fashion, by reducing the cost C_M^D according to Equation (3.2).

3.2.5 Map-Matching Framework

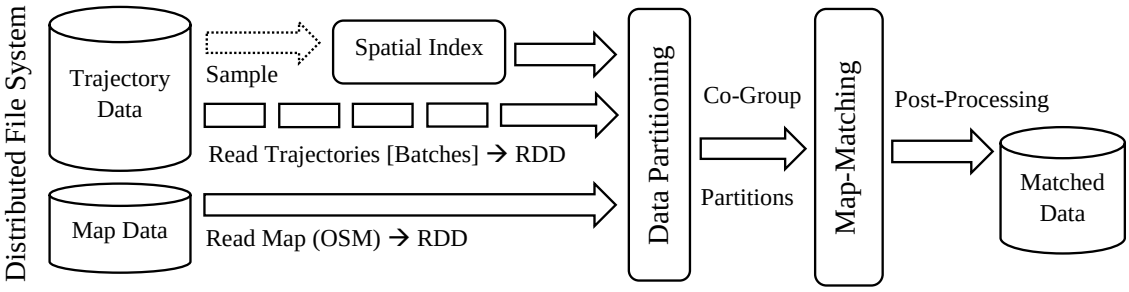


Figure 3.6: Spark map-matching framework overview.

Our goal is to reduce the overall cost of distributed map-matching according to Equation 3.2. In addition, we aim to reduce cluster memory consumption with Spark. The following steps summarize our proposed framework in Spark, as shown in Figure 3.6. In the next sections we provide a detailed description of our framework.

1. **Sampling-based index construction:** We select a small sample of the input trajectory dataset to build a quad-index in the master node. After that, the index is broadcast to the memory of all slave nodes.

2. **Data reading and partitioning:** We read both map and trajectory datasets as a Spark's RDD and assign every trajectory segment and map edge to its intersecting partition using the quad-index, and accounting for boundary objects.

3. **Co-grouping:** We co-group partitions containing both map edges and trajectory segment by spatial index into a single co-partition RDD.

4. **Map-matching computation:** We perform map-matching in each RDD co-partition in a parallel fashion using Spark.

5. **Post-processing:** A final post-processing step is performed to group the match results by trajectory key.

By building our quad-index from a sample of the input data we aim to reduce the C_{index} in Equation 3.2, since it accounts for the cost of building the spatial index and partitioning the dataset afterward, and also allow this model to be used in Spark. By co-grouping both map and trajectory data into balanced partitions, we aim to reduce the cost of map-matching processing by increasing parallelization and reducing the communication cost C_{net}. By wisely replicating boundary segments, we aim to reduce both the number of replications r_m and r_n, and the post-processing cost C_{pos}, without affecting accuracy.

Sample-based Space Decomposition

Since we need to provide Spark with a data partitioning model, before it can load and partition both map and trajectory data, we select a sample of the input trajectory dataset to build a quad-index in the master node. We employ a Quadtree space decomposition for both map and trajectory records, for it provides a fairly uniform partitioning of spatial records.

Firstly, we must estimate the best number of spatial partitions in the quad-model, this can be calculated by taking the maximum between the number of processing units available in the cluster, and the dataset size to load over the Spark's RDD block size, as in the following Equation (3.3):

$$N = \max\left(\frac{|T|+|M|}{|RDD_{Block}|}, U\right) \tag{3.3}$$

where N is number of partitions, $|T|$ and $|M|$ are the trajectory and map datasets size respectively in bytes, and $|RDD_{Block}|$ is the RDD data block size (64MB by default), and U is the number of processing units (e.g. CPU cores).

Given the number of partitions N, we build our Quadtree model using Spark as follows: We select a sample of the dataset S of size $|S|$ to build the quad-index I by the driver program (i.e. master node). We decide to split a partition N_i in the Quadtree when the number of records r_i in the partition is $r_i \geq 4*(|S|/N)$, this is to ensure that each partition will have roughly the same quantity of data record, since $|S|$ is the size of the input sample dataset, and N is the number of desired partitions, we want each partition to have $(|S|/N)$, since in quadtree partitioning a spatial partition is divided into 4 once it reaches a certain limit, we set this limit to $4*(|S|/N)$. After its construction, the index model I is broadcast to the memory of all slave nodes.

Data Partitioning

Given the quad-index model I, we partition the input datasets using MapReduce [181] on top of Spark's RDD, so that the number of data records in each partition is roughly uniform for load balancing. Furthermore, to allow parallel map-matching we must ensure that both map and trajectory records in a same spatial region are assigned to the same partition, so that there is no need to look for

matching paths in other partitions. Therefore, we load both map and trajectory datasets into Spark's RDD in main-memory, and assign every trajectory segment and map edge to its intersecting spatial partition using I. Finally we co-group all records mapped to the same spatial partition into the same data block to be processed in parallel, as shown in Figure 3.6.

Batch Processing: In order to reduce memory usage, we partition and match trajectories in batches. Once building the quad-index and partitioning the map data, the map-matching process is independent for every trajectory, that is, once a trajectory T_i is assigned to its intersecting partitions, we can match T_i independently of the remainder records. Therefore, to reduce cluster memory consumption, we split and load the trajectory dataset into RDD in smaller chunks. We load trajectories into our application in batches of roughly $(U * |RDD_{Block}| * \alpha)$ in size. Where U is the number of processing units, so that we use all available cores, and α is the Spark's RDD in-memory storage fraction (0.8 by default). In this scenario we assume the cluster memory can comfortably fit the map data and at least one trajectory data batch in-memory.

For the best of our knowledge, no other related work has applied batch processing for map-matching. Furthermore, as in Spark batch processing is not spatially-aware, the data batches are partitions of the input dataset read from disk or memory. Using the proposed framework, however, we can control the partitions load to memory in each batch, and make sure that each data batch (both map and trajectory) contains the data in a same spatial region, hence the distributed processing is fully decentralized, thus no other regions need to be loaded/searched for candidate matches.

Boundary objects and Replication: When assigning data records to partitions, however, we expect some trajectories and road segments to overlap with more than one partition, in this case we split the segments according with its intersecting partitions. However, during the map matching computation some boundary points can be mismatched, thus we replicate both road segments and trajectory segments within a certain distance threshold from the partition boundary in order to reduce both replication and uncertainty. Our empirical study performed in Section 3.2.6 demonstrated that boundary extensions β greater than 500m did not affect the map-matching accuracy, therefore we employ a $\beta = 500$m threshold in our spatial partitions.

In addition, the metadata about the whole trajectory (e.g. speed, length, sampling rate, etc.) is stored along with its sub-trajectories during the partitioning, to be used by global map matching algorithms when necessary.

Map-Matching Computation

Parallel Matching using MapReduce: Given a map-matching algorithm $M()$, each data partition N_i, containing both map and trajectory data, is sent to be processed in parallel in the Spark cluster using MapReduce with $M()$ as follows: For every sub-trajectory sub_j^T in N_i – where sub_j^T refers to the *j-th* sub-trajectory of a parent trajectory T – the $map()$ function outputs a $\langle key, value \rangle$ pair containing the parent-trajectory identifier T as *key*, and the path P_j^T that best matches sub_j^T with regards to $M()$ as

value, that is, the mapper outputs $\langle T, P_j^T \rangle$. We employ the a nearest-neighbor and HMM map-matching algorithm [101] in our framework; however, any map matching algorithm from the state-of-the-art can be used in our framework.

Post-Processing: The $reduce()$ function groups sub-paths by parent-trajectory key T (i.e. the post-processing C_{pos} step) and outputs the final results. The purpose of the post-processing step if to merge sub-paths of trajectories which might have been split and sent to separated spatial partitions due to their large extension. This post-processing step is not covered by related work which use spatial partitioning, they either assume the entire trajectory fits in the partition, or don't merge the sub-trajectories in the final step.

User Interface

Additionally, we provide an easy-to-use user interface with our framework, shown in Figure 3.7. Users are able to setup the Spark and partitioning parameters, load data, extract OSM map data from the Internet, as well as choose the matching technique to apply. The application was built using a component-based design, thus it allows easy plug in of additional map-matching algorithms inside the framework. The application is available to download in the project repository [4].

3.2.6 Experiments

We present a set of highlighted experiments on a real trajectory dataset to evaluate the performance, accuracy, and scalability of our approach.

Experimental Setup

We provide two different implementations to perform the experiment as follows:

1. Firstly we implemented our proposed batch-based framework, where after loading and partitioning the entire map data, we load and partition the input trajectory dataset in batches to reduce distributed memory consumption.

2. In the second implementation, we bulk load and co-partition the entire datasets (map and trajectory) into main-memory to speed up map-matching at the cost of cluster memory usage.

We use a 54GB trajectory dataset collected throughout China, and a 6GB OSM map from the Chinese road network. The trajectory dataset contains around 22 million heterogeneous trajectories from taxis and personal vehicles in a period of five days; while the OSM data contains around 1.5 million nodes. The data is initially stored in HDFS, we use the default MR block size of 64MB.

All algorithms are implemented in the Spark Java library version 2.0.1. Experiments are conducted on a cluster with 16 physical nodes (1 master and 15 slaves). Each node has eight cores and 64GB of

[4] https://github.com/douglasapeixoto/map-matching-framework

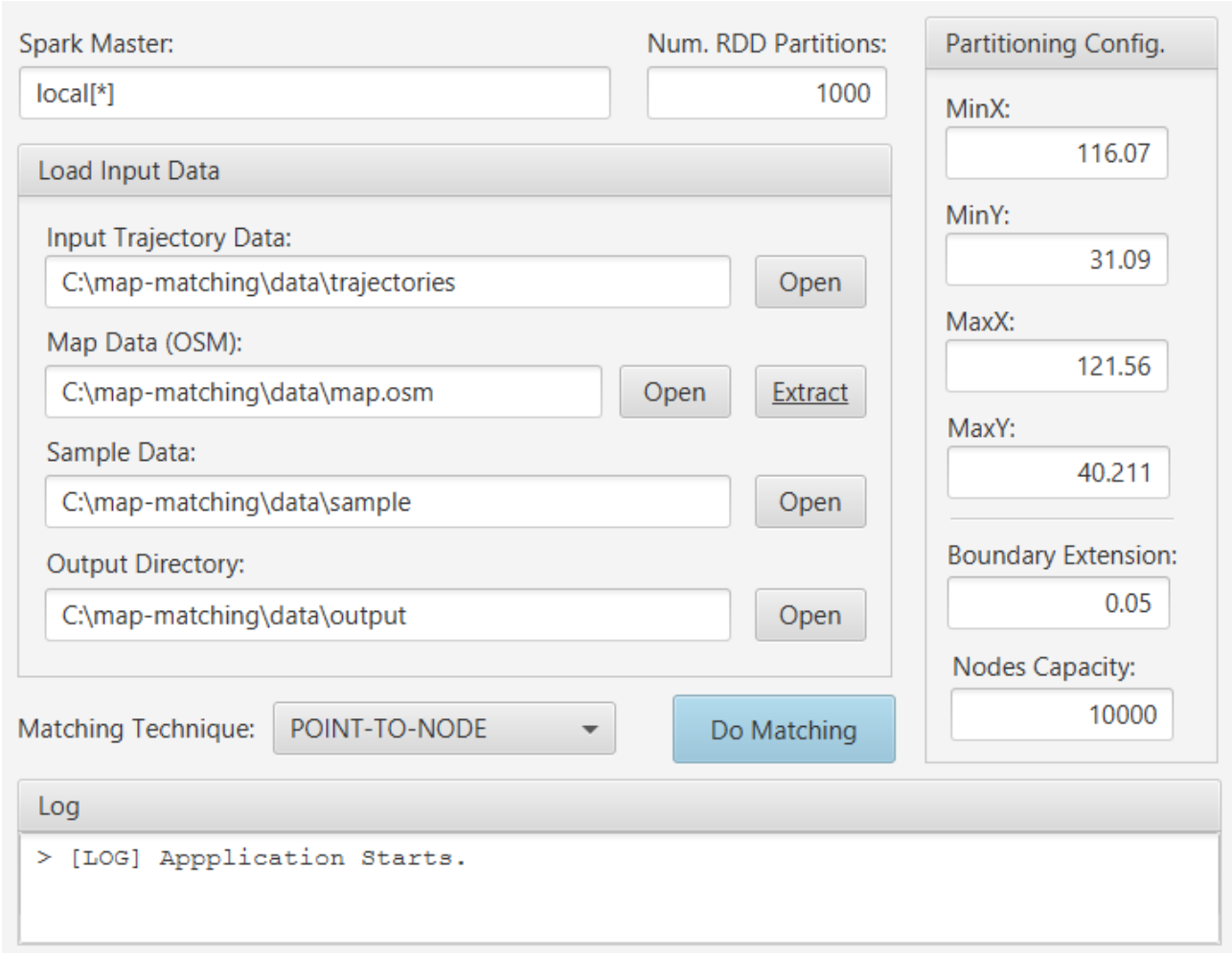


Figure 3.7: Application User Interface.

memory – in our experiments we configured Spark to use 7 cores and 60GB of memory in each slave node.

We evaluate our framework's performance and scalability by varying both the dataset size and the number of nodes in the cluster. We evaluate our work using three different adaptive spatial partitioning structures, i.e. Quatree, k-d Tree, and STR-Tree, and the HOM method proposed in [72] for MapReduce using grid partitioning. We adapt our framework for all aforementioned spatial structures using Spark. Adaptive spatial indexes were built using one million sample trajectories. Figure 3.8 illustrates the spatial structures used in our evaluation, including the spatial boundary extension β as described in Section 3.2.5. We evaluate our framework by employing the incremental nearest-neighbor matching algorithm, and the global HMM algorithm [101], which demonstrated better accuracy over sparse and noisy trajectory datasets.

Study of Parameters

In this section we study the effect of the number of partitions, as well as the boundary threshold size β, on the number of data replications, and in the overall performance. In this experiment we selected

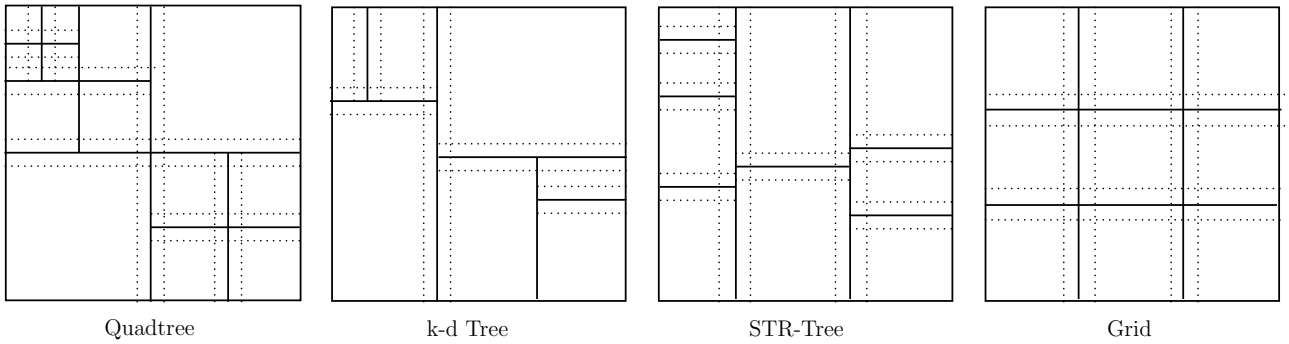


Figure 3.8: Spatial partitioning structures used in the comparative study, with their respective boundary extensions (dotted lines).

1,000 trajectories with high density of sample points (i.e. average sampling rate of one second, and a minimum of 100 points per trajectory), the total number of sample points is 320,000. Table 3.4 demonstrates the number of records and the map-matching accuracy as β grows. We fixed the number of spatial partitions N to 1,000 in this experiment. We evaluate the accuracy by comparing the results of our distributed framework with the original implementation of two map-matching algorithms, iterative and HMM, running in a single machine in a greedy manner (i.e. without parallelization or partitioning). From the results in Table 3.4 we use a boundary threshold of $\beta = 500$m in our framework.

Boundary (β)	# of Records	Iterative	HMM
0m	320,000	94.5%	86.9%
100m	326,785	97.4%	92.5%
200m	334,596	98.8%	96.3%
300m	335,598	99.7%	98.4%
500m	336,714	100.0%	100.0%

Table 3.4: Effect of the boundary extension threshold (in meters) on data replication and map-matching accuracy, with $N = 1,000$.

Boundary replication is necessary to reduce uncertainty, as demonstrated in Table 3.4, however, replication will also increase memory usage and the computation cost C_M^D, by increasing the number of data records to match, as given in Equation 3.2. Figure 3.9 show the overall running time of the framework varying the boundary threshold, this experiment was performed using the entire map and trajectory datasets. As expected, the overall running time is proportional to the number of records in the partitions, which increases with higher boundary values due to replication, and also increased the cost of the post-processing phase to find the best match in the duplicated records.

Similarly, the spatial partitioning granularity (i.e. number of spatial partitions N) will affect the number of replications, as shown in Table 3.5 (with boundary extension fixed in $\beta = 500$m). Even though a large number of partitions N can reduce the overall cost of a distributed map-matching C_M^D, as given in Equation 3.2, it will also increase the number of replications r_m and r_n. This is due to the

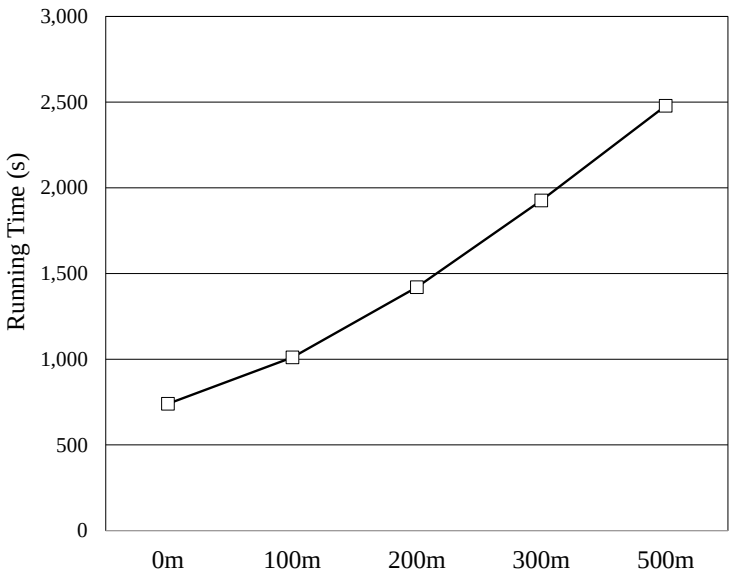


Figure 3.9: Running time varying the boundary extension threshold. Using 54GB trajectory data and 6GB OSM data.

increasing number of boundary point in partitions of small granularity. Therefore, in our framework we choose the number of spatial partitions N according to Equation 3.3 based on the Spark data block size and the available cluster resources.

# of Partitions (N)	# of Records
1,000	336,714
2,000	401,430
5,000	581,491
10,000	914,012

Table 3.5: Effect of the number of partitions on data replication. Partitions with a boundary threshold of 500m.

Performance and Scalability Study

Dataset size: Figure 3.10 compares the running time for each phase of the map-matching computation as we increase the dataset size. We use one to four times the input trajectory dataset to evaluate scalability. The partitioning phase accounts for the C_{index} cost of the quad-index model construction, and Spark data partitioning with all the dataset stored in-memory.

The balanced spatial structures (i.e. Quadtree, k-d Tree, and STR-Tree) had a better overall performance due to their more homogeneous distribution of the spatial data across the partitions, thus providing better load-balancing, which plays a key role in Spark performance. Uniform Grid, on the other hand, performed poorly executing over 17k seconds for 4x the dataset, this is mainly due to

high density of data in some spatial partitions as the dataset grows, this lead to contention in some processing units, as well as an increase in cluster communication and shuffle, i.e. data transfer cost C_{net}.

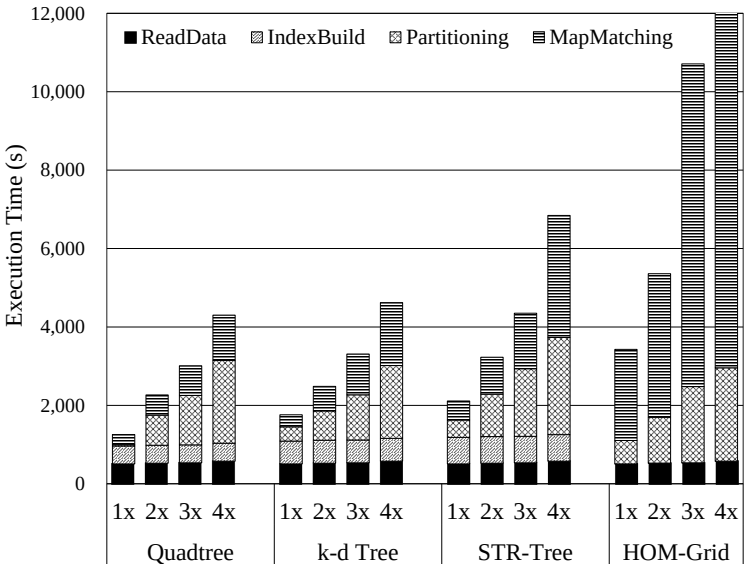


Figure 3.10: Spatial-aware map-matching execution time comparison (in seconds) on Spark, using multiple spatial partitioning methods as the dataset grows.

In Figure 3.10, *ReadData* accounts for the time to read both trajectory and map data from HDFS into Spark. We noticed that, even though the size of the map dataset is smaller than the trajectory, the time to read and parse the OSM data to our application was considerably higher using the Spark XML library, thus, the time taken to read additional copies of the input trajectory dataset was not significant to increase the overall read time.

The *IndexBuild* accounts for the time taken to read a sample of the trajectory dataset from the HDFS, and build the dynamic spatial index in the master node. Indexes were built with the same number of sample trajectories in all experiments. Even though a static grid can be constructed in basically zero time, our experiments demonstrated that using sample-based index construction was sufficient to improve performance compared to static Grid with no sampling.

The *Partitioning* accounts for the time taken to co-partition both map and trajectory data with regards to the spatial index. Boundary objects handling is also performed in this phase. As observed, the co-partitioning was the most demanding phase for balanced spatial structures, however, this cost was compensated by the gains in map-matching performance. The poor data distribution on static Grid, however, resulted in a much higher map-matching cost.

Finally, the *MapMatching* phase accounts for the execution time to process map-matching algorithm in the co-partitions, as well as perform the post-processing phase to merge resulting sub-paths by trajectory. Overall, our Quadtree based approach demonstrated better tread-off between the partitioning and the map-matching phases.

Batch processing: Figure 3.11 shows the average execution time of map-matching in batches as the

dataset grows. Based on our cluster configurations, i.e. $U = 105$ cores, $|RDD_{Block}| = 64$MB, we split our trajectory dataset into 10 batches of roughly 5.4GB. Each batch is loaded into our framework and processed by Spark in FIFO mode. We could optimize the process by overlapping the batches computation to a full use of the cluster resources, however, our goal with this approach is to reduce memory storage usage, therefore we do not overlap neither the batches loading nor the batches computation. Again the dynamic spatial models demonstrated better performance over uniform Grid partitioning. In this scenario, our Quadtree method outperformed the remainder methods showing a slight linear increase as the dataset grows.

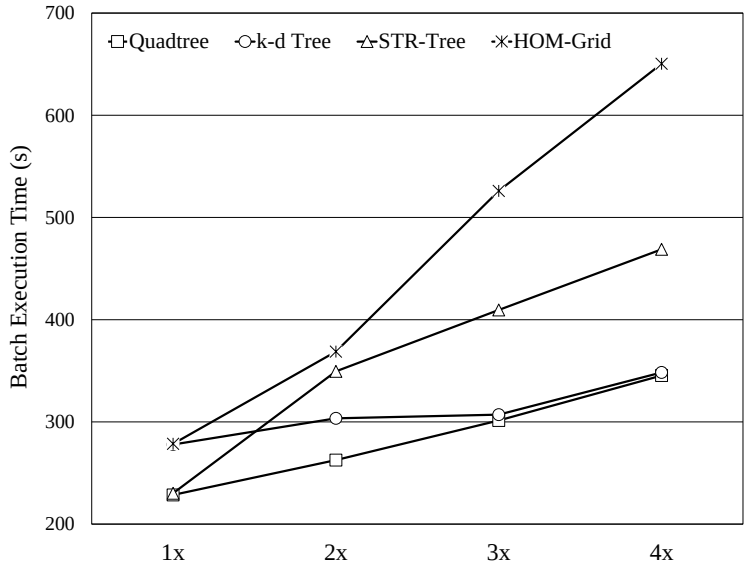


Figure 3.11: Average batch processing time comparison (in seconds) by multiple spatial partitioning methods as the dataset grows. Execution time accounts for the average time to read, partition, and process map-matching on each data batch using Spark.

Figure 3.12 compares the execution time between the two approaches to process the entire dataset. Although storing all the data in-memory had a better performance gain for most approaches – due to a better process and resources allocation by Spark – batch loading has a better trade-off between performance and memory usage.

Figure 3.13 shows the memory consumption using batch loading for each individual batch, against storing the entire trajectory dataset in-memory. As in the previous experiment, the trajectory dataset was split into 10 batches of roughly 5.4GB each. The batch approach has an overall gain of 5.2x in memory consumption against storing the entire dataset in memory; however, the number of batches can be adjusted to fit the available resources. One would expect a gain close to number of batches (i.e. 10x), however, this was not achieved mainly due to the map data being entirely stored in-memory in our framework.

Number of nodes: Similarly, Figures 3.14 and 3.15 depict the results when we vary the number of slaves nodes in the cluster. We compare the previous methods using one copy of the dataset. We use 5,

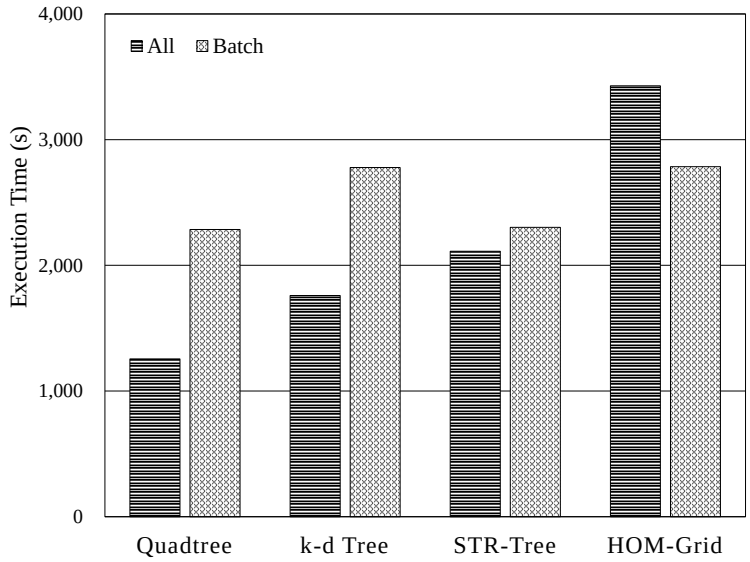


Figure 3.12: Batch loading vs. all dataset loading, performance comparison.

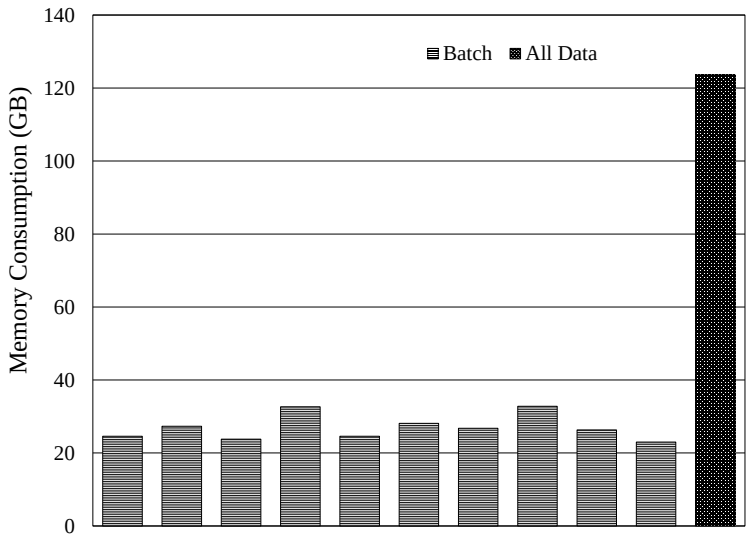


Figure 3.13: Memory consumption comparison (in GB), using batch loading (individual batches) and all-data loading.

10, and 15 slave nodes respectively to evaluate the effect of the number of nodes on each method's execution time.

As in the previous experiments, the balanced spatial structures performed better in all scenarios, with our Quadtree-based method performing better in all phases as show in Figure 3.14. Also for batch processing, Figure 3.15, our Quadtree method had the best performance gains and scalability, demonstrating near linear performance improvement as the number of nodes increases.

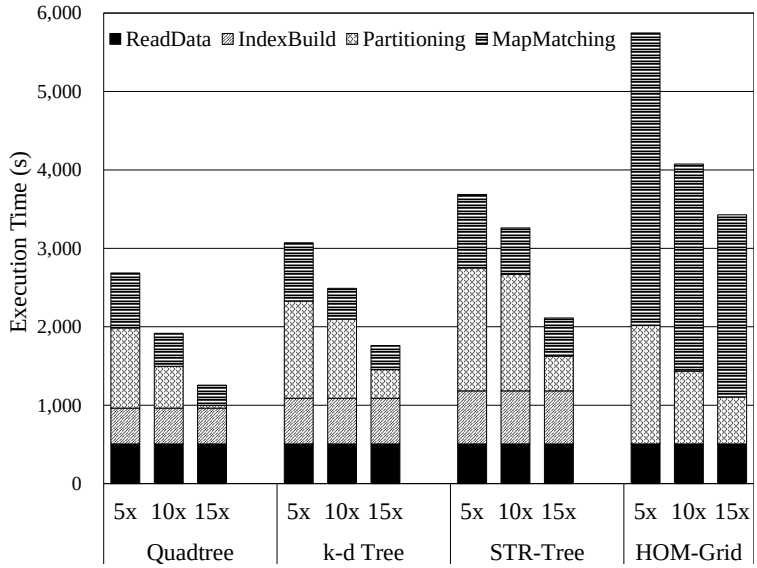


Figure 3.14: Spatial-aware map-matching execution time comparison (in seconds) on Spark, using multiple spatial partitioning methods by increasing the number of computing nodes.

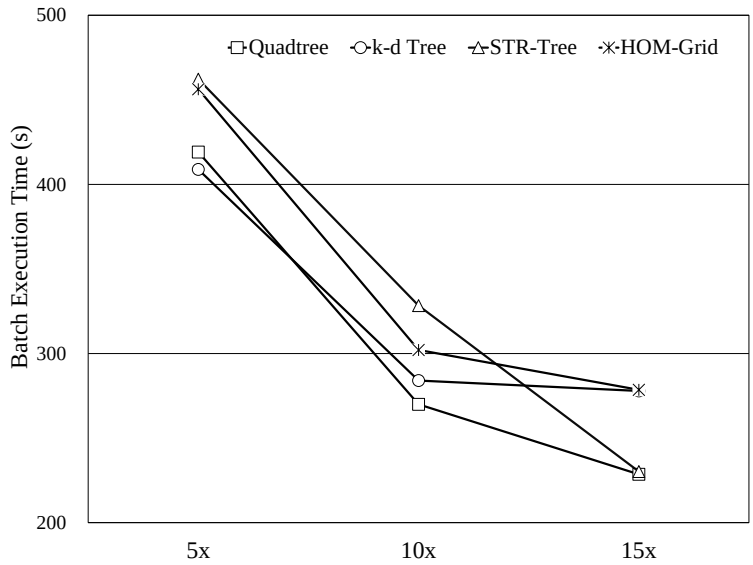


Figure 3.15: Average batch processing time comparison (in seconds) by number of nodes. Execution time accounts for the average time to read, partition, and process map-matching on each data batch using Spark.

3.2.7 Summary

Map-matching is an important pre-processing step to improve trajectory data quality and reduce uncertainty, due to inaccuracy of raw GPS data. The large amount of digital data available, however, has introduced a new problem of how to match massive amounts of both map and trajectory data in an efficient manner. In this chapter we introduced a Spark-based framework for the problem of large-scale offline map-matching. We introduced new features on top of Spark to allow efficient, scalable, and memory-wise processing of large-scale map-matching. First, we introduced a cost function for the distributed map-matching problem. Secondly, we use a sample-based quad-index construction, and

Quadtree co-partition of map and trajectory data to allow parallel and load-balanced map-matching. We build our partitions on top of Spark's RDD to achieve efficiency and scalability. We employ a safe boundary threshold, and wise split strategy to reduce replication. Finally we proposed a batch-based method for large-scale map-matching, using data loading and processing in smaller batches to reduce memory usage. A comparative study and experiments demonstrate that our framework achieved good efficiency and scalability on map-matching processing with lower memory consumption.

Chapter 4

Workload-aware and Memory-wise Trajectory Data Storage

4.1 Introduction

Trajectory data has become ubiquitous, and have been generated in unprecedented rates. Spatial-temporal trajectory data contains rich information about moving objects and phenomena; hence scientists, industry, and the community have been using trajectory data for a great number of applications, such as routes recommendation [97]; time prediction of public transportation [34]; finding points of interest (POI) in a given spatial region [32] [196]; identifying gathering patterns [194]; trip recommendation [197] [198]; transportation mode prediction [196]; drivers pastern analysis, city traffic planing, dynamic event identification, human interaction, and so on [195] [199]. In the mean time, in-memory distributed systems like Spark [181] have been used in several application domains to achieve efficiency and scalability, such as graph processing [169], relational database and SQL [14], data streams computation [182], and Spatial databases [168] [178] [175]. Spark [181] provides a robust distributed in-memory data structure, and can fill the gap between performance, scalability, and fault-tolerance, as well as efficient resources allocation of parallel and concurrent jobs.

Database systems dedicate their efforts towards reliable and efficient data storage and fast query performance. The complexity of the query highly depends on the type of data in the dataset. Spatial-temporal selection is the most fundamental query operation in trajectory databases, thus has received plenty of attention, e.g. [35] [158] [172]. Spatial-temporal selection queries are useful to select a small sample of a big dataset for a given time interval and spatial predicate, for example:

"Retrieve all trajectories in the city centre of Brisbane, between 09:00AM and 18:00PM of the current date.".

Furthermore, with the increasing demand for low-latency services over large-scale data, a trajectory database system should be able to serve multiple requests over large-scale datasets, providing good scalability, high throughput, and fast query response. A common approach for achieving high throughput and efficient query processing over large datasets is by means of in-memory data storage

on distributed environments, where large datasets are partitioned and distributed across the memory of a cluster of computers. In spatial and spatial-temporal databases this is done using dynamic and adaptative data structures, such as Quadtree, k-d tree, and Rtree for load-balancing. However, storage and processing of spatial-temporal trajectory data using Spark is challenging, since Spark is not equipped for supporting sequential and spatial-temporal data in its core. Furthermore, data partitioning and storage in Spark is done using a read-only static data structure (i.e. RDD); therefore, adaptative partitioning is not directly supported. Moreover, since Spark is a in-memory-based framework, data storage and query processing on top of Spark should be memory-wise, for instance, the trajectory datasets may be too large to comfortably fit in the cluster memory. A solution to reduce memory consumption is to react to changes in the query workload efficiently, since some spatial regions, such as urban areas, and time intervals, such as working hours, receive more query requests (hotspots), thus data records in such regions and intervals should receive priority for in-memory storage over least requested data, which can be stored on disk to safe memory space. However, even though Spark possesses both in-memory and on-disk storage, the exchange of data from memory to disk is not based on the query workload, but in the memory availability. however. Optimizing load-balancing and memory usage are essential to a good Spark application.

Existing works for spatial data using Spark and MapReduce [6] [48] [103] [168] [12] employ balanced partitioning structures, such as Quadtree, k-d Tree, and STR-Tree, to provide workload balancing, and prune the search space at query time. However, current distributed systems for spatial and spatial-temporal data are unable to provide all the mentioned features. For instance, AQWA [12] [11], MD-HBase [103], SpatialHadoop [48], ScalaGiST [94], and HadoopGIS [6] are disk-based systems and do not provide support trajectory data. Simba [168], GeoSpark [175], SparkGIS [16], and SpatialSpark [174] provide an in-memory-based system for spatial data on top of Spark; however, they do not provide support for trajectory data storage and query. CloST [146], TRUSTER [172] and PRADASE [98] provides support for trajectory data storage and query using spatial partitioning (i.e. grid and quadtree), however they are disk-based systems thus do not address memory usage and storage. OceanST [178] provides a distributed in-memory storage for trajectories on top of Spark, however it does not consider the query workload (i.e. query hotspots), and assumes the entire data fits in the cluster memory. CloST [146] is a Hadoop-based spatial-temporal storage system. CloST proposes a new data model and file format to store trajectory data in HDFS. CloST uses a three-level hierarchical partitioning in MR, where in the first level trajectories are grouped into coarse *buckets* according to the moving objects OID; in the second level each *bucket* is partitioned into spatial *regions* using Quadtree; in the third level each *region* is divided into fine-grained 1-D *blocks* of time. The goal of CloST is to support efficient *single-object queries* (i.e. spatial-temporal selection) and *all-object queries* (i.e. selection by object OID), however, CloST is a disk-based storage system, and does not account for memory usage and query workload.

In this chapter we introduce an architecture for large-scale trajectory storage and querying, and formalize the requirements for effective in-memory trajectory data management using Spark as two problems: *adaptative data partitioning* for spatial-temporal trajectories, and *memory-wise storage*

based on the query workload, and present solutions for them.

We approach the problem of adaptative data partitioning on Spark by combining a cost driven approach with sampling-based partitioning. We employ a hierarchical Quadtree-based partitioning [146], since it provides a fairly uniform partitioning of spatial-temporal trajectory records. Since building a dynamic spatial index model from a large dataset can be cumbersome, and a data partitioning model must be provided to Spark beforehand, we address this limitation by providing a sampling-based quad-index construction using a cost model, and finally employ the sampling-based model to Spark.

Our solution for memory-wise storage uses a reactive technique, to organize the data partitions into different storage levels (i.e. memory and disk) based on the query-workload. Firstly, we extend the adaptative Quadtree index to store the status of the data partitions, so that we can efficiently identify query hotspots. Then, we introduce the *Active-Time Window* mechanism, which reacts to changes in the query-workload, and controls the storage level of each data partition in real-time, such that only RDD partitions containing query hotspots are kept in memory, thus reducing memory usage. We apply this new feature to the core of the Spark's RDD so that data exchange between memory and disk in Spark is now based on the query workload.

Least-Recently-Used LRU: In Spark, when memory space is not sufficient for RDD caching, data partitions will be evicted, if these partitions are used again further, they will be reproduced by the Lineage information and cached in memory again. Cached datasets that do not fit in memory are either spilled to disk or recomputed on the fly when needed, as determined by the RDD's storage level.

Spark's gives in-memory storage priority for the latest data partitions used in case the memory available is not enough to store all data partitions (LRU least-recently-used). Although our approach aims to achieve the same goal, Spark's LRU is not spatial-temporal aware, which means when partitions are build for storage in Spark they are not organised based on their spatial-temporal proximity, which means data from a same spatial region may be sent to different physical partitions. Hence, when a query request data for execution, the number of partitions loaded from disk may be too large, with several false positives, since if a partition contains only one record necessary for the query execution, the entire partition will be loaded. In our approach, on the other hand, we make sure the spatial-temporal relationship between the data is known and kept during the partitioning, so that the data in a same spatial-temporal region is put together in a same physical partition. Therefore, when a query requests data for execution, the number of data records loaded will be smaller than in the former approach.

In sum, this chapter makes the following contributions.

- Section 4.2: Firstly we formally describe the background of our work in terms of spatial-temporal trajectory data storage and query for Spark, then we introduce the requirements for an effective architecture for in-memory large-scale trajectory data management.

- Section 4.3: We define the problem of large-scale trajectory data management in terms of data partitioning, storage, and retrieval. We provide an estimative of the distributed spatial-temporal

search query cost, in order to optimize data retrieval, in-memory storage, and sampling-based partitioning.

- Section 4.4: Based on the requirements and the cost model, we introduce an workload-aware architecture for trajectory data management on top of Spark, to support multi-user queries and analytics over large-scale trajectory data with memory-wise utilization. Based on the architecture we developed a system that can react to changes in the query workload in order to reduce resources utilization. We exploit the in-memory nature and distributed parallel properties of Spark for scalable and low-latency trajectory data storage and processing. Finally, we provide an efficient data retrieval module for concurrent queries; we focus on spatial-temporal data retrieval, since it is the most fundamental operation in trajectory databases.
- Section 4.5: We evaluate our Spark architecture using real-world trajectory datasets with billions of records, showing that our technique scales for large datasets, and it is memory efficient. We achieved similar performance and throughput in data retrieval, yet with up to 3.5x gains in memory consumption compared with the related work.

4.2 In-memory Large-scale Trajectory Data Management

In this work we focus on large-scale trajectory data storage and query for multi-user applications.

Data partitioning and data storage play a key role in any distributed database application. We must account for load-balancing when partitioning a dataset for distributed parallel storage and computation. In spatial database, this is achieved by using spatial data structures to partition the data space, such as R-tree, Quadtree, and kd-Tree, for instance. For spatial-temporal trajectories, we must also account for sequentiality and the temporal dimension.

Given a large trajectory dataset $\mathbb{T}$, a spatial partitioning model M, and a query workload composed of a set of input queries (i.e. data access requests) $\mathbb{Q}$, we want to partition $\mathbb{T}$ using M, and w.r.t. $\mathbb{Q}$, such that both the process time of every query request $Q_i \in \mathbb{Q}$ and the memory store cost of $\mathbb{T}$ are minimized, that is, we want to find a trade-off between throughput and memory consumption.

To solve this problem, first we must choose the best partitioning model M to partition the data space of $\mathbb{T}$ taking into account load balancing, then choose the best memory storage strategy w.r.t. the query workload $\mathbb{T}$.

In this work we focus on large-scale trajectory data storage and spatial-temporal search for multi-user applications, that is, for a batch of k user queries $\mathbb{Q} = [Q_1, Q_2, ..., Q_k]$, where $Q_i = (R_i, t_1^i, t_0^i)$, we want to find $ST(\mathbb{S}, R_i, t_0^i, t_1^i)$ for every query $Q_i \in \mathbb{Q}$. We address the problem spatial-temporal selection over large-scale trajectory data on top of Spark, in order to achieve low query latency, high throughput, and scalability with fault-tolerance.

In summary, these are the main requirements for an effective architecture for large-scale trajectory data management using Spark.

- Efficient, reliable, and scalable storage of spatial-temporal trajectory data, with fault-tolerance.

- Multi-user query processing environment, with high throughput and low latency queries.
- Resource-wise utilization, in order to reduce memory consumption without affecting the system's performance.

4.3 Search Query Workload Estimative

We provide an estimative of the workload for a single spatial-temporal (ST) search query, in a distributed parallel fashion using space partitioning based on the following observations:

4.3.1 Distributed ST-Search Cost

Given a list of user input queries $\mathbb{Q} = [Q_1, Q_2, ..., Q_k]$, where $Q_i = (R_i, t_1^i, t_0^i)$, the cost C_{st}^i of a single search query Q_i for a given trajectory dataset $\mathbb{S}$, spatial query region R_i, and time interval $[t_0^i, t_1^i]$, can be estimated on the total number of trajectory segments n in $\mathbb{S}$, i.e. $C_{st}^i = O(n)$, since we simply need to check for segments intersecting R_i during $[t_0^i, t_1^i]$.

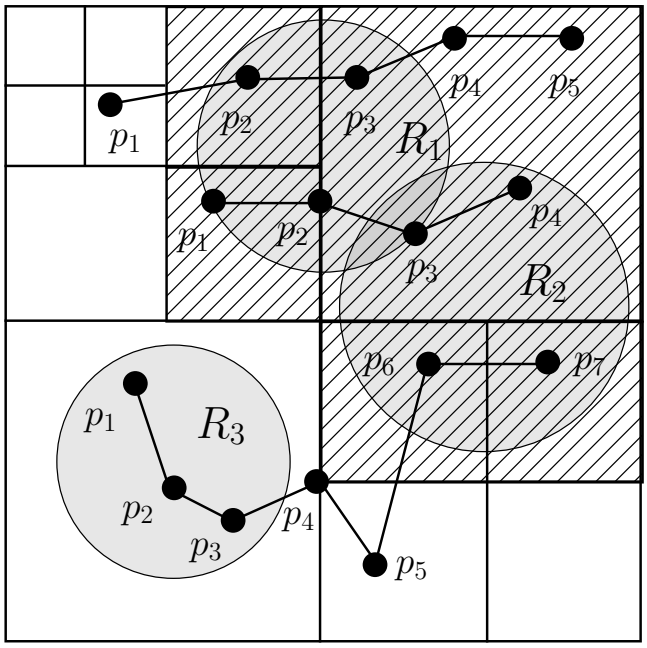


Figure 4.1: Example of Quadtree space partitioning for trajectories.

Since ST-Search is intrinsically parallelizable using space decomposition, we can greatly decrease the computational cost C_{st}^i by partitioning the input dataset using some space partitioning method, then select only the partitions containing candidate trajectories, that is, the partitions intersecting the query region R_i during $[t_0^i, t_1^i]$. For instance, Figure 4.1 shows an example of space decomposition using Quadtree with three query regions R_1, R_2, and R_3. To process R_1 we just need to consider data in the three spatial partitions the query region intersects with. Finally, we perform a precise search in each candidate partition in parallel. Equation (4.1) depicts the estimate cost for spatial-aware ST-Selection

query Cp^i_{st} in parallel with space partitioning,

$$Cp^i_{st} = C_{io} + \frac{n}{\arg\min(B,U)} + C_{pos} \tag{4.1}$$

where n is the number of data records in the candidate partitions, B is the number of candidate partitions/blocks, and U is the number of processing units available to process the query i (supposing all units with same computational power).

C_{io} is the I/O cost to load the candidate partitions from the file system, and it is relative to the total number of records to read in the candidate partitions. Notice that, if the data partitions are stored in main-memory, then $C_{io} = 0$.

Furthermore, there is a post-processing step to merge segments from trajectories that have been split across multiple partitions; this step also adds a cost C_{pos} to the final workload. Notice that the cost C_{pos} depends on how we handle boundary records, as well as the partitioning granularity. For instance, in Figure 4.1 notice that the query region R_1 intersects trajectories in multiple spatial partitions. If we decide to assign boundary crossing trajectories to all intersecting partitions (i.e. multiple assignments), then C_{pos} is simply the cost of removing duplicated results. If, however, we decide to split boundary trajectories into sub-trajectories according to their containing spatial partitions (i.e. single assignment), then C_{pos} is the cost of merging the sub-trajectories at the end of the processing. If we decide, however, to split boundary trajectories into sub-trajectories according to their containing spatial partitions (i.e. single assignment), then C_{pos} is the cost of merging the resultant sub-paths at the end of the processing. For both strategies C_{pos} also depends on the partitioning granularity, for instance, in Figure 4.1 increasing the partitioning granularity would either increase the number of replications for multiple assignments, or increase the number of splits for single assignment, thus increasing the post-processing cost. Overall, we can estimate C_{pos} on either the number of replicated trajectories on multiple assignment policy, or the number of sub-trajectories to merge in single assignment.

In Equation (4.1) we suppose B as a set of disjoint and homogeneous spatial partitions. However, real life trajectory and spatial datasets are not uniformly distributed, for instance, the density of data records in a city center is much larger than in the suburbs. In distributed parallel applications, a poorly partitioned dataset can lead to contention, and increase communication and data transfer between the computing nodes. Furthermore, a unbalanced partitioning will increase the number of False Positives (FP), that is, records in the candidate partitions that are not part of the query result, hence increasing C_io and the overall cost. Therefore, we must make sure we employ a partitioning strategy that takes the data distribution into account for better load balancing.

In addition, when partitioning the data space, we must account for boundary objects, once both trajectories can intersect with more than one spatial partition. The simplest solution is to replicate boundary segments, however, for regions with high density of boundary records, replication will negatively affect the computation cost by increasing the number of data records; moreover, for in-memory storage frameworks, such as Spark, replication will also increase memory usage. In Equation (4.2) we estimate Cd^i_{st} the cost of distributed ST-Selection query,

$$Cd^i_{st} = C_{io} + \frac{(n + r_n)}{\arg\min(B,U)} + C_{pos} + C_{net} \tag{4.2}$$

where r_n is the total number of replication of trajectory segments, and C_{net} is the network cost of communications and data transfers between the nodes. In MR-based systems, such as Spark, C_{net} is basically the cost of the distributed shuffle operation to send data from *mappers* to *reducers* [39], and it is dependent of the network configurations, such as bandwidth, and both data locality and load balancing. For instance, in the MR model the slave nodes redistribute data based on the output keys (e.g. partition ID), such that all data belonging to one key (partition) is located on the same slave node, furthermore, MR always try to assign work to idle notes to best use the cluster resources, which means that a poorly distributed dataset would cause the slave nodes to shuffle more data, increasing networking cost. Therefore, for Spark ST-Selection query C_{net} is highly dependent on the space partitioning strategy.

4.3.2 Spatial Partitioning and Indexing Cost

Although partitioning the data space can reduce the query workload cost, by pruning the search space and allowing parallelization, partition the entire dataset adds a cost C_{index} to the system. C_{index} is the cost of building the spatial index model, and partitioning the input dataset after data loading. The cost of building the spatial index depends on both the index strategy employed (e.g. balanced or static index), and the number of data records used to build the index model. The data partitioning accounts for the cost to partition the entire dataset using the spatial index model. For instance, in the case of our Spark system, a quad-index is constructed in the master node from a sample P of the input trajectory dataset; the data partitioning, on the other hand, is done for the entire datasets in the Spark cluster. Therefore, in this scenario C_{index} is the cost to build a quad-index with B spatial partitions from P in the master node, plus the cost to partition the trajectory dataset into B spatial partitions in the Spark cluster. Although dynamic partitioning structures, such as Quadtree and k-d Tree, have a higher partitioning cost C_{index} compared to static structures such as Grids, this cost is small compared to the gains in load balancing.

Bearing that in mind, our goal is to propose a storage architecture for efficient and scalable query search in a distributed fashion, by reducing the individual cost Cd^i_{st} according to Equation (4.2). Ideally, for a list of k user input queries, we expect:

$$C_{index} + \sum_{i=1}^{k} Cd^i_{st} \ll \sum_{i=1}^{k} C^i_{st}$$

by optimizing the cost Cd^i_{st} of each individual query q_i. In addition, we want to reduce the system's main memory storage consumption.

4.4 Storage System Architecture

We propose a Spark-based architecture for scalable and memory-wise storage of GPS trajectory data, and low-latency query processing with fault-tolerance and high throughput. An overview of our architecture with its main components is given in Figure 4.2. Together, the *Data Manager* and

Distributed Storage System compose the Data Layer of our architecture. While the *Query Manager* and *Task Scheduler* compose the Query Layer of our architecture. Both layers are build on top the Spark framework. We describe the components in the next sub-sections.

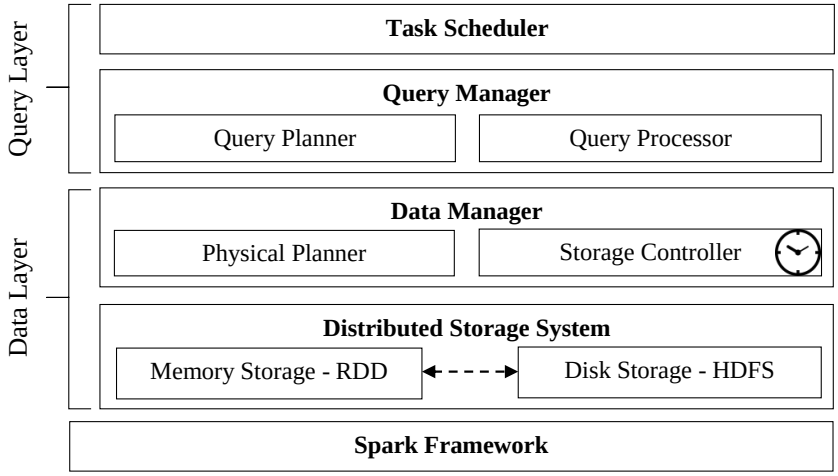


Figure 4.2: System architecture overview.

4.4.1 Data Manager

Together, the *Data Manager* and the *Distributed Storage System* are responsible for data storage and management. The *Data Manager* is a independent component responsible for the partitioning, indexing, organization, and retrieval of trajectory data. In another words, its main job is to manage the data partitions in the distributed file system. The *Data Manager* is composed by two main sub-components, the *Physical Planner* and the *Storage controller*.

Physical Planner

The *Physical Planner* is responsible for raw trajectory data loading, partitioning, index construction, and physical planning in the distributed file system. An overview of the physical planner is given in Figure 4.3.

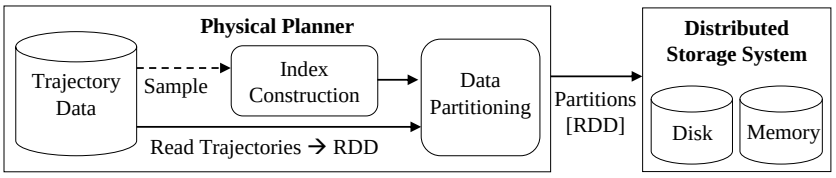


Figure 4.3: Physical planner overview.

Our architecture is build on top of Spark's RDD and HDFS, which both provide a reliable and fault-tolerant distributed storage for large datasets. Since building a dynamic spatial index model from a large dataset can be cumbersome, the physical planner employs a sampling-based hierarchical spatial-temporal index construction, using a extended version of the quad-tree model CloST [146]

for load balancing and storage level control. Once the hierarchical model is built, we broadcast the model to all nodes. Finally, we read the entire trajectory dataset and employ the model to Spark for data partitioning.

Since a data partitioning model must be provided to Spark before it can load and partition the trajectory data, the Physical Planner selects a sample of the input dataset to build a spatial-temporal quad-index in the master node.

By building our quad-index from a sample of the input data we aim to reduce the C_{index} in Equation 4.2, since it accounts for the cost of building the spatial-temporal index and partitioning the dataset afterward, and also allow this model to be used in Spark. By grouping trajectory data into balanced partitions, we aim to reduce the cost of query processing by increasing parallelization and reducing the communication cost C_{net}.

Firstly, we must estimate the best number of spatial-temporal partitions, this can be calculated by taking the maximum between the number of processing units available in the cluster, and the dataset size to load over the Spark's RDD block size, as in the following Equation (4.3):

$$N = \max\left(\frac{|T|}{|RDD_{Block}|}, U\right) \tag{4.3}$$

where N is number of spatial-temporal partitions, $|T|$ is the trajectory dataset size in bytes, and $|RDD_{Block}|$ is the RDD data block size (64MB by default), and U is the number of processing units (e.g. CPU cores). However, in the CloST partition, each spatial partition is divided into time slices of size ϑ, therefore, we can calculate the number of spatial partitions (i.e. quadtree partitions) using Equation (4.4):

$$B = \left(\frac{N}{\theta/\vartheta}\right) \tag{4.4}$$

where θ the total time extent of the trajectories in the dataset, and ϑ is the time slice size.

Given the number of spatial partitions B, we build our Quadtree model using Spark as follows: We select a sample of the dataset S of size $|S|$ to build the quad-index I by the driver program (i.e. master node). We decide to split a partition N_i in the Quadtree when the number of records r_i in the partition is $r_i \geq 4 * (|S|/N)$, this is to ensure that each partition will have roughly the same quantity of data record, since $|S|$ is the size of the input sample dataset, and N is the number of desired spatial-temporal partitions, we want each partition to have $(|S|/N)$, since in quadtree partitioning a spatial partition is divided into 4 once it reaches a certain limit, we set this limit to $4 * (|S|/N)$. An additional variable is created to keep the current storage level of each partition (i.e. in-memory or on-disk). After its construction, the index model I is broadcast to the memory of all slave nodes.

The Physical Planner does not assume any prior knowledge of the query-workload, and therefore has two starting modes: namely *Hot* and *Cold*. The former will consider all partitions as *Hot*, and store all data partitions in memory when the application starts; while the latter will initially consider all partitions as *Cold*, thus starting with all partitions on-disk. In the *Hot* mode we consider the entire dataset fits in the main-memory.

Storage Controller

The *Storage controller* is a workload-aware caching component, responsible to manage the storage level of the data partitions in the distributed file system, that is, it controls which data partitions are stored in main-memory (RDD) and which partitions are stored on disk (HDFS). Data partitions are exchanged between memory and disk on request of the *Storage Controller* based on the query workload.

Our system architecture can react to changes in the query workload in order to reduce memory usage. Some spatial regions and time intervals, such as urban areas during working hours, are more likely to be accessed frequently, namely *query hotspots*, we introduce the *Active-Time Window* mechanism inside the *Storage Controller* to adapt the data storage based on the query workload. We do so by adding one more level in the CloST index, to keep information about query requests on each partition, such as the number of queries performed, and the last time the partition was requested ("active-time"), as well as the partition status, i.e. Hot or Cold. Based on that information, after a given time interval φ if a partition is not requested, then it becomes *"cold"*, and is therefore stored in the distributed file system on disk (i.e. HDFS) on request of the *Storage Controller*. On the other hand, if a *cold* partition is requested, the *Storage Controller* load it back to main-memory, and its status is updated to *"hot"*. Loading *cold* partitions from disk to main-memory increases the system's I/O cost, however, our goal is to achieve a better tread-off between query performance and caching. Furthermore, the threshold value φ can be adjusted to fit the system needs and the resources available in the cluster.

4.4.2 Query Manager

Together, the *Task Scheduler* and the *Query Manager* are responsible for user's requests handling, query processing, and concurrency control. The *Data Manager* is composed by two main sub-components, the *Query Planner* and the *Query Processor*. Figure 4.4 shows the architecture's query workflow.

Briefly, queries are performed in a *filter-and-refinement* fashion using a MapReduce algorithm. First we prune the search space by filtering spatial-temporal partitions containing candidate trajectory segments, according to the queries parameters in the *Query Planner*. If any candidate partition is not in-memory, the *Storage Controller* signals the *Distributed Storage System* to load the partitions from disk into main-memory. The status of each partition selected in the filter step is updated at this phase by the *Storage Controller*. Finally, a precise check is done by the *Query Processor*, and the query results are returned to the users. We describe the query workflow, and main query components in the following sections.

Query Planner

The *Query Planner* uses the knowledge of the logical data storage plan to identify candidate partitions based on the M requests from the *Task Scheduler*. In another words, the *Query Planner* is responsible for the *filter* step of the queries processing. It identifies the partitions intersecting with the queries

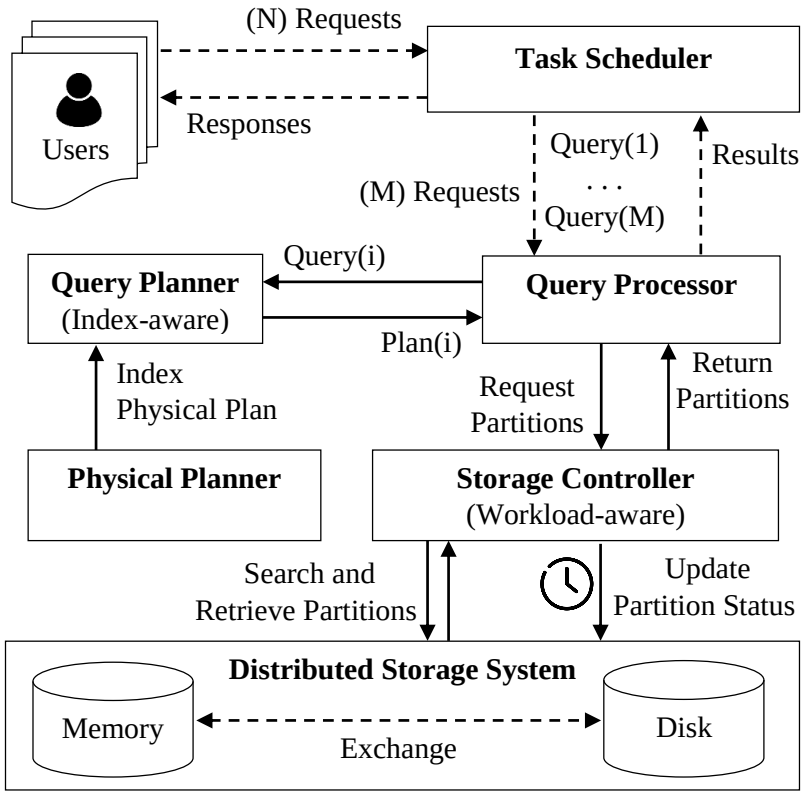


Figure 4.4: Query workflow.

using the extended CloST index from the *Physical Planner*, and requests those partitions from the *Data Manager*.

For instance, taken the example of Figure 4.1, suppose the *Task Scheduler* receives three requests on the query regions R_1, R_2, and R_3, that is $N = 3$, and submits the first two queries in R_1 and R_2 to the *Query Manager*, that is $M = 2$; to process R_1 the *Query Planner* only considers data in the three spatial partitions the query region intersects with. If any requested partition is Cold, then they are loaded into main-memory by the *Storage Controller*, and their status is updated.

Query Processor

The *Query Processor* performs the final stage of the queries processing and post-processing. In another words, the *Query Processor* is responsible for the *refinement* step on the candidate partitions from the *Data Manager*, by running a precise check in each partition to collect segments/sub-trajectories satisfying each of the M queries. Finally, a post-processing step is performed to merge segments/sub-trajectories by trajectory ID.

4.4.3 Task Scheduler

The *Task Scheduler*, illustrated in Figure 4.5, is responsible for receiving requests and scheduling the execution of user queries. The *Task Scheduler* determines how to move user requests between the Job and Ready queues, and then to the application for execution.

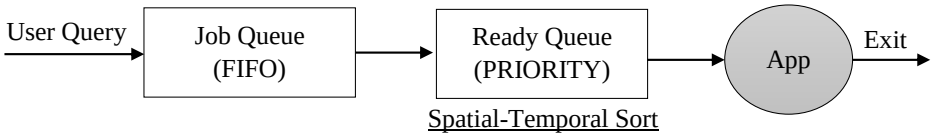


Figure 4.5: Task Scheduler Overview.

The *Task Scheduler* maintains two separated queues, the *Job Queue* and the *Ready Queue*. In summary, the *Job Queue* keeps all the user requests in the application. The *Ready Queue* keeps a priority queue with the user jobs ready and waiting to execute.

Job Queue: The *Job Queue* receives and queues a number of N user requests as soon as they arrive in the application. Upon request of the *Task Scheduler* it sends up to M jobs waiting in the head of the queue in FIFO order the *Ready Queue* for execution.

The goal of the *Job Queue* is to ensure that jobs are sent to the *Ready Queue* as they arrive, and thus avoid user queries to wait indefinitely for execution. The order in which the query jobs are sent for execution in the cluster, however, is determined in the *Ready Queue*.

Ready Queue: The *Ready Queue* receives up to M queries from the *Job Queue* and constructs a spatial-temporal-aware priority queue for execution. It gives priority to queries which intersect with one another in time and space. This is to ensure that queries requesting data from the same spatial-temporal partitions are executed first, because the larger the number of queries accessing a same spatial-temporal region the more likely they are to be accessing a query hotspot, hence it is more likely that the required partitions for execution of those queries are already in memory; therefore those queries executions are more likely to finish sooner since the data is in-memory, this will release computation resources sooner for the next queries. On the contrary, if queries requesting data from cold partitions were to be sent first, they would cause a longer wait in the process queue, since data from cold partitions need to be loaded from disk, thus holding computational resources for longer and delaying the remainder queries in the queue. Therefore, we ensure that queries which are more likely to finish first, i.e. accessing query hotspots, are given priority for execution.

The degree d of priority in the queue is calculated based on the number of intersections, for instance, in Figure 4.6 seven queries Q_1 to Q_7 are submitted to the application and queued in *Job Queue* as they arrive; if $M = 5$ then the first five queries in the queue are moved to the *Ready Queue* upon request. Now suppose the queries Q_1, Q_3, and Q_5 intersect, that is $d_{\{1,3,5\}} = 3$, and that Q_2 and Q_2 intersect, that is $d_{\{2,4\}} = 2$, hence the first set of queries are given higher priority for execution over the second. For instance, in Figure 4.1, if $M = 3$ then the queries R_1 and R_2 are given priority of execution over R_3, since they intersect.

No more than M queries can be running at same time. The number of queries M submitted by the *Task Scheduler* can be adjusted, and depends on the cluster capabilities (i.e. resources availability and computational power). After submission, the M queries are executed in a "round-robin" fashion using

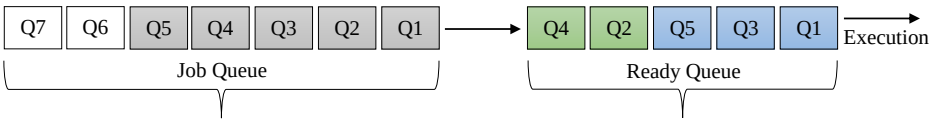


Figure 4.6: Query Scheduling Example, for $M = 5$.

Spark's *FAIR* job scheduling, so that all queries get a roughly equal share of the cluster resources, and also to avoid a query with low priority to wait indefinitely in to complete execution. When one query execution is completed, other from the head of the *Job Queue* is moved to the *Ready Queue*.

4.4.4 User Interface

Figure 4.7 shows the system's user interface prototype. System administrators are able to setup the cluster configurations, as well as the parameters for data partitioning, physical planning, storage controlling, and the number of concurrent tasks supported by the cluster. Users are able to select a spatial region (e.g. rectangle, circle, polygon) using the system's map interface, and choose the query's time interval, and submit spatial-temporal query request to the *Task Scheduler*.

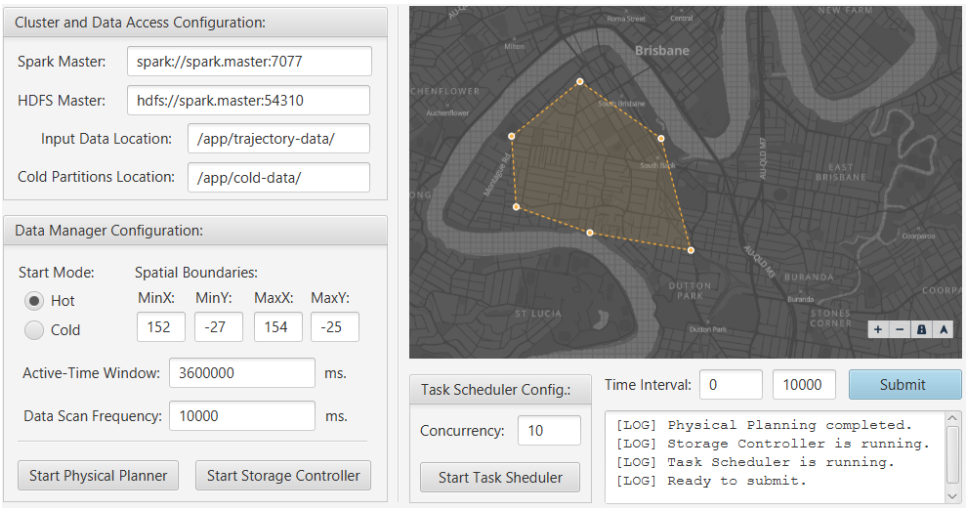


Figure 4.7: System user interface.

Our system is distributed as an open source, which allows contributors to further extend its functionalities. Documentation and download links can be found at the systems repository [1]. We expect that our system will be refined and improved by the research community and developers. The system is built upon a component-based architecture, in which new operations, query predicates, and features can be easily plugged in. While researchers can use our system for experimentation and benchmarking, professionals working on low-latency trajectory-based systems, and where memory is a constraint, can apply its components in the core of their applications and analytics.

[1] https://github.com/douglasapeixoto/spark-trajectory-system

4.5 Experimental Results

We present a set of comprehensive experiments on a real and synthetic trajectory datasets to evaluate the throughput, scalability, and memory usage of our approach.

4.5.1 Experimental Setup

We use a 64GB dataset containing real trajectory data collected throughout China, the dataset contains around 65 million heterogeneous trajectories from taxis and personal vehicles in a period of five days. The dataset is initially stored in HDFS, we use the default MR block size of 64MB.

We implemented a system using our architecture to perform the experiments. The system was implemented using the Spark Java library version 2.0.1. Experiments are conducted on a cluster with 16 physical nodes (1 master and 15 slaves). Each node has eight cores and 64GB of memory – in our experiments we configured Spark to use 7 cores and 60GB of memory in each slave node.

We compare the scalability, throughput, and memory usage of our system against OceanST [178], another state-of-the-art system for trajectory data storage using Spark. We evaluate the scalability of our work by varying the dataset size and the number of nodes in the cluster; throughput is evaluated based on the number of user queries executed within a time frame. We evaluate memory usage by comparing the total memory used by the system as the time passes, and versus the number of input queries in both Hot and Cold modes.

Data Manager Configurations: The spatial model was built from 100,000 sample trajectories randomly selected. Spatial partitions were divided into 10 time pages, that is, $\theta/\vartheta = 10$. The Active-Time window φ, i.e. the time in which the partitions are allowed to be stored in main memory without been queried until they are considered as Cold, was set as 10s, 30s, 60s, 120s respectively in the experiments.

Query Manager Configurations: We simulate 3,000 spatial-temporal user queries generated randomly to cover the dataset area, where in 90% are placed in the same spatial-temporal region, covering a total of 20% of the map area, in order to simulate hotspots; 10% were randomly generated throughout the entire spatial-temporal region of the input dataset. User queries are submitted to the Task Scheduler in batches of 105 concurrent queries (one per available slave core).

We compared four different configurations of our system with the state of the art, in order to demonstrate the saves in memory usage of our system, yet keep nearly same throughput as the state-of-the-art system.

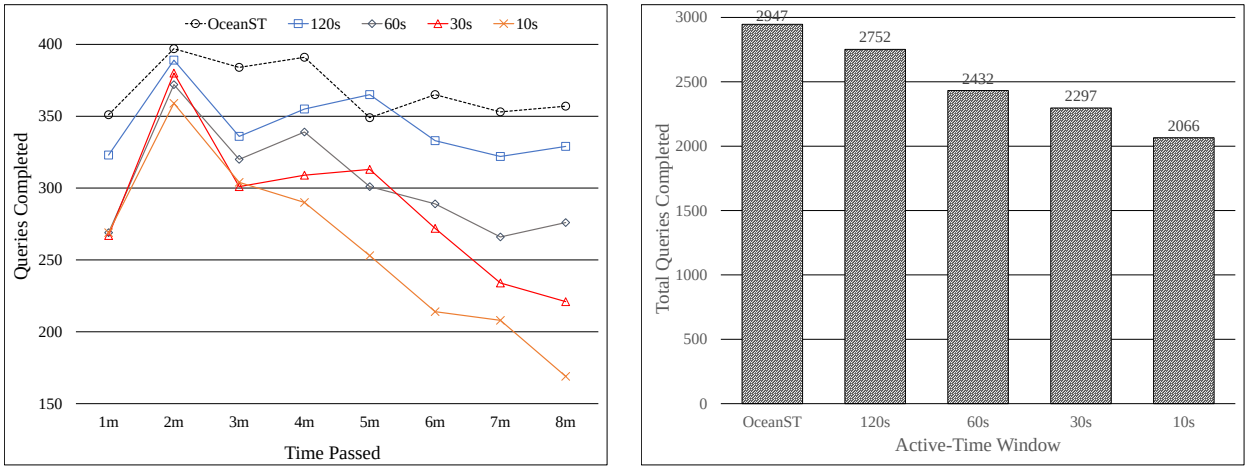
4.5.2 System Throughput and Scalability

In this section we evaluate the system throughput under different circumstances in both Hot and Cold modes. We submit 3,000 queries in batches of 105 concurrent queries in the same order in all experiments, and measure the throughput (number of completed queries) every 10 seconds; the results are displayed in clusters of 1 minutes.

Effect of the Active-Time Window

Figure 4.8(a) shows the system throughput during a period of 8 minutes in Hot-mode. There is a sharper increase in throughput in the first minutes due to the fact that all partitions are initially stored in memory; the throughput, however, decreases with time once cold partitions send to be stored on disk. The decrease in throughput is more evident with lower Active-Time windows, since performance is affected by higher I/O operation with lower Active-time Windows. For higher Active-time Windows, however, the tendency is to throughput stabilize near CloST. Valleys in the throughput are caused by queries on cold areas being executed, which also affects performance due to I/O, this is more evident with lower time windows, since partitions become cold faster.

Figure 4.8(b) shows the total number of queries completed after a period of 8 minutes in Hot-mode. As the figures demonstrate, our architecture achieved near same throughput as CloST. Although the system throughput decreases with the Active-Time Window, this is justified by high gains in memory usage, as we show in the next experiments.



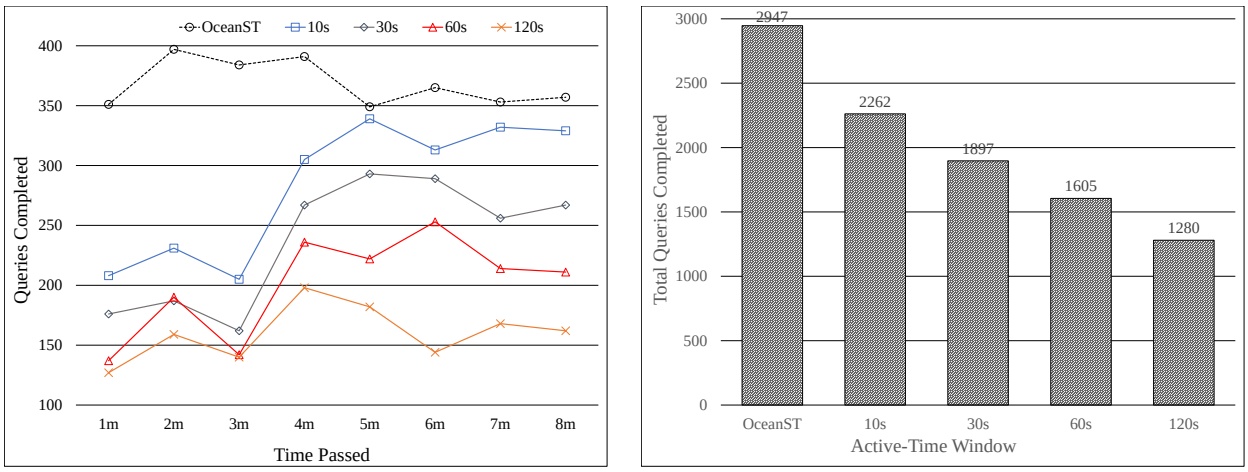
(a) System throughput as the time passes.

(b) Total system throughput after 8 minutes.

Figure 4.8: System throughput for different Active-time windows in Hot-Mode (all the dataset initially stored in Memory).

Similarly, Figures 4.9(a) and (b) show the same experiments, now with the system starting in Cold-Mode. In contrast with the previous experiment, the system throughput increases with time, since the first queries to be executed need to read the necessary partitions from disk. As the time passes, the number of hot partitions loaded to memory increase, so does the performance of future queries and consequently the overall system throughput. As shown in Figure 4.9(b), the total throughput in Cold-Mode in inferior than that in Hot-Mode, this is due to the heavier load on the first queries, since they are executed with partitions on disk. However, as the time passes the throughput of both approaches tends to stabilizes in the same values as expected.

In conclusion, starting the application in Hot-Mode conducts to a higher overall throughput; however, the throughput in both modes tends to converge as the system stabilizes. In addition, smaller Active-Time windows tends to negatively affect the overall system throughput, due to the increase



(a) System throughput as the time passes.

(b) Total system throughput after 8 minutes.

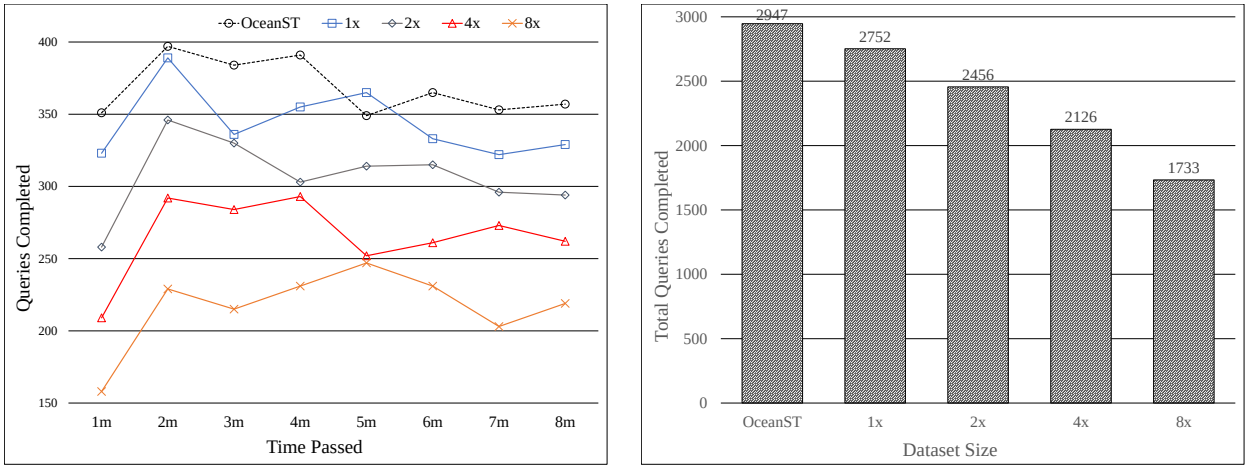
Figure 4.9: System throughput for different Active-time windows in Cold-Mode (all the dataset initially stored on Disk).

in I/O and decrease in the number of partitions in memory. However, in Cold-Mode the system achieved higher gains in memory usage, the same happened for smaller Active-Time windows, as we demonstrate in further experiments.

Effect of the Dataset Size:

In this experiment we evaluate the scalability of our system, and investigate how the throughput varies with datasets of different sizes. We perform the previous experiment using 1x, 2x, 4x, and 8x the size of the original input dataset. We fix the Active-Time window to 120s in this experiment, since it showed better outcome in previous experiments. Figure 4.10(a) shows the system throughput during a period of 8 minutes in Hot-mode using multiple copies of the input dataset against CloST. In all experiments the throughput varied with same pattern, with larger datasets presenting smaller throughput as we expected; this is due to the fact that query performance is affected with larger datasets since partitions become larger, which affect the I/O and the time the partitions are kept in-memory, consequently affecting the overall throughput. However, as Figure 4.10(b) shows the total number of queries completed after 8 minutes, the decrease in throughput is linear, and smaller than the order of increase in the dataset size, which proves the scalability of our approach.

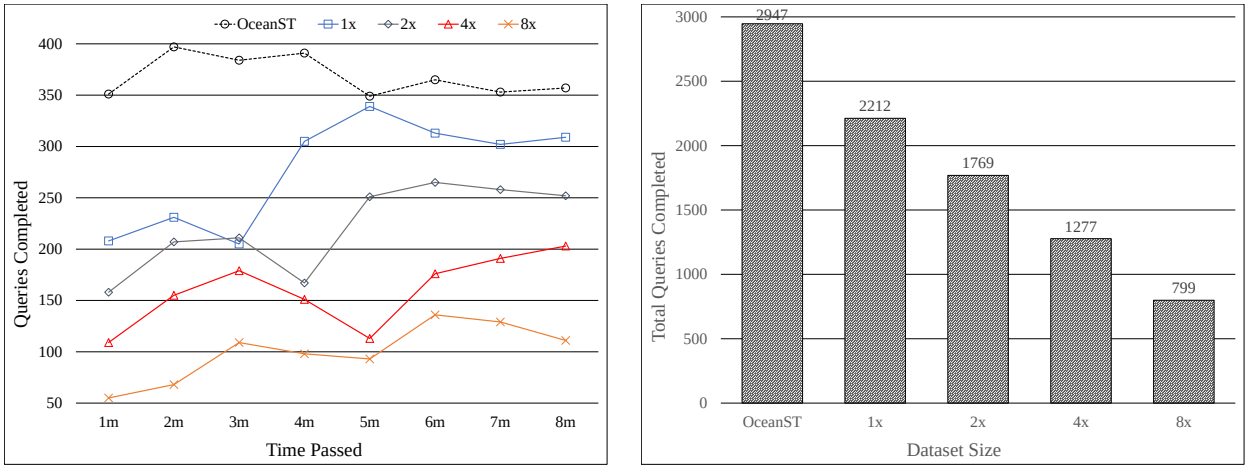
Similarly, Figures 4.11(a) and (b) show the same experiments, now with the system starting in Cold-Mode. Performance and throughput increased in the same pattern in all experiments as data partitions are loaded to main memory and become hot; however, as in the previous experiment, due to delays in query processing for larger datasets, some partitions tend to become idle in the memory for longer, and are more likely to become cold, hence sent to disk, thus the decrease in both query performance and throughput. However, as in Hot-Mode, although more accentuated, the decrease in performance and throughput are linear, and smaller than the order of increase in the dataset size, which proves the scalability of our approach even in Cold-Mode.



(a) System throughput as the time passes.

(b) Total system throughput after 8 minutes.

Figure 4.10: System throughput for different dataset sizes in Hot-Mode (all the dataset initially stored in Memory).



(a) System throughput as the time passes.

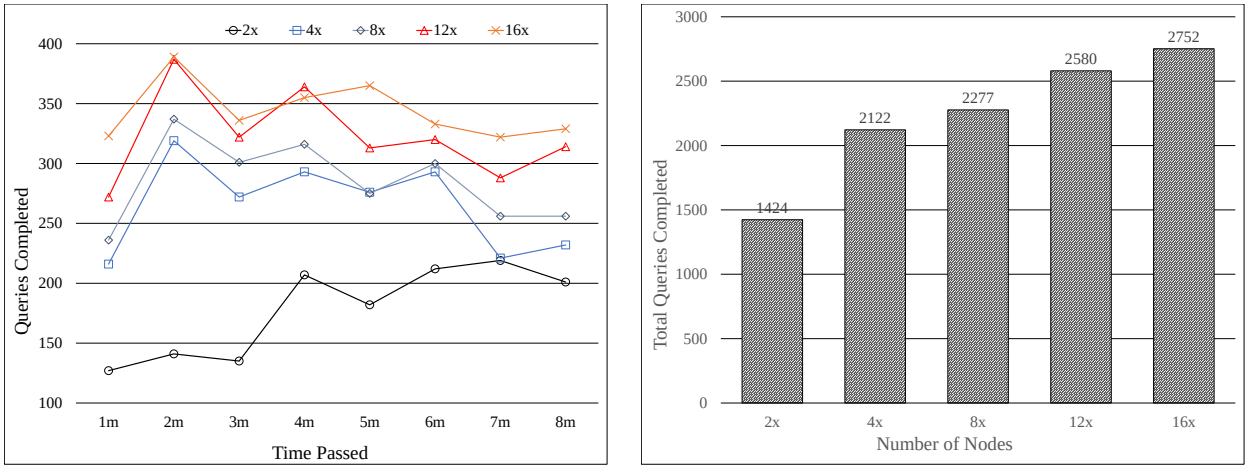
(b) Total system throughput after 8 minutes.

Figure 4.11: System throughput for different dataset sizes in Cold-Mode (all the dataset initially stored on Disk).

In conclusion, starting the application in Hot-Mode conducts to a higher overall throughput in all experiments; however, the throughput in both modes tends to converge to the same values as the system stabilizes. Again, the total throughput in Cold-Mode is inferior than that in Hot-Mode, due to the heavier load on the first queries, since they are executed with partitions on disk. In addition, larger dataset tends to negatively affect the overall system throughput, due to the increase in I/O, this is more evident with larger dataset, firstly because the queries refinement step is affected with larger data in the filtered partitions, and secondary because partitions become cold more often, increasing both query execution time and disk I/O. However, as in previous experiment, in Cold-Mode the system achieved higher gains in memory usage, as we demonstrate in further experiments.

Effect of the Number of Nodes:

In this experiment we evaluate the scalability investigating the throughput variation with the number of nodes in the cluster. We perform the previous experiment using 1, 2, 4, 8, 12, and 16 nodes. We fix the Active-Time window to 120s in this experiment. Figure 4.12(a) shows the system throughput during a period of 8 minutes in Hot-Mode. As with the dataset size, the throughput varied with same pattern for most experiments as the number of nodes in the cluster increases, with exception for 2 nodes where the system reached the peak of memory available in in the cluster. As expected, the greater the number of nodes in the cluster, the better the throughput, this is mostly due to the more cores are available the more queries can be executed concurrently; since up to 4 node the cluster was able to fit the entire dataset in-memory comfortably, memory did not have a much negative effect here. Near 2 nodes, however, performance is jeopardized with not enough memory to store the entire dataset. However, as queries are executed, cold partitions are send to disk, and the performance increases sharply. Figure 4.12(b) shows the total number of queries completed after 8 minutes. As expected the increase in throughput is linear with the number of node available, which also demonstrated the scalability of our approach.

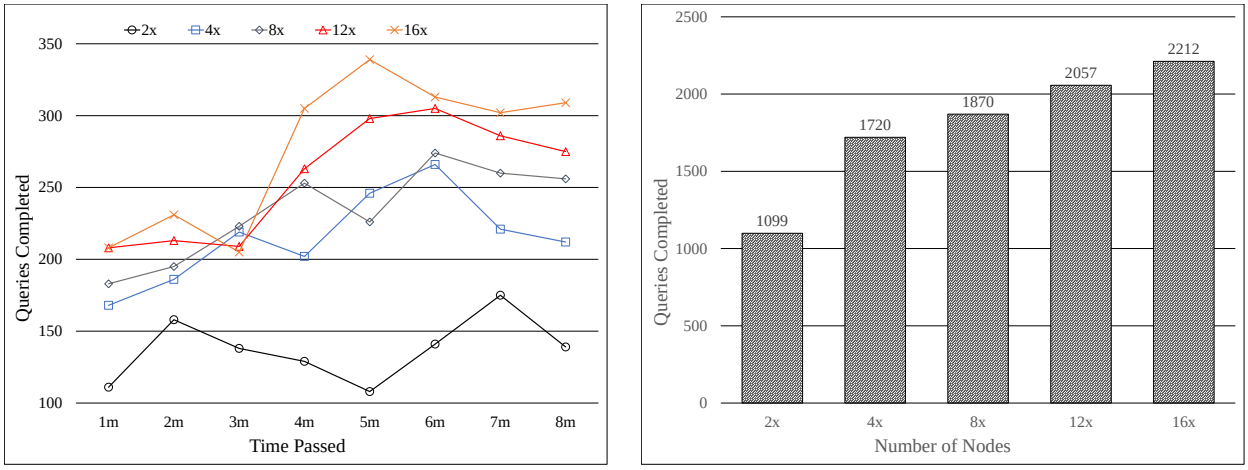


(a) System throughput as the time passes.

(b) Total system throughput after 8 minutes.

Figure 4.12: System throughput for different number of nodes in the cluster in Hot-Mode (all the dataset initially stored in Memory).

Similarly, Figures 4.13(a) and (b) show the same experiments, now with the system starting in Cold-Mode. Performance and throughput increased with the number of nodes in all experiments. In addition, throughput increased more sharply as queries are executed, since memory availability increases due to cold partitions being sent to disk. Similar to Hot-Mode, with 2 nodes the performance is jeopardized by memory availability, leading to higher disk I/O, longer query response time, and delays in the scheduling queues. However, even though the overall throughput in Cold-Mode was smaller than in its Hot-Mode counterpart, as shown in Figure 4.13(b), the increase in throughput with the number of nodes is linear in a same degree as in the former experiment, which also demonstrates the scalability of our approach in Cold-Mode.



(a) System throughput as the time passes.

(b) Total system throughput after 8 minutes.

Figure 4.13: System throughput for different number of nodes in the cluster in Cold-Mode (all the dataset initially stored on Disk).

In conclusion, as in previous experiments, starting the application in Hot-Mode conducted to a higher overall throughput in all experiments. However, the throughput in both modes tends to converge to the same values as the system stabilizes; the exception is for 2 nodes, where the cluster reaches the memory availability. In summary, if memory availability is not a constraint, performance and throughput are affected by the number of processing units, which becomes the bottleneck for larger numbers of concurrent queries. On the other hand, with small numbers of node in the cluster, memory becomes the bottleneck, nevertheless, throughput tends to increase and stabilizes with time as cold partitions are released from memory.

4.5.3 Memory Consumption

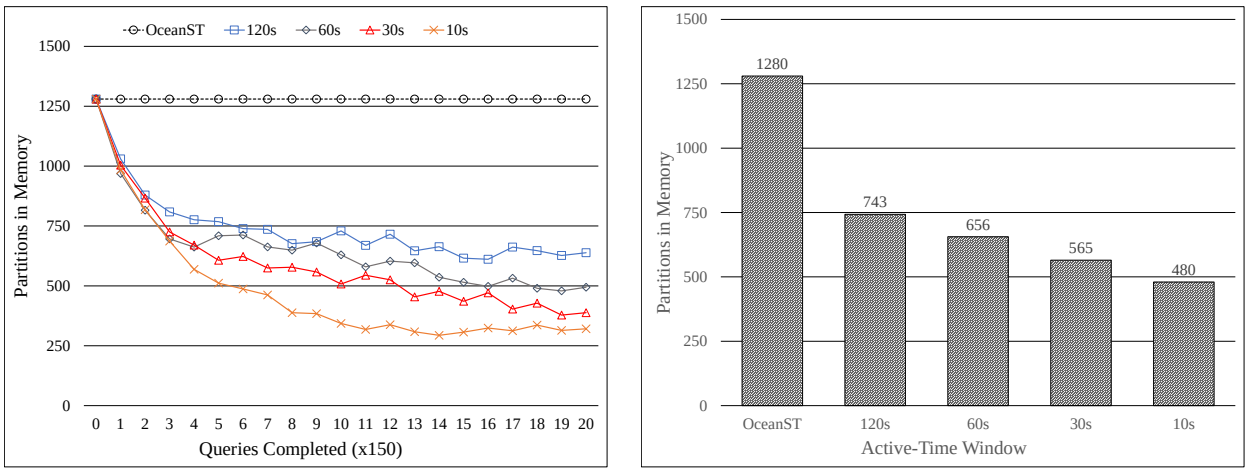
In this experiment we evaluate the effect of the Active-Time Window in the memory usage. Since measure the exact amount of memory used in the cluster by Spark is challenging, we evaluate the consumption based on the number of Hot partitions loaded to memory as queries are executed.

Using a dataset of 64GB raw data with 10 time pages per spatial partition, the total number of spatial-temporal partitions in the dataset after the partitioning phase is equals to 1280, which is in accordance with Equation 4.3, the size of each partition in memory is roughly 60MB (which also accounts for Java objects serialization, thus the total amount is greater than 64GB). We compare the memory usage of our system against CloST using different Active-Time windows in both Hot and Cold modes. We measured the number of partition in memory after every 150 out of 3,000 queries are completed.

Figure 4.14(a) shows the number of partitions in memory as queries as completed. Using CloST all data is stored in main memory, thus the number of partitions in memory is always a constant. With our system in Hot-Mode, however, all partitions are stored in memory only when the application starts; as queries are being submitted the number of partitions in memory decreases with cold partitions being

sent to disk. This is more evident at the beginning of the application runtime, where the system is still learning from the query workload which partitions to keep in memory. In all experiments the number of partitions in memory tends to stabilize with a few small variations due to cold queries being executed. The smaller the time windows the smoother is the learning process, since cold partitions are identified and sent to disk faster, this also explain the lower memory consumption with smaller time windows. For higher time windows, on the oder hand, partitions are kept for longer in memory, and cold partitions take longer to be identified, this also leads to partitions requested by cold queries to stay longer in memory, what explains the more skewed curve in higher time windows.

Figure 4.8(b) shows the average number of partitions in memory after all queries are complete. As mentioned in previous experiments, although the system demonstrated a inferior throughput for smaller time windows, the average memory consumption improves with smaller Active-Time windows, achieving up to 3x gain in memory usage. However even for higher time windows, our system used 2x less memory in Hot-Mode then the other system.

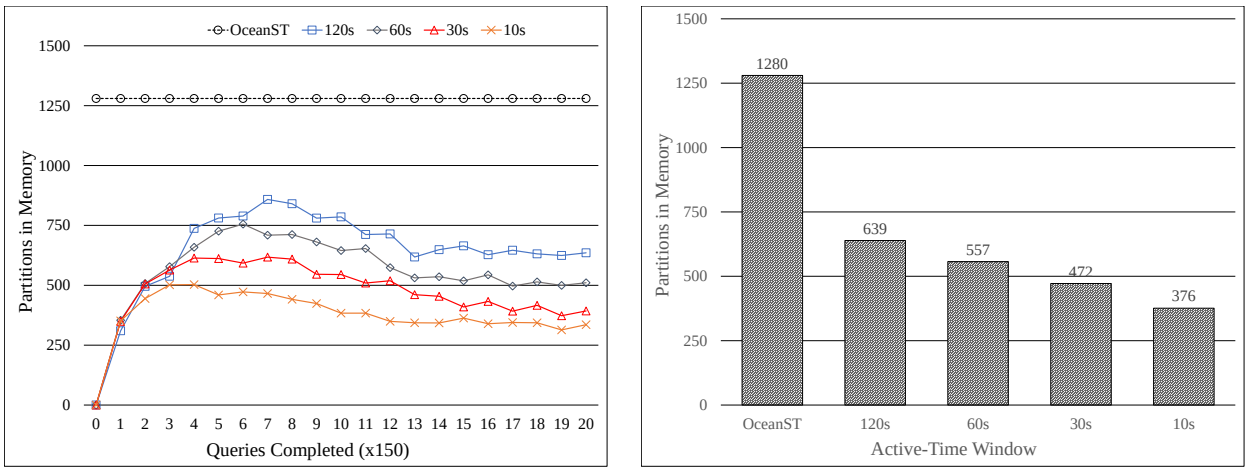


(a) Partitions in memory as queries are executed.

(b) Average number of partitions in memory.

Figure 4.14: System memory usage (number of partitions stored in memory) in Hot-Mode (all the dataset initially stored in Memory).

Similarly, Figures 4.15(a) and (b) show the memory consumption now with the system starting in Cold-Mode, where all partitions are initially stored on disk after partitioning. As queries are executed, partitions are loaded to main memory and become hot. Similarly to Hot-Mode, the number of partitions in memory converges to an optimum while the system learns from the query workload and identifies the hotspots, with increase in memory usage being smoother for smaller Active-Time windows for the same reasons previously described. In both Hot and Cold modes, however, the number of partitions in memory converge to the same figures as expected. The main difference between Hot and Cold modes are in the average memory consumption, as shown on Figure 4.15(b), where the system achieved a smaller average memory consumption in Cold mode due to the entire data being initially stored in memory, therefore partitions that are never requested by user queries, are not stored in memory at any time.



(a) Partitions in memory as queries are executed.

(b) Average number of partitions in memory.

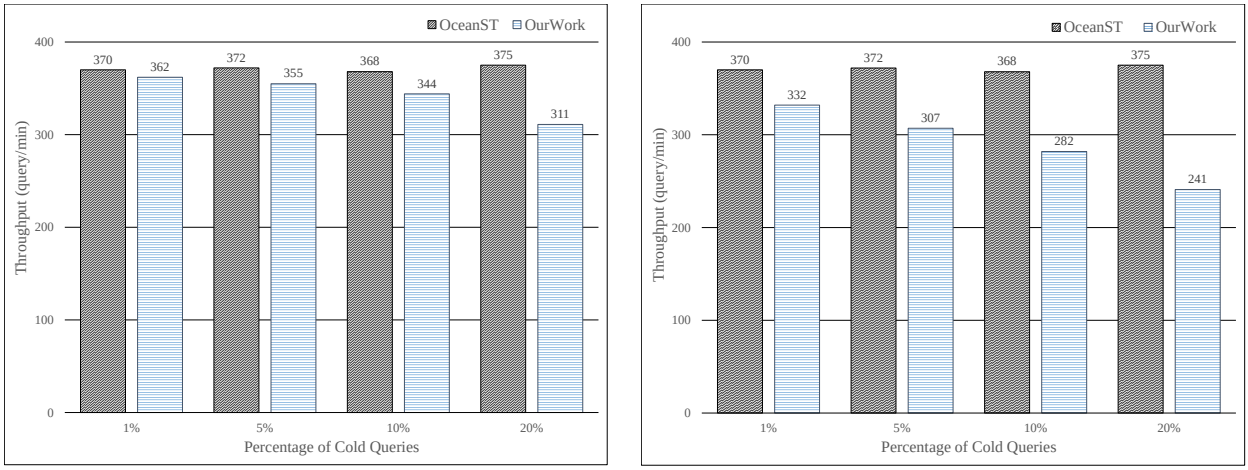
Figure 4.15: System memory usage (number of partitions stored in memory) in Cold-Mode (all the dataset initially stored on Disk).

In conclusion, in both Hot and Cold modes the memory consumption converges to the same figures as the system learns the query hotspots form the workload, with the learning process being smoother for smaller Active-Time windows. The trade-off between memory consumption and throughput is, therefore, achieved based on the Active-Time window size that best fits the memory available in the cluster.

4.5.4 Query Diversity

In this experiment we evaluate the throughput variation as we increase the number of cold queries submitted. We perform this experiment with 1%, 5%, 10%, and 20% of the queries in cold regions respectively, to demonstrate the effects of query diversity on the system's throughput and compared the results with the benchmark system. We fix the Active-Time window to 120s in this experiment. Figure 4.16(a) shows the average system throughput (queries per minute) as we vary the number of cold queries in Hot-Mode. In the benchmark system the throughput remains constant due to all data bein stored in-memory all the time, thus any query, either hot or cold, will have access to the requested data partitions readily in-memory. We noticed a light decrease in throughput as the percentage of query coldspots increased using our approach, this was due to more data being loaded to main-memory due to a larger number of cold queries, which increases disk I/O. However, the decrease in query throughput was very small compared to the increase in the number of query coldspots, furthermore, the memory savings of our approach justify these results, as we previously demonstrated. Figure 4.12(b) shows the same experiment with the system starting in cold mode; here the overall throughput decreased due to more data being loaded from disk; the effects of query variety, however, are similar to the previous experiment, and as such are justified by the savings in memory consumption.

In conclusion, in both Hot and Cold modes our system achieved good throughput, close to the benchmark, with little decrease in throughput as the number of cold queries submitted increases. The



(a) Throughput in Hot-Mode.

(b) Throughput in Cold-Mode.

Figure 4.16: System throughput (queries per minutes) varying the number of cold queries.

savings in memory consumption of our approach, however, justify these results.

4.5.5 Summary

In this chapter we proposed a trajectory storage architecture on top of the Spark framework with resource-wise utilization, and concurrency control for multi-user environments. We exploit the in-memory nature and distributed parallel properties of Spark for scalable and low-latency trajectory data storage and processing. A key feature of our architecture is the ability to identify query hotspots, and exchange data between main-memory and disk based on the query workload, yet leveraging the scalability, fault-tolerance, efficiency, and concurrency control features of Spark. We developed a system on top of our proposed architecture, where administrators are able to setup the cluster configurations, as well as the parameters for data partitioning, physical planning, storage controlling, and the number of concurrent tasks supported by the cluster. Users are able to submit spatial-temporal queries for parallel concurrent processing. We developed a prototype of our system, and demonstrated its ability to process multiple concurrent requests over a large-scale data, yet maintaining steady performance and wise memory consumption, under different query workloads and configurations. Our experiments demonstrated that our system architecture achieved high throughput compared to the state-of-the-art, yet achieving up to 3.5x gain in memory usage. We believe our system will support scientists and professionals working with large-scale trajectory-based applications.

Chapter 5

Top-k Most Similar Trajectories using Spark

5.1 Introduction

Top-k most similar trajectories search (k-NN) is frequently used as classification algorithm and recommendation systems in spatial-temporal trajectory databases. The problem is useful for automatic classification, origin-destiny analysis, and identify objects that move in a same pattern, for instance. However, k-NN trajectories is a complex operation, and a multi-user application should be able to process multiple k-NN trajectories search concurrently in large-scale data in an efficient manner. The k-NN trajectories problem has received plenty of attention, however, state-of-the-art works neither consider parallel processing of k-NN trajectories search nor concurrent queries in distributed environments, or consider parallelization of k-NN search for simpler spatial objects (i.e. 2D points) using MapReduce, but ignore the temporal dimension of more complex data, such as spatial-temporal trajectories. In this work we propose a parallel approach to the k-NN trajectories problem in a distributed and multi-user environment using the Spark framework. We propose a space/time data partitioning based on Voronoi diagrams and time pages, named Voronoi Pages, in order to provide both spatial-temporal data organization and process decentralization. In addition, we propose a spatial-temporal index for our partitions to efficiently prune the search space, improve query latency and system throughput. We implemented our solution on top of Spark's RDD data structure, which provides a thread-safe environment for concurrent MapReduce tasks in main-memory databases. We perform extensive experiments to demonstrate the performance and scalability of our approach.

Motivation and Applications: Given a query trajectory T, a constant k, a time interval $[t_0, t_1]$, and a trajectory dataset S, the top-k nearest neighbor trajectories problem (k-NN), (k-NN, also know in the literature as k-most-similar trajectories), is to find in S the k closest (or most similar) trajectories from T active during $[t_0, t_1]$. k-NN trajectories is one of the most traditional query operations in trajectory databases, and has received plenty of attention, e.g [31], [56], [130], [153], [158]. Applications include, for example, to identify the top-k vehicle's trajectories in a frequent path in order to calculate their average fuel consumption during a certain period of time (e.g. peak hours with more traffic jam),

in order to optimize gas stations placement or logistics optimization. Other applications include identifying seasonal pattens in natural phenomena, such as hurricanes and tornadoes; determine migration patterns of certain groups of animals along the year; and sport research to aid coaches to identify movement patterns of top players. However, processing k-NN trajectories in a multi-user environment is challenging; the application may be serving hundreds of requests over the network, and k-NN search in general demands extensive use of computational resources. Furthermore, k-NN search for trajectories is a complex operation, unlike other simpler spatial objects, trajectories are essentially non-uniform sequential data with variable length, attached with both spatial and temporal attributes; one may also need to consider data uncertainty [191]. Overall, trajectories are considered similar if they follow a certain motion pattern, or move in a similar way (i.e. keep spatially close to each other) for the majority of their time extent.

The Case for Spark: The massive amount of GPS data available, as well as the increasing number of trajectory data application users, demands more robust, reliable and scalable solutions, since real-world location-based service should be able to serve multiple requests over large-scale datasets. Therefore, a typical solution is to consider distributed parallel computation with frameworks such as MapReduce (MR) [39], which provides an abstraction for parallel computation and efficient resources allocation of concurrent threads. MR has became very popular with the increasing interest in moving data into cloud-based systems, multi-core servers, and commodity clusters. Spark [181], on the other hand, provides a MR solution with the goal of speeding up data processing by storing data in main-memory [14], [105], [182], [185]; furthermore, Spark is particularly suitable for iterative algorithms, such as k-NN search.

5.1.1 State-of-the-art

Current state-of-the-art for k-NN trajectories, however, mainly focus on centrally-based computation in single user environments, and cannot be easily tailored to the MR paradigm [31] [56] [130] [153]. Existing research to support spatial queries using MR, e.g. [4] [6] [12] [48] [95], utilize either a multi-core *divide-and-conquer* strategy, where each *mapper* is responsible to process a sub-query over a subset of the dataset, while the intermediate results from the *map* are refined by the *reducers*; or utilize spatially-aware partitioning techniques, such as Grid cells, Quadtrees, and Voronoi diagrams, in order to organize the space into disjoint groups of spatially close objects, providing both process decentralization and efficient space pruning; hence reducing I/O and minimizing data transfer across nodes.

The main drawback of the *divide-and-conquer* approach is that computational resources may be wasted by processing data blocks that does not contribute for the query result. On the other hand, spatial-aware partitioning strategies in MR can achieve up to 10x faster performance than *divide-and-conquer* by maintaining data locality [44], [48], [200], since only a smaller number of partitions containing query candidates are selected for processing, reducing query latency and avoiding unnecessary I/O. The later approach is preferred for MR environments and concurrent threads; first

because the faster the query response time, the sooner it gives resources back to the application; and secondly, location-based services and MR systems are often serving more than one application at same time, e.g. Spark and Hadoop might be serving other applications through their wide set of tools, or serving concurrent jobs on the same application, e.g. multiple k-NN queries. Hence, reducing query latency and resources use allows the system to serve more concurrent requests, and permit our application to work with other MR systems in a non-intrusive way.

The current MR works on k-NN, however, either apply for the spatial dimension only, ignoring the sequential nature and temporal dimension of trajectory data, i.e. k-NN of points [7] [86] [95] [186]; or consider the temporal dimension of trajectories, but does not support similarity-based search [98] [172].

Lu et al. [95] and Akdogan et al. [7] use a Voronoi diagram-based approach to partition the space and index spatial objects based on its closest pivots during the *map* phase, and processing k-NN and RNN queries [7] and k-NN join [95] in iterative *MR* tasks; and outperforms similar MR works based on grid-based partitioning for k-NN query [160], [200] and k-NN join [186]. Overall, Voronoi-based partitioning has been shown to outperform other methods for nearest neighbors search [86], [83], [134]. Our partitioning method is closely related to that in [7], [95], except we extend Voronoi diagrams for spatial-temporal dimension of trajectories in order to support trajectory similarity search, we also use RDD to support in-memory based computation and concurrent queries.

5.1.2 Our Proposal

Our goal is to improve performance and throughput of k-NN trajectory query using MR, and allow concurrent k-NN queries in multi-user servers. Thus, we propose a bulk-loading in-memory partitioning strategy based on Voronoi diagrams and time pages, named *Voronoi Pages*, to support multiple k-NN trajectories query in MR, and a spatial-temporal composite index, named *Voronoi Spatial Index (VSI)* and *Time Page Index (TPI)*, to prune the search space and speed up trajectory similarity search. Voronoi-based partitioning have been successfully used for spatial queries processing in MapReduce, in special distance-based search [7], [86], [95].

Briefly, we uniformly partition the space into Voronoi cells using k-Means clustering, and each Voronoi cell into static temporal partitions (i.e. pages). Trajectories are split into sub-trajectories according to their spatial-temporal extent, such that each sub-trajectory is mapped to one Voronoi Page. We build our Voronoi Pages partitions on top of RDDs to speed up query processing. We process a k-NN query in parallel in a *filter-and-refinement* manner, first filtering candidate pages using our proposed spatial-temporal index, and then running a precise check on each candidate page. Each process unit can manage a number of pages within a RDD in parallel, and multiple concurrent queries can be served by Spark over its RDD. For the best of our knowledge, this is the first work to address similarity-based search for trajectory data using Spark.

5.2 Problem Statement

A k-NN trajectory query retrieve the k closest, or most similar, trajectories from a given query trajectory T. k-NN trajectory queries are useful in pattern recognition, classification, and recommendation systems.

Given a large set of GPS trajectories $\mathbb{T}$, a trajectory distance function $d(T_a, T_b)$, a set of n user specified queries $\mathbb{Q} = \{Q_1, Q_2, ..., Q_n\}$, where $Q_i = (T_i, k_i, t_0^i, t_1^i)$, for every $Q_i \in \mathbb{Q}$ we want to find in $\mathbb{T}$ the k_i-NN(T_i, t_{ia}, t_{ib}), that is, the k_i closest (or most similar) trajectories from T_i w.r.t. $d(T_a, T_b)$, and active during $[t_0^i, t_1^i]$.

We are interested in large-scale k-NN in multi-user applications. Our goal is to improve performance and throughput of k-NN trajectory search using MR in-memory, and allow concurrent queries in multi-user servers.

A naive way to perform the k-NN trajectories in MapReduce is to randomly partition the dataset into blocks, and let each `map()` function calculate the k-NN in each data block in parallel, while the `reduce()` function receives from each *mapper* the k-NN candidates, and finally calculates the final outcome from those candidates.

The main drawback with this approach is that the `map()` may be processing data blocks that does not contribute to the query result, which means a waste of computational resources that should be used in other processes, and expensive *shuffle* cost of sending intermediate results from *mappers* to *reducer*.

An efficient way is to partition the dataset in a spatial-temporal aware manner (i.e. cluster trajectories based on their spatial and temporal proximity – more likely to share neighborhood), such that the amount of data processed by each query is minimized, avoiding unnecessary I/O and use of CPU; hence improving query latency and throughput.

Distance measure: We need a distance function $d(T_i, T_j)$ to calculate the distance between two trajectories [157]. In this work, we adopt an edit distance based measure (i.e. EDwP) as trajectory similarity function. Edit Distance with Projections (EDwP) [130] uses dynamic interpolation to match sample points and calculate how far two trajectories are based on their edit-distances, that is, how many projections must be done to make the trajectories similar to each other; the cost of the projections is calculated on the Euclidean distance over the segments being edited. EDwP can cope with local time shifts and non-uniform sampling rates, which are essential in real-world trajectory datasets. Furthermore, EDwP is threshold-free.

In addition, a good way to choose an appropriate distance measure for a particular application is to use our tool for evaluation of trajectory distance measures, discussed in Appendix A. Tens of similarity measures for trajectory data have been proposed; every technique claim an advantage over the others in a different aspect. Hence, it's difficult for users to choose the best-suited technique, as well as the appropriate parameter values, since each technique has distinct performance and characteristics depending on various factors. Therefore, we develop an application that allows to evaluate several techniques in different aspects (accuracy, sensitivity to trajectory features, performance, etc.). Each

technique has distinct capabilities. This tool will serve as a practical guideline for how to select well-suited trajectory distance measure on particular application scenarios. Users are allowed to vary configurable parameters and visualize their effects. Through empirical observations, users can select the appropriate parameter configuration for their applications. Our tool comes with a library containing all described distance measures and transformations, and makes it easy to add new features and visualize the results. Therefore, using our tool as a reusable framework, developers can reduce development effort.

5.3 Preliminaries

5.3.1 The Case for k-NN using Spark

Spark is particularly suitable for iterative processes, where it is necessary to apply a function repeatedly on the dataset (e.g. gradient descent, k-means, k-NN), since the MR framework reload the data from the file system in every iteration, which incurs in a significant performance loss [135], [181]. Some spatial operation, such as k-NN, demand an iterative neighborhood search in order to process the query answer (see Section 5.6.3). With Spark we can perform iterative MR processing faster by storing data in main-memory.

5.3.2 Spatial Partitioning

Most MR-based works developed strategies to optimize I/O and reduce the network and I/O cost of sorting and grouping the data output from *mapper* to *reducer* (i.e. *shuffle*). In large spatial database applications this can be achieved by means of locality-aware partitioning, which can reduce the number of data objects access, thus reducing network and I/O costs [184].

The cost of executing a k-NN query can be measured by the number of input records it has to read and process [12]. Spatial-aware partitioning strategies, such as Grid cells and Voronoi Diagrams (VD), aim to organize multidimensional data into smaller partitions of spatially close objects to reduce the number of query candidates, hence reducing network and I/O costs [184], [201]. In this work we extend a VD data partitioning for spatial-temporal trajectories in MR, for it maintains data proximity and provides uniform distribution for skewed datasets. VD is particularly suitable for distance-based search [7], [83], [86], [95], where grid partitioning suffer from a significant loss of pruning power [95], [102].

5.3.3 Voronoi-based Space Partitioning

Given n generator pivot elements (e.g. spatial points), a Voronoi diagram partitions the space into n disjoint polygons, where every object in the dataset space is associated with its closest pivot element. Each pivot has a Voronoi polygon (VP) consisting of all spatial elements associated with the pivot. The set of all VPs and their associated elements is called a Voronoi Diagram (VD). An example of

Voronoi diagram is shown in Figure 5.1. The main properties of Voronoi diagrams useful for this work are enlisted as follow. The proof of these properties can be found at [106].

1. The Voronoi diagram of a given set of generator pivots PV is unique.
2. Let n and n_e be respectively the number of pivots and the number of edges in a Voronoi diagram, then $n_e \leq 3n-6$.
3. The nearest generator pivot p_j from another generator pivot p_i is among the pivots whose Voronoi polygons share edges with $VP(p_i)$ (locality preserving property).
4. From property 2, and given that every edge in a Voronoi diagram is shared by exactly two polygons, then the average number of edges per Voronoi polygon is less equal than six, i.e., $2(3n-6)/n = 6-12/n \leq 6$.

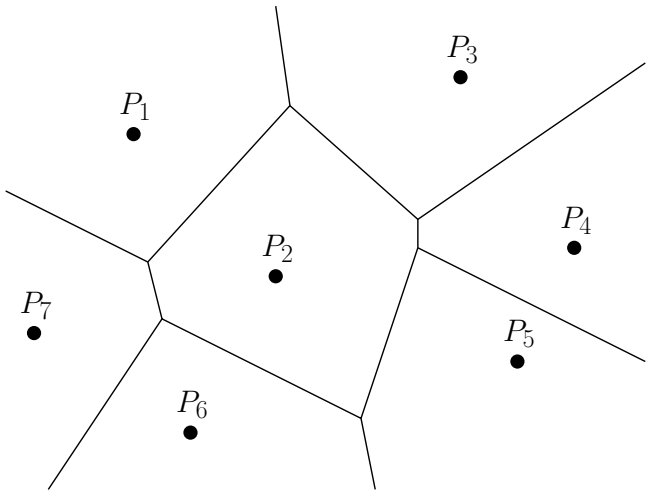


Figure 5.1: Example of Voronoi diagram with seven generator pivots.

5.4 Voronoi Pages Overview

Given an input trajectory dataset, we read and split each trajectory into a set of sub-trajectories, according to its spatial and temporal extent, such that each sub-trajectory is assigned to only one spatial-temporal partition.

Space Partitioning: Given a set of n generator pivots in the dataset space, $PV = \{p_1, p_2, ..., p_n\}$, where $p_i = (x_i, y_i)$, we partition the dataset space into m disjoint spatial partitions, where each trajectory sample point is assigned to its closest pivot (i.e. Voronoi polygon), by computing the Euclidean distance between the trajectory sample points and each $p_i \in PV$. Figure 5.2 illustrates eight trajectories, T_1 to T_8, partitioned across seven Voronoi polygons, P_1 to P_7. Boundary trajectories, e.g. T_1 and T_4, are split into sub-trajectories, where each sub-trajectory is assigned to its overlapping polygon. Section 5.4.1 explains how we address boundary trajectories more precisely. In Section 5.4.2 we discuss how we choose n and PV.

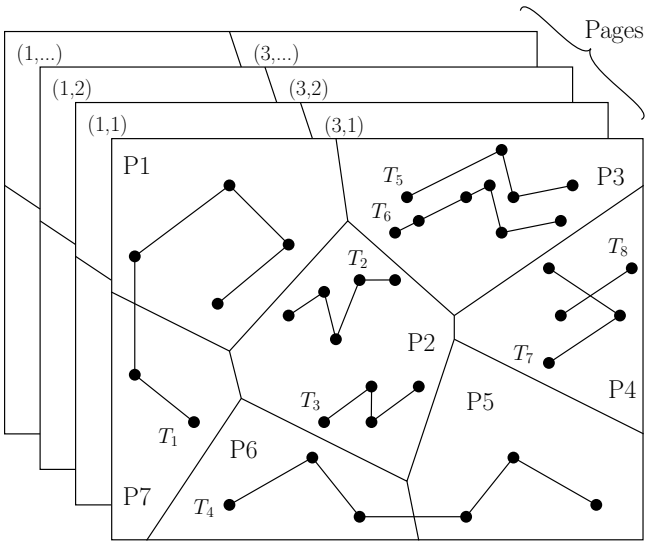


Figure 5.2: Trajectories partitioned across Voronoi polygons, and overview of Voronoi Pages.

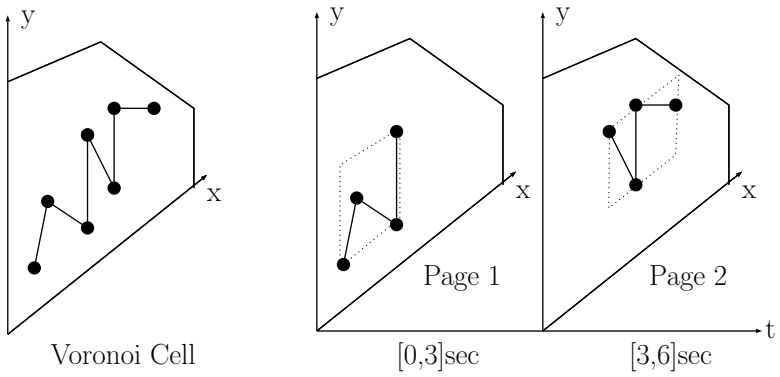


Figure 5.3: Sub-trajectory partitioning into Voronoi Pages, $TW = 3\ sec$. Each page contains sub-trajectories that overlap with both the Voronoi polygon area and time window.

Time Partitioning: Given a time window size TW we split the time space of each Voronoi polygon into static time pages of size = TW, and assign each sub-trajectory sample point inside a polygon to a time page according to its time-stamp. Figure 5.3 illustrates a sub-trajectory in a given Voronoi cell split into time pages. For the sake of simplicity we assume that each sub-trajectory sample point in Figure 5.3 was uniformly collected once every one second, that is $t_i = [0,6]sec$; however, this approach is for both uniform and non-uniform samples.

Voronoi Page: Each time page for a given Voronoi polygon is called a Voronoi Page (VPage), identified by a spatial-temporal index $\langle VSI, TPI \rangle$, where **VSI** (Voronoi Spatial Index) is the page's polygon identifier, and **TPI** (Time Page Index) is the page's time window identifier. In this manner, each sub-trajectory is assigned to a spatial-temporal structure. Each VPage is composed of two structures: (1) a local R-Tree of sub-trajectories in the page, and (2) a list containing the IDs of the trajectories in the page (i.e. the parent's identifier). In our implementation, we use simple R-Tree of sub-trajectory bounding boxes. However, any other index access method for sub-trajectories can be used within a VPage.

5.4.1 Handling Boundary Trajectories

While partitioning both space and time, we expect some trajectories to intersect more than one VPage. The number of intersections highly depends on both the number of spatial partition and the time window size. Tight polygons and small windows size are more likely to have a greater number of intersections. To minimize replication, we split boundary trajectories and replicate only the boundary segments, that is, if a trajectory segment $\overline{p_i p_{i+1}}$ crosses any polygon boundaries (i.e. segment endpoints p_i and p_{i+1} assigned to different polygons), we split the trajectory and assign the boundary segment to both sub-trajectories; each sub-trajectory is assigned to its overlapping polygon. For the temporal dimension the situation is likewise.

5.4.2 Generator Pivots

We choose the number of generator pivots n based on the size of the dataset and the default RDD block size (i.e. 64 MB), so that each task can process data blocks with roughly the same number of polygonal partitions. We study the effect of n for the k-NN performance in Section 5.7.3.

In addition, we must choose the pivots in order to break the space into uniform clusters to avoid load imbalance. Therefore, we use the parallel k-Means++ heuristic [15], provided in the Spark machine learning library (MLlib) [99], which provides a fair approximation of the deterministic k-Means. k-Means partition the dataset into k clusters, in which each spatial object belongs to the cluster with the nearest mean, this results in a partitioning of the space into a Voronoi diagram.

5.5 Voronoi Pages in MapReduce

We assume each input file contains one trajectory per line, as a sequence of spatial-temporal points. We build our VPages structure as a RDD with a `map()` and `reduce()` functions on the input split. The partitioning process returns a RDD of Voronoi Pages.

Map: The *mapper* reads and splits a trajectory T into m sub-trajectories, according to its spatial-temporal dimension, and emits a list of $\langle (VSI, TPI), T_i^{sub} \rangle$ with m pairs, $i \in [1,..,m]$, consisting of a sub-trajectory T_i^{sub} as *value*, and the spatial-temporal index of the VPage containing T_i^{sub} as *key*.

Reduce: The *reducer* receives a list of sub-trajectories (*values*), and groups them by VPage index (*key*), adding each sub-trajectory to the VPage R-Tree. At the end of the parallel process, the *reduce* returns a RDD of $\langle (VSI, TPI), VPage \rangle$ pairs, consisting of the spatial-temporal VPage index, and the final VPage.

The pseudo-code for the partitioning function in Spark MR is shown in Algorithm 2. The Spark context variable **sc** reads a dataset from local or HDFS file system, then map each line of the files to a trajectory object; next each trajectory is mapped to a list of $\langle (VSI, TPI), T_i^{sub} \rangle$ w.r.t. VD and TW.

Algorithm 2 Pages RDD Construction (Steps)

```
1: function PARTITIONING(data: Dataset, VD: VoronoiDigram, TW: TimeWindow)
2:     ℙ^pages_RDD ← sc.textFile("data")
3:         .map(Line ⇒ Trajectory)
4:         .flatMapToPair(VD,TW)(Trajectory ⇒ list[(VSI,TPI), T_i^sub])
5:         .aggregateByKey(Page)(Page.add(T_i^sub))
6:         .cache()
7:     return ℙ^pages_RDD
8: end function
```

The result $\mathbb{P}_{RDD}^{pages}$ is a read-only RDD structure containing the Voronoi Pages for the input parameters. Finally, the $\mathbb{P}_{RDD}^{pages}$ is cached in-memory with the `cache()` command.

5.5.1 Trajectory Track Table (TTT)

We must keep track of sub-trajectories across VPages, so that we can retrieve and rebuild a trajectory when processing a k-NN query. For this purpose, we propose a table-like structure, named **Trajectory Track Table** (TTT). The TTT is a in-memory structure, where each tuple of the table is a pair composed of a trajectory ID and a set of references to VPage (page index hash) containing the pages a trajectory intersects with. The TTT is constructed as an RDD (i.e. $\mathbb{T}_{RDD}^{table}$) so that all nodes have access to it without the need of replication. We build the $\mathbb{T}_{RDD}^{table}$ with MR as follows.

Map: The *mapper* reads and map each input trajectory to a list of pairs $\langle T_{id}, (VSI, TPI)\rangle$, containing the trajectory identifier for each VPage index T_{id} overlaps with.

Reduce: The *reducer* groups VPage indexes by trajectory *key* into a set of $\langle T_{id}, Set\{(VSI, TPI)\}\rangle$ page indexes . Each pair $\langle T_{id}, Set\{(VSI, TPI)\}\rangle$ is henceforth called a table tuple.

5.6 k-NN Trajectories Overview

Given a query trajectory Q, and a time interval $[t_0, t_1]$, we want to retrieve the k-NN of Q within the time interval $[t_0, t_1]$. By using a VD-based approach we focus on the spatial proximity to the specified query location. Let $VP(Q)$ be the set of Voronoi polygons covered by Q, and $VP_N(Q)$ be the set of neighbor polygons of $VP(Q)$, to process k-NN trajectory queries we take advantage of the neighborhood properties of Voronoi diagrams as follows.

5.6.1 NN Trajectory Search Overview

From property 3, the nearest neighbor NN(Q) of a query object Q is either in $VP(p_i)$, where p_i is the nearest pivot from object Q, or among the Voronoi neighbors of $VP(p_i)$, for Q might be a boundary object. However, because our query object Q is a trajectory, we must check all polygons intersecting with Q and their neighbors. For instance, if our query trajectory is T_4 in Figure 5.2, we must search

for NN(T_4) inside P_5 and P_6, and their neighbors P_2, P_4 and P_7. Moreover, we are interested in a spatial-temporal k-NN, thus we have to look in the specific time pages inside each partition. More precisely, assuming our query object is T_4, and we are interested in a time interval $[t_0,t_1]$, we search for the NN(T_4,t_0,t_1) inside the Voronoi Pages set $\mathbb{F} = \{(2,[t_0,t_1]), (4,[t_0,t_1]), (5,[t_0,t_1]), (6,[t_0,t_1]), (7,[t_0,t_1])\}$. Nevertheless, trajectories in $\mathbb{F}$ may span to other spatial-temporal partitions depending on their spatial and temporal extent, for instance, T_1 in P_7 also spans to P_1. We must ensure that the whole trajectories are returned from the previous step in order to evaluate their distances. Thus, from this point we visit the TTT to retrieve the index of other VPages containing the trajectories in $\mathbb{F}$ (if there is any). We filter from the $\mathbb{P}^{pages}_{RDD}$ the sub-trajectories in the VPages returned from the $\mathbb{T}^{table}_{RDD}$ – except those previously retrieved – and append the remainder sub-trajectories to $\mathbb{F}$. A post-processing step is done to merge sub-trajectories in $\mathbb{F}$, and finally compute the NN(Q,t_0,t_1).

5.6.2 k-NN Trajectories Search Overview

To calculate the remainder (k-1)-NN of Q we use an approach similar to that in [7]. Suppose both Q and NN(Q) are inside P_3, $Q = T_5$ and NN(T_5) $= T_6$ for instance, thus we also look for the second NN of Q in pages inside the neighborhood of P_3, that is P_1, P_2 and P_4. The remainder NNs are retrieved in the same recursive process; the search stops at the k_{th} iteration if the number of candidates c is $c \geq k$, or continues the search until $c \geq k$. From property 2, the number of neighbor partitions we have to look for time pages in every iteration is at most six for every partition containing the current candidate.

5.6.3 k-NN Trajectories in MR

The VPages containing the k-NN result are unknown until the query is executed, thus, we calculate k-NN(Q,t_0,t_1) with k iterative *filter-and-refinement* MR jobs, so that in every i_{th} iteration we have the i_{th}-NN(Q,t_0,t_1) result. Iterative MR processes are better performed by choosing the RDD in-memory storage level [180], [181], thus we run our k-NN algorithm by persisting the $\mathbb{P}^{pages}_{RDD}$ in main-memory only.

First Filter: In the first *filter* we select all pages in the interval $[t_0,t_1]$, for every polygon $P_i \in (VP(Q) \cup VP_N(Q))$ (lines 1–6 in Algorithm 3). Finally, we perform a whole selection using the $\mathbb{T}^{table}_{RDD}$ to collect all trajectories inside the filtered VPages, and active during $[t_0,t_1]$ (lines 8–12), as stated in Section 5.6. Algorithm 3 contains the steps for the *filter* task in Spark, and returns a RDD of candidate trajectories $\mathbb{T}^{candidates}_{RDD}$ within the candidate VPage $\mathbb{F}^{pages}_{RDD} \subset \mathbb{P}^{pages}_{RDD}$. Algorithm 3 uses the RDD's `filter()` function, which returns a subset from a parent RDD with objects checked against a given predicate (e.g. VPage index, trajectory id).

First Refinement: The first *refinement* receives the RDD of candidate trajectories $\mathbb{T}^{candidates}_{RDD}$ from the *filter* step, and returns a list of trajectories sorted by distance to Q. The pseudo-code for the NN refinement step is in Algorithm 4, it sets the distance from every trajectory T in the candidate

Algorithm 3 NN Trajectory Filter (Steps)

1: **function** QUERYFILTER(Q: QueryTrajectory, $[t_0,t_1]$: TimeInterval, TW: TimeWindow)
/* (1) get candidate pages index */
2: $VSI_{list} \leftarrow VP(Q) \cup VP_N(Q)$
3: $TPI_0 \leftarrow (t_0/TW)+1$
4: $TPI_1 \leftarrow (t_1/TW)+1$
/* (2) filter pages by index */
5: $\mathbb{F}^{pages}_{RDD} \leftarrow \mathbb{P}^{pages}_{RDD}$.filter(
6: Page $\Rightarrow$ Page.index.VSI **in** VSI_{list} **and**
7: Page.index.TPI **in** $[TPI_0, TPI_1]$)
/* (3) the ids of the trajectories in $\mathbb{F}^{pages}_{RDD}$ */
8: $T^{id}_{set} \leftarrow \mathbb{F}^{pages}_{RDD}$.getTrajectoryIdSet()
/* (4) filter from the TTT tuples w.r.t. T^{id}_{set} */
9: $I^{index}_{set} \leftarrow \mathbb{T}^{table}_{RDD}$.filter(Tuple $\Rightarrow T^{id}_{set}$.contains(Tuple.key))
/* (5) filter other pages w.r.t. I^{index}_{set} */
10: $\mathbb{F}^{pages}_{RDD} \leftarrow \mathbb{F}^{pages}_{RDD} \cup \mathbb{P}^{pages}_{RDD}$.filter(Page $\Rightarrow I^{index}_{set}$.contains(Page.index))
/* (6) collect w.r.t. T^{id}_{list} and post-process */
11: $\mathbb{T}^{candidates}_{RDD} \leftarrow \mathbb{F}^{pages}_{RDD}$
12: .flatMapToPair(T^{id}_{list})(Page $\Rightarrow$ pairsList(T_{id},T^{sub}_i))
13: .reduceByKey((T^{sub}_i, T^{sub}_j) $\Rightarrow$ postProcess(T^{sub}_i, T^{sub}_j))
14: **return** $\mathbb{T}^{candidates}_{RDD}$
15: **end function**

partitions to Q. If one is interested in the 1-NN(Q,t_0,t_1) only, the application returns the first element in $NN(Q,t_0,t_1)_{candidates}$ list as the 1-NN(Q,t_0,t_1) result.

Algorithm 4 NN Trajectory Refinement (Steps)

1: **function** QUERYREFINEMENT(Q: QueryTrajectory, $[t_0,t_1]$: TimeInterval, $\mathbb{T}^{candidates}_{RDD}$: CandidatePartitions)
2: $NN(Q,t_0,t_1)_{candidates} \leftarrow \mathbb{T}^{candidates}_{RDD}$
3: .map(T $\Rightarrow$ T.setDistance($d(T,Q)$))
4: .sort().collect()
5: **return** $NN(Q,t_0,t_1)_{candidates}$
6: **end function**

Next Filter-Refinement: For every i_{th}-NN of Q remaining, we perform a *filter-and-refinement* process in a fashion as similar as before. More precisely, taking the example on Figure 5.2, suppose $Q = T_5$, and the first element in $NN(Q,t_0,t_1)_{candidates}$ list is 1-NN(T_5) = T_6, in the same Voronoi Page of T_5. The second NN of Q is found by adding to the candidates list the trajectories in the neighborhood of 1-NN(Q), that is, pages in the interval $[t_0,t_1]$ inside $VP_N(T_6) = \{P_1,P_2,P_4\}$. However, $VP_N(T_6)$ are already known from the previous step, so we can return the first and second elements from the candidates list as the result of 2-NN(Q,t_0,t_1). Now, assuming the second NN of Q is inside P_4, the third NN is found by adding to the candidates list the trajectories covered by pages in the interval $[t_0,t_1]$ inside $VP_N(T_4) = \{P_2,P_5\}$; P_2 is already known, so we only filter pages inside P_5. The process is

Table 5.1: Trajectory dataset information. Time, speed, and length columns are the average values.

#Trajectories	#Points	Time	Speed	Length
4,000,000	354,294,752	543 s	37 km/h	6.5 km

repeated for every i_{th}-NN remaining. At the end of each i_{th} stage the intermediate results are collected and the candidates list is updated in the application master.

5.7 Experiments

We conduct a set of experiments on a real trajectory dataset to evaluate the performance and scalability of our approach. We compare the performance and scalability of our prosed VD based approach against a Grid-cell based approach, also commonly used in spatial MR works, e.g. [48], [172]. The grid-based approach is similar to the VD one, except the space is partitioned into a uniform grid. Throughout this section we refer to the VD approach and Grid cells approach as *VPages* and *GPages* respectively. To process k-NN queries in *GPages* we employ a technique similar to that in SpatialHadoop [48] to prune the search space, except we use the trajectories' centroid distances to select candidate trajectories. To a fair comparison, both *VPages* and *GPages* have same spatial partitioning granularity, and same time window size; we also apply the same trajectory splitting strategy on both approaches. We perform our experiments with RDDs in main-memory storage level only.

5.7.1 Experimental Setup

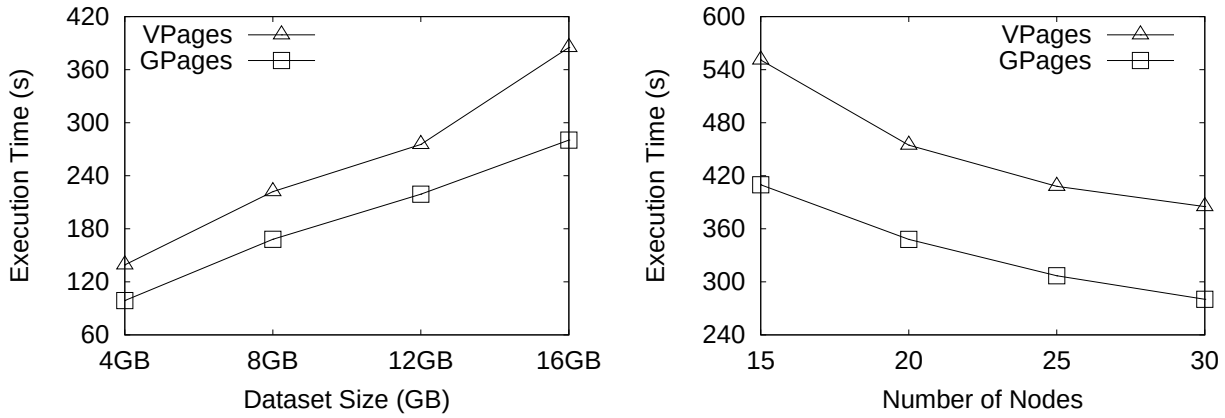
We use a 16GB trajectory dataset collected from Shanghai and southern region of China. The dataset contains 4 million heterogeneous trajectories from taxis and personal vehicles in a period of five days. The data is initially stored in HDFS. More information about the dataset is given in Table 5.1. Each input file contains one trajectory per line in the format: trajectory identifier, and a list of (x coordinate, y coordinate, and time-stamp).

All algorithms are implemented in the Spark Java library version 1.5.1. Experiments are conducted on a cluster with 30 nodes. Each node is a Ubuntu 14.04 LTS with dual-core processor and 3GB of memory, all nodes are connected through gigabit Ethernet. We employ Spark-JobServer [140] to allow multiple concurrent jobs in our application (i.e. concurrent queries over the *VPages/GPages* RDDs), we set 0.6 as Spark's default data cache value (i.e. 60% of RAM for cache data, and 40% for shuffle).

Table 5.2 shows the default values and the range of each parameter using during the experiments. We evaluate our method for both NN and k-NN trajectory queries. We set $k = 10$ by default. The time window size was also fixed at $1,200sec$ (based on the mean $\mu = 543s$ and standard deviation $\sigma = 700s$ of trajectories duration), so most trajectories fit into one time page. We chose the number of Voronoi cells as 250, 500, $1,000$ and $2,000$, so that the RDD contains roughly $8, 4, 2$ and 1 polygonal partitions per block respectively. We also noticed that with less than 15 nodes we were not able to cache the entire dataset into main-memory and perform concurrent queries with our limited cluster resources,

Table 5.2: Parameters Settings.

Parameter	Default	Range
# of Polygons/Cells	1,000	250-2,000
Time Window Size	1,200 sec	-
Dataset Size	16GB	4GB-16GB
# of Nodes	30	15-30
# of Concurrent Queries	5	5-30
# of Neighbors (k)	1 & 10	1 & 10-40
# of Input Queries	100	-



(a) Index contruction time by dataset size.

(b) Index construction time by number of nodes.

Figure 5.4: Index construction evaluation.

thus we set 15 as the minimum number of nodes. As query input value, we randomly selected 100 trajectories from the dataset, the query time was set as the beginning and ending time of each query trajectory; we perform the queries in batches of 5 concurrent threads by default. We evaluate the performance on building the *VPages* partitions, and the scalability and throughput of our approach on processing concurrent queries for different parameters.

5.7.2 VPages Construction Evaluation

In this section we evaluate the performance to create the *VPages* with different numbers of generator pivots, and the scalability against *GPages*. Figure 5.4 shows the overall results of this experiment.

Index Construction Scalability: Figure 5.4 (a) demonstrates the execution time for reading the data from HDFS and building both *VPages* and *GPages* RDDs for different dataset sizes, i.e. from 1/4x to 1x the original dataset. *GPages* outperformed *VPages* on index construction time on all scenarios due to the one-to-one complexity of parsing trajectory data points to a uniform grid, against the O($n * k$) complexity of Voronoi diagram construction. This is also true for different numbers of computing nodes as shown in Figure 5.4 (b). Overall, *GPages* demonstrated to be 10%–50% more scalable than *VPages* on index construction, however, *VPages* outperformed *GPages* in query latency and throughput as we will discuss on next sections.

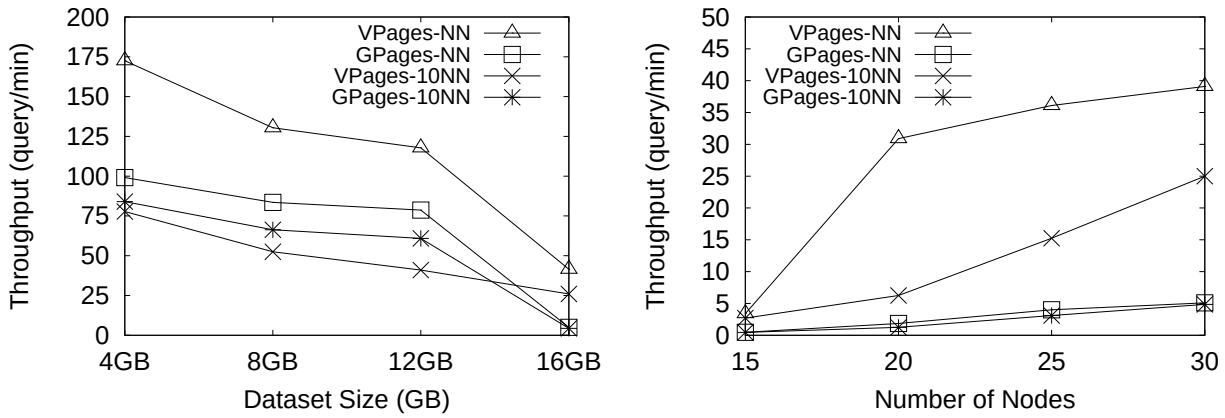
Table 5.3: Trajectories distribution across *VPages* by number of pivots. The #Splits column contains the average values.

#Pivots	#VPages	#Sub-Trajectories	#Splits	Latency (s)
250	78,715	6,045,863	1.51	247.5
500	146,479	6,276,712	1.57	301.5
1,000	265,700	6,538,746	1.63	385.0
2,000	464,912	6,949,443	1.74	497.0

Effect of the Number of Pivots: Table 5.3 gives statistical information about trajectories distribution across *VPages* and the execution time on building the *VPages* RDD for different numbers of Voronoi cells. As expected, the execution time tends to increase with the number of cells, this is due the increasing number of comparisons during the map phase. The number of trajectories' splits increase with the spatial partitioning granularity, this is due to increasing number of boundary trajectories in more tight partitions. However, query throughput increases for larger numbers of Voronoi cells as we will discuss in the next sections.

5.7.3 System Performance and Scalability

In this experiment we study the system performance and scalability to process NN and *k*-NN trajectory queries on both *VPages* and *GPages*. We measure the system throughput by the number of queries completed per minute for each approach. Figure 5.5 shows the overall results for this section.

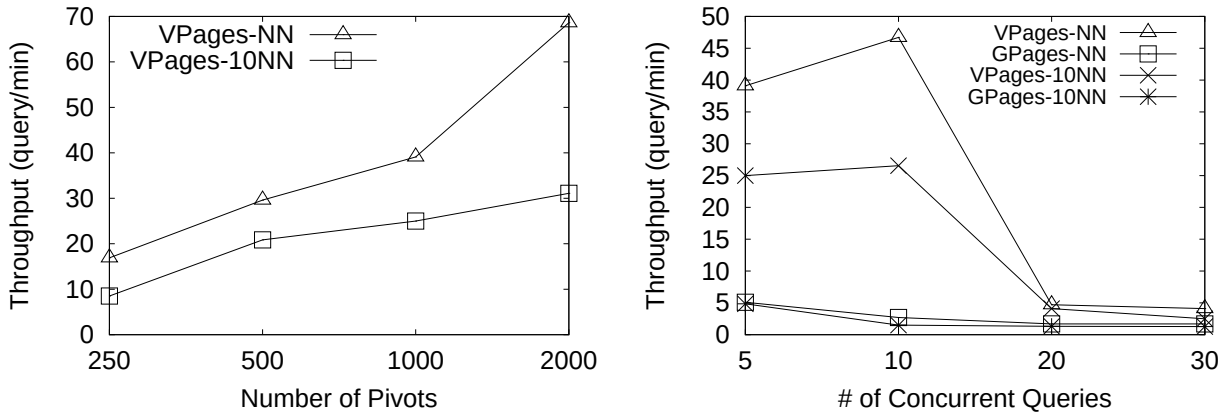


(a) System throughput by dataset size.

(b) System throughput by number of nodes.

Figure 5.5: System throughput evaluation.

Scalability Evaluation: Figure 5.5 (a) shows the system throughput for NN and 10-NN queries on both *VPages* and *GPages* RDDs. Overall, *VPages* performed up to 10x better than *GPages* for both NN and *k*-NN queries as the dataset grows. This is mainly due to two reasons: first the filter step of *VPages* is more accurate than its *GPages* counterpart on filtering candidate trajectories; secondly, *VPages* presented a more uniform data distribution across partition using k-Means clustering than the grid-based approach, which caused the load imbalance in *GPages*. However, for dataset smaller



(a) System throughput by number of pivots.

(b) System throughput by number of concurrent queries.

Figure 5.6: System throughput by number of pivots and by number of concurrent queries.

than 16GB, *GPages* performed k-NN search slightly better than *VPages*; this is due to the iterative neighborhood search on *VPages*, which seeks for the query result on neighbor cells even for small input datasets. This difference, however, disappears as the dataset grows due to the most homogeneous data distribution of *VPages*.

Near 16GB for *GPages*, however, the cluster resources utilization reaches its limits for the default parameters, once each concurrent query needs to cache and process its own copy of the filtered RDD partitions, which causes Spark to shuffle more data and spill some data to disk for larger input datasets, causing both network and I/O bottleneck, thus the performance deterioration on *GPages*. Therefore, *VPages* outperformed *GPages* in 10x for 16GB, i.e. 40.0 throughput in *VPages* versus 4.0 in *GPages*. Overall, *VPages* demonstrates to be more scalable than *GPages* for both NN and k-NN and the dataset grows. The situation is likewise with number of nodes smaller than 20 nodes, as shown in Figure 5.4 (b), where *VPages* outperformed *GPages* in all scenarios up to 25x in NN search and up to 10x in k-NN search.

Effect of the Number of Pivots

Figure 5.6 (a) gives the system throughput using *VPages* for different numbers of Voronoi cells. Overall, finer-grained partitions tends to positively affect query latency and throughput, this is due to the filter step to be more precise when retrieving candidate trajectories. In other words, more polygons leads to less false positives in the filter step, hence a faster refinement. This improvement in query latency increases the resources availability in the cluster, hence increasing parallelism and system throughput.

Concurrency Evaluation

Here we evaluate the effect of the number of concurrent queries to the system throughput. We submit queries to the application in batches of 5 to 30, and start one thread per query job using the Spark-JobServer [140] framework. Queries are executed in a "round-robin" fashion using Spark's *FAIR* job

scheduling, so that all queries get a roughly equal share of cluster resources, which is the indicated mode for multi-user applications in Spark[1]. Figure 5.6 (b) gives the overall results for this experiment.

For *VPages* on both NN and *k*-NN queries the system throughput increased from 5 to 10 concurrent queries, this is due to the best use of our cluster resources. We noticed that with fewer than 10 concurrent queries the cluster resources were not at full use with some idle nodes. Near 10 concurrent queries, however, the resources utilization reaches its peak, hence its maximum throughput. Furthermore, even with dataset in main-memory the overhead of managing large numbers of concurrent jobs can lead to more contentions and strongly limit the system scalability [109]. For the default parameters our cluster was unable to support greater numbers of concurrent queries, which caused the performance deterioration due to network and I/O bottleneck. For *GPages* the situation was much worse, with its peak near 5 concurrent jobs. In summary, *VPages* demonstrated to be up to 15x better on handling multi-user application and concurrent jobs at the cluster resources utilization peak. The maximum batch size can adjusted accordingly based on the cluster's memory available.

Effect of Number of Neighbors (k)

Here we evaluate how the cardinality of the number of neighbors *k* affects the system throughput for *k*-NN trajectories search. Figure 5.7 gives a comparative on the system throughput as *k* grows in both *VPages* and *GPages*.

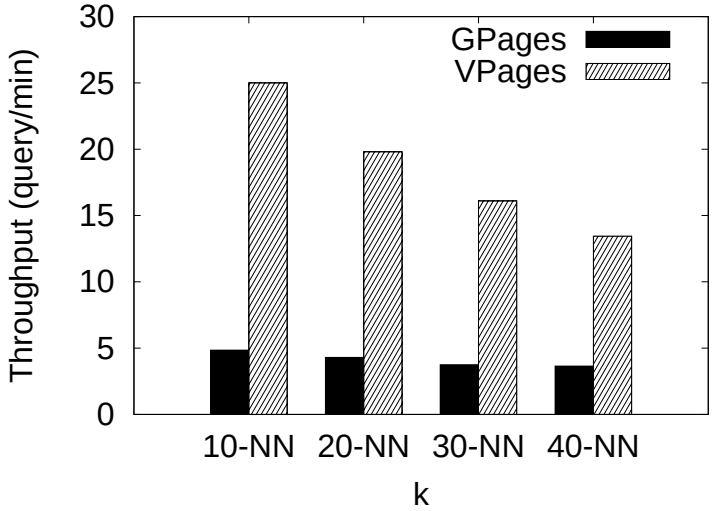


Figure 5.7: Query Number of k.

On both *VPages* and *GPages* approaches the partitions containing the *k*-NN are unknown until the query is executed; however, spatial locality is not always preserved in grid-based, which means we need to extent the search space in *GPages* further than in *VPages* to retrieve the candidate trajectories, which negatively impacts the performance of *GPages* for all values of *k*. Recalling Section 5.6, in each iteration on *VPages* the current *i*-NN is retrieved, along with its neighbor trajectories; however, due to the locally preserving property of VDs, most neighbors of a given object are in the nearby polygons,

[1]Spark Job Scheduling: https://spark.apache.org/docs/1.3.0/job-scheduling.html

thus are retrieved in the first iterations; and due to the homogeneous distribution of the data in the diagram, the number of trajectories in the neighbor partitions to retrieve are roughly the same, which leads to near linear effect on query latency as k increases; the system throughput, therefore, is directly affected by queries latency. Although the throughput of approaches decrease near linearly as k grows, *VPages* demonstrates to be more sensitive to k than *GPages*, i.e. by linear regression $\alpha_v = -0.38$ and $\alpha_g = -0.04$, where α_v and α_g are the angular coefficients for *VPages* and *GPages* respectively. However, *VPages* is only as poor as *GPages* for very big values of k, where a great number of partitions need to be track.

5.8 Summary

In this section we present a multi-user system to process concurrent k-Most-Similar trajectories (k-NN) search using Spark's RDD, a thread-safe and resilient distributed data structured for large-scale data processing in main-memory using the MapReduce model. We introduced a novel spatial-temporal data partitioning approach, named Voronoi Pages, built on top of RDD to a scalable and fast processing of multiple k-NN trajectories search in MR. Voronoi Pages provides both homogeneous data partitioning and spatial-temporal locality preserving, essentials for MR-based systems. Our experimental results based on a real trajectory dataset demonstrates the superiority in performance and scalability of our approach against another common approach used in MR for spatial data.

Chapter 6

Conclusion

In this thesis we proposed a novel database system for trajectory data management on top of the Spark framework. We developed a wide range of techniques and applications for large-scale spatial-temporal trajectory data management, aiming three important aspects of large-scale trajectory data management, (1) data preparation and preprocessing; (2) scalable, reliable, and resource-wise storage; and (3) efficient and accurate query processing. For the best of our knowledge, this is the first work to cover all this range of important features for large-scale trajectory data. In summary, the main contributions and achievements of this thesis are:

Trajectory Data Representation and Integration

We developed a novel parallel system for trajectory data integration and representation, with support for lossless trajectory data compression, and synthetic trajectory data generation. This system also provides templates for trajectory data representation (e.g. spatial-temporal attributes, textual attributes) providing a single data model for integration of different input datasets. In addition, in order to represent and integrate data from different formats, we introduced the *Trajectory Data Description Format (TDDF)*, a data description format for spatial-temporal trajectory data. Moreover, this application is responsible to collect statistics of the input dataset (i.e. metadata). Finally, our application has been published in the DASFAA conference [117].

Efficient Map-Matching at Scale

Map-matching is an important pre-processing step to improve trajectory data quality and reduce uncertainty, due to inaccuracy of raw GPS data. The large amount of digital data available, however, has introduced a new problem of how to match massive amounts of both map and trajectory data in a efficient manner. In this thesis we proposed a Spark-based framework for the problem of large-scale offline map-matching. We introduced new features on top of Spark to allow efficient, scalable, and memory-wise processing of large-scale map-matching. First, we introduced a cost function for the distributed map-matching problem. Secondly, we use a sample-based quad-index construction, and

Quadtree co-partition of map and trajectory data to allow parallel and load-balanced map-matching. We build our partitions on top of Spark's RDD to achieve efficiency and scalability. We employ a safe boundary threshold, and wise split strategy to reduce replication. Finally we proposed a batch-based method for large-scale map-matching, using data loading and processing in smaller batches to reduce memory usage. A comparative study and experiments demonstrated that our framework achieved good efficiency and scalability on map-matching processing with low memory consumption. Finally, the results of our work have been accepted for publication in the DAPD journal [119].

Workload-Aware Trajectory Data Storage and Retrieval

With the increasing demand for low-latency services over large-scale trajectory data, a database system should be able to serve multiple requests over large-scale datasets, providing good scalability, high throughput, and fast query response. In this thesis we proposed a trajectory storage architecture on top of the Spark framework with resource-wise utilization, and concurrency control for multi-user environments. We exploit the in-memory nature and distributed parallel properties of Spark for scalable and low-latency trajectory data storage and processing. Our architecture was designed to react to changes in the query workload efficiently, since some spatial regions, such as urban areas, receive more query requests (hotspots), thus data records in such areas receive priority for in-memory storage over least requested data. In addition, we used a hierarchical partitioning for trajectory data loading efficiency. We developed a system on top of our proposed architecture, where administrators are able to setup the cluster configurations, as well as the parameters for data partitioning, physical planning, storage controlling, and the number of concurrent tasks supported by the cluster. Users are able to submit spatial-temporal queries for parallel concurrent processing. Our experiments demonstrated that our system architecture achieved high throughput compared to the state-of-the-art, yet achieving up to 3.5x gain in memory usage.

Distance-based Trajectory Data Storage and Processing

In this thesis we proposed a multi-user system to process concurrent k-Most-Similar trajectories (k-NN) search using Spark's RDD, a thread-safe and resilient distributed data structured for large-scale data processing in main-memory using the MapReduce model. We introduced a novel spatial-temporal data partitioning approach, named Voronoi Pages, built on top of Spark's RDD to scalable and fast processing of k-NN trajectories search in MR. Voronoi Pages provided both homogeneous data partitioning and spatial-temporal locality preserving, essentials for Spark-based systems. Our experimental results based on a real trajectory dataset demonstrated the performance and good scalability of our approach against another common approach used in MapReduce for spatial data. Finally, the results of our work have been publish in the ADC conference [115].

Trajectory Distance Measures Evaluation

Measuring the similarity (or distance) between trajectories of moving objects is a common procedure taken by most trajectory data-driven applications. However, tens of similarity measures for trajectory data have been proposed; every technique claim an advantage over the others in a different aspect. Hence, it's difficult for users to choose the best-suited technique, as well as the appropriate parameter values, since each technique has distinct performance and characteristics depending on various factors. Therefore, in this thesis we developed an application to evaluate trajectory distance measures. The target users (researchers and developers) can use our tool to configure and evaluate state-of-the-art algorithms for a potential application. This tool is built upon a component-based architecture, in which new techniques can be easily plugged in. We believe that this tool will serve as a practical guideline for both researchers and developers. While researchers can use our tool to assess existing or new techniques, developers can reuse its components to reduce the development complexity. Our tool has been submitted for demonstration in MDM [116].

Final Considerations

Range query and k-NN query are the two most fundamental query operations in trajectory databases. Our system architecture was designed to support both operations, in addition to data integration and pre-processing. Due to the nature and complicity of each operation, two different approaches were designed to solve and optimize each query. Each approach provides its own data partitioning technique and query workflow. However, users can access both functionalities together in the system, since both approached were designed on top of the Spark's RDD. Users can use one single dataset in the system to perform any of these queries, since an RDD can be built separately to solve each problem; furthermore, each approach was designed to reduce memory usage, which reduces the weight of storing the same dataset into two different RDDs in-memory. Therefore, both approaches can be used intertwined in the same distributed environment, since every approach has its own storage controller and query processor component.

For the best of our knowledge, this is the first work to cover all this range of important functionalities for large-scale trajectory data. Furthermore, our system was built using a component based design, and it's well documented, therefore its easy to include new features and components into our application. We believe our system will serve as the API of choice for trajectory data management and analytics. Since our system is provided as open-source, we expect the scientific and industrial community to contribute and extend our system, including more features and functionalities for trajectory data management, mining and analytics.

Appendix A

Concept for Evaluation of Techniques for Trajectory Distance Measures

A.1 Introduction

Measuring the similarity (or distance) between trajectories of moving objects is a common procedure taken by most trajectory data-driven applications. One of the biggest challenges of trajectory distances measurement is that the distance needs to be carefully defined in order to reflect the true underlying similarity. This is due to the fact that trajectories are essentially non-uniform sequential data with variable length, attached with both spatial and temporal attributes, which may or may not be considered for similarity measures. Therefore, tens of similarity measures for trajectory data have been proposed; every technique claim an advantage over the others in a different aspect. Hence, it's difficult for users to choose the best-suited technique, as well as the appropriate parameter values, since each technique has distinct performance and characteristics depending on various factors. Therefore, we develop an application that allows to evaluate several techniques in different aspects (accuracy, sensitivity to trajectory features, performance, etc.). We believe that this tool will be able to serve as a practical guideline for both researchers and developers. While researchers can use our tool to assess existing or new techniques, developers can reuse its components to reduce the development complexity.

Motivation and Applications: The problem of detecting similar trajectories is useful for decision making applications based on moving objects analysis; for instance, one may be interested in planning a road network capacity, planning municipal transportation or detect usual road paths in a city to avoid traffic jam. In this sort of problem, trajectory similarity analysis and query processing, such as the k-NN trajectories [115], play an important role.

However, trajectory distance measurement is challenging due to the nature and complexity of trajectory data. Besides, one must take into account other variants such as shape, time shifting, non-uniform sampling rates, and rotation, for instance. Overall, trajectories are considered similar if they follow a certain motion pattern, or move in a similar way for the majority of their time extent.

To fully tackle this challenge, dozens of similarity measures for trajectory data have been proposed in the literature [162] [157]. For example, there are similarity measures only considering the spatial dimension, such as Euclidean distance, DTW [173], EDR [31], ERP [30], LCSS [153], DISSIM [56], LIP [120], EDwP [130], TID [152], OWD [91], and PDTW [80]; whereas there are trajectory similarity measure for both spatial and temporal dimensions, such as STED [179], STLIP [120], Frechet [9], and STLCSS [154]. Many of these works, and their extensions, have been widely cited in the literature and applied to facilitate the processing and mining of trajectory data.

However, each work claim superiority in identifying similar trajectories under different circumstances, such as noisy data, different sampling rates or scale, or under rotation and translation. As a result, understanding the capability of these techniques, for a given type of application, is difficult to comprehend. Therefore, we present an Application for Evaluation of Techniques for Trajectory Similarity/Distance Measures with the following functionalities:

- **Choose well-suited techniques.** Each technique has distinct capabilities. This tool will serve as a practical guideline for how to select well-suited trajectory distance measure on particular application scenarios.

- **Guide to select appropriate parameters.** Allow users to vary configurable parameters and visualize their effects. Through empirical observations, users can select the appropriate parameter configuration for their applications.

- **Reduce development complexity.** Due to the number and complexity of approaches, it can be challenging and time-consuming for users to understand and implement all techniques. Our tool comes with a library containing all described distance measures and transformations, and makes it easy to add new features and visualize the results. Therefore, using our tool as a reusable framework, developers can reduce development effort.

To support these functionalities, we design our tool with three main features: (i) trajectory data transformation module, (ii) re-implement state-of-the-art trajectory distance measures within a common framework, (iii) a mean to evaluate these techniques with different parameters using a GUI. To the best of our understanding, this is the first system to provide these attractive features.

A.2 System Design

Figure A.1 illustrates the application GUI, which is built upon three modules:

1. **Transformation module:** is responsible to load the datasets and perform a set of transformations on the second dataset as per user specification. The supported trajectory transformations are: add noise, shift points, add or remove points, change sampling rate or scale, time shifting, rotation, and translation.

2. **Distances Computing module:** for a given user-specified trajectory distance function, and two input trajectory datasets A and B, this module computes the distances between every trajectory in A to every trajectory in B (i.e. distance join) after the required transformations.

3. **Visualization module:** Once the results from every experiment has been completed, users are able to load the results for visualization and analysis.

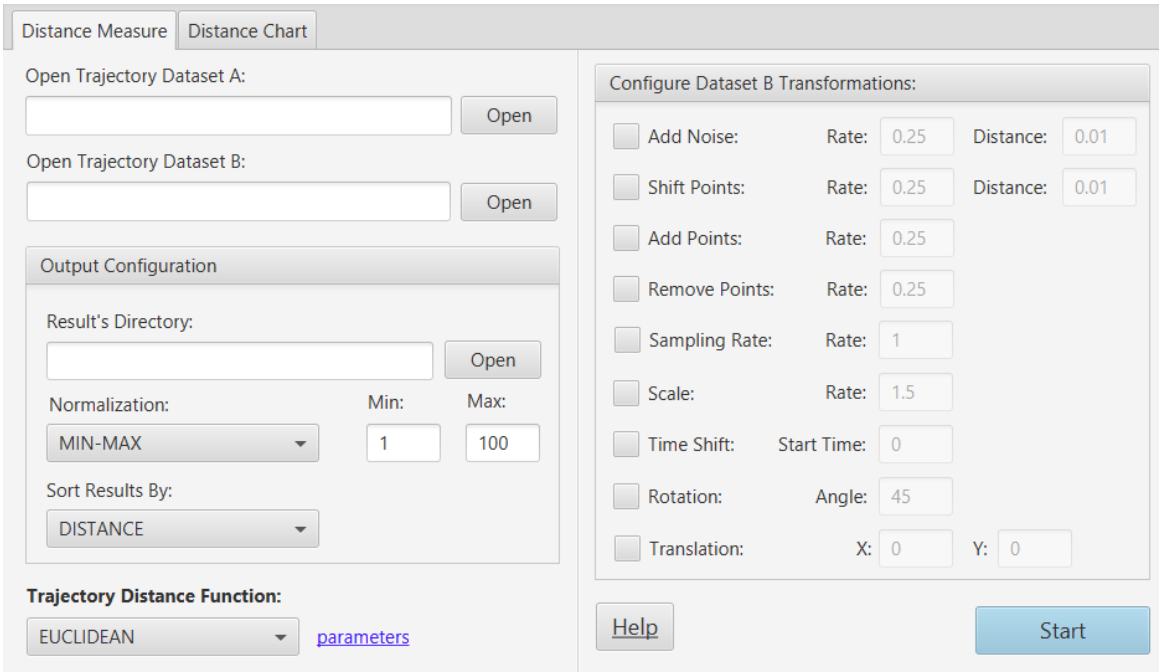


Figure A.1: User Interface.

The project is available to the public at our repository [1].

A.3 Demonstration

Load and transformation: Users are able to load two trajectory datasets A and B, and choose any of the provided transformations to be performed on the dataset B. Users are free to set the parameters of each transformation.

Distance computation: Users are able to choose among 15 different techniques for trajectory distance measure. Since every technique has its own distinct set of parameters, user are able to configure every technique individually once the function is selected.

Normalization: Since there is no consent about the values, or range of values, returned from each distance technique, the application is able to output the results using either Min-Max or Mean-Std normalization, as per user specification.

Output results: After computation, results are saved in a CSV format containing the list of distances for every trajectory in the dataset A to every trajectory in the dataset B, after the required

[1] https://github.com/douglasapeixoto/trajectory-distance-benchmark

transformation and for the specified distance measure. Results can be sorted either by the trajectories IDs or by distance, which is useful for k-NN computation, for instance.

Results visualization: Users are able to load the result files into the application for visualization, as illustrated in Figure A.2.

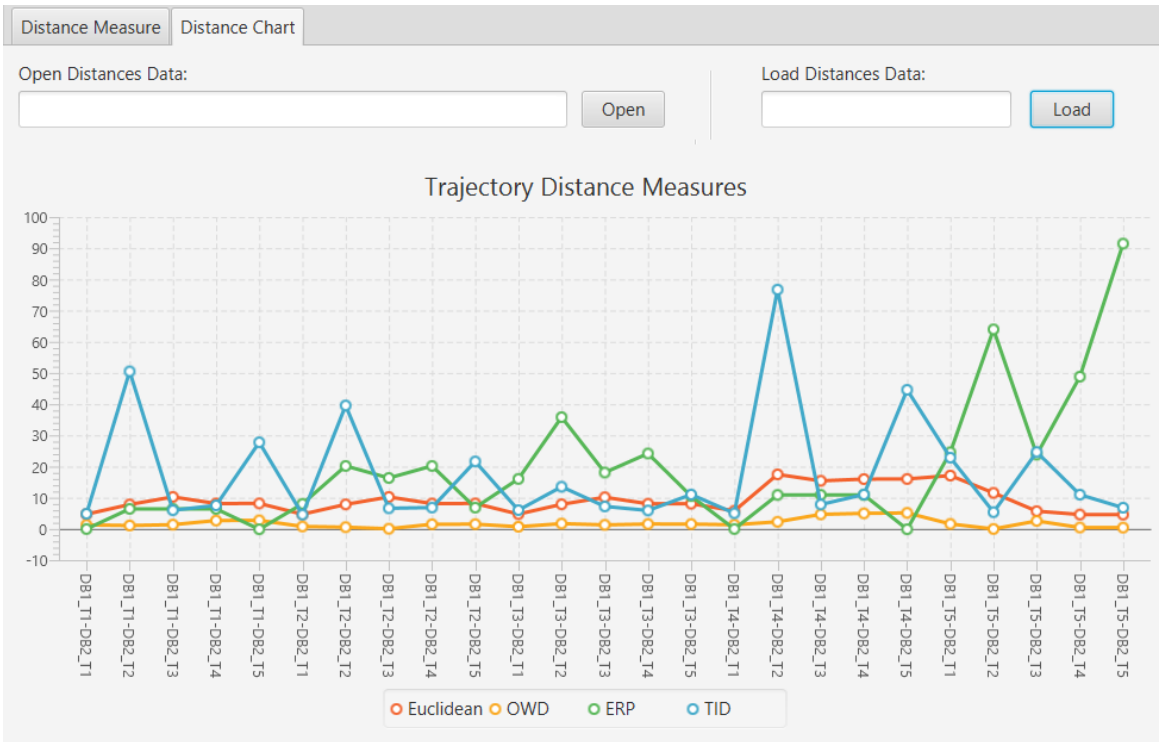


Figure A.2: Trajectory Distances Comparison Chart.

Bibliography

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *VLDB*, 2(1):922–933, 2009.

[2] Y. Ahmad and C. Koch. Dbtoaster: A sql compiler for high-performance delta processing in main-memory databases. *VLDB*, 2(2):1566–1569, 2009.

[3] A. Aji, G. Teodoro, and F. Wang. Haggis: Turbocharge a mapreduce based spatial data warehousing system with gpu engine. In *SIGSPATIAL*, pages 15–20. ACM, 2014.

[4] A. Aji and F. Wang. High performance spatial query processing for large scale scientific data. In *SIGMOD*, pages 9–14, 2012.

[5] A. Aji, F. Wang, and J. H. Saltz. Towards building a high performance spatial query system for large scale medical imaging data. In *SIGSPATIAL*, pages 309–318. ACM, 2012.

[6] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop-gis: a high performance spatial data warehousing system over mapreduce. In *VLDB*, volume 6, pages 1009–1020, 2013.

[7] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi. Voronoi-based geospatial query processing with mapreduce. In *International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 9–16. IEEE, 2010.

[8] H. Alt, A. Efrat, G. Rote, and C. Wenk. Matching planar maps. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 589–598. Society for Industrial and Applied Mathematics, 2003.

[9] H. Alt and M. Godau. Computing the fréchet distance between two polygonal curves. *Intl. Journal of Computational Geometry & Applications*, 1995.

[10] L. O. Alvares, V. Bogorny, B. Kuijpers, J. A. F. de Macedo, B. Moelans, and A. Vaisman. A model for enriching trajectories with semantic geographical information. In *SIGSPATIAL*, page 22. ACM, 2007.

[11] A. M. Aly, H. Elmeleegy, Y. Qi, and W. Aref. Kangaroo: Workload-aware processing of range data and range queries in hadoop. In *International Conference on Web Search and Data Mining*, pages 397–406. ACM, 2016.

[12] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah. Aqwa: adaptive query workload aware partitioning of big spatial data. In *VLDB*, volume 8, pages 2062–2073, 2015.

[13] M. Armbrust, T. Das, A. Davidson, A. Ghodsi, A. Or, J. Rosen, I. Stoica, P. Wendell, R. Xin, and M. Zaharia. Scaling spark in the real world: performance and usability. *VLDB*, pages 1840–1843, 2015.

[14] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *SIGMOD*, pages 1383–1394, 2015.

[15] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii. Scalable k-means++. In *VLDB*, volume 5, pages 622–633, 2012.

[16] F. Baig, M. Mehrotra, H. Vo, F. Wang, J. Saltz, and T. Kurc. Sparkgis: Efficient comparison and evaluation of algorithm results in tissue image analysis studies. In *VLDB Workshop on Big Graphs Online Querying*, pages 134–146. Springer, 2016.

[17] T. Barclay, J. Gray, and D. Slutz. Microsoft terraserver: a spatial data warehouse. In *SIGMOD Record*, volume 29, pages 307–318. ACM, 2000.

[18] V. Botea, D. Mallett, M. A. Nascimento, and J. Sander. Pist: An efficient and practical indexing technique for historical spatio-temporal point data. *GeoInformatica*, 12(2):143–168, 2008.

[19] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On map-matching vehicle tracking data. In *VLDB*, pages 853–864. VLDB Endowment, 2005.

[20] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: identifying density-based local outliers. In *ACM SIGMOD Record*, volume 29, pages 93–104, 2000.

[21] Y. Cai and R. Ng. Indexing spatio-temporal trajectories with chebyshev polynomials. In *SIGMOD*, pages 599–610. ACM, 2004.

[22] X. Cao, L. Chen, G. Cong, C. S. Jensen, Q. Qu, A. Skovsgaard, D. Wu, and M. L. Yiu. Spatial keyword querying. In *Conceptual Modeling*, pages 16–29. Springer, 2012.

[23] A. Cary, Z. Sun, V. Hristidis, and N. Rishe. Experiences on processing spatial data with mapreduce. In *International Conference on Scientific and Statistical Database Management*, pages 302–319. Springer, 2009.

[24] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *VLDB*, 1(2):1265–1276, 2008.

[25] V. P. Chakka, A. C. Everspaugh, and J. M. Patel. Indexing large trajectory data sets with seti. In *CIDR*, volume 1001, page 12, 2003.

[26] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *Transactions on Computer Systems*, 2008.

[27] G. Chatzimilioudis, C. Costa, D. Zeinalipour-Yazti, W.-C. Lee, and E. Pitoura. Distributed in-memory processing of all k nearest neighbor queries. *TKDE*, 28(4):925–938, 2016.

[28] S. S. Chawathe. Segment-based map matching. In *IEEE Intelligent Vehicles Symposium*, pages 1190–1197. IEEE, 2007.

[29] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: an experimental evaluation. In *VLDB*, volume 6, pages 217–228. VLDB Endowment, 2013.

[30] L. Chen and R. Ng. On the marriage of lp-norms and edit distance. In *VLDB*.

[31] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD*, pages 491–502, 2005.

[32] Z. Chen, H. T. Shen, and X. Zhou. Discovering popular routes from trajectories. In *ICDE*, pages 900–911, 2011.

[33] Z. Chen, H. T. Shen, X. Zhou, Y. Zheng, and X. Xie. Searching trajectories by locations: an efficiency study. In *SIGMOD*, pages 255–266, 2010.

[34] C. Coffey, A. Pozdnoukhov, and F. Calabrese. Time of arrival predictability horizons for public bus routes. In *SIGSPATIAL*, pages 1–5. ACM, 2011.

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

[36] J. Dai, B. Yang, C. Guo, and Z. Ding. Personalized route recommendation using big trajectory data. In *ICDE*, pages 543–554, 2015.

[37] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2008.

[38] M. De Choudhury, M. Feldman, S. Amer-Yahia, N. Golbandi, R. Lempel, and C. Yu. Automatic construction of travel itineraries using social breadcrumbs. In *ACM Conference on Hypertext and Hypermedia*, pages 35–44, 2010.

[39] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[40] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *SIGMOD*, pages 1243–1254. ACM, 2013.

[41] R. Ding, Q. Wang, Y. Dang, Q. Fu, H. Zhang, and D. Zhang. Yading: fast clustering of large-scale time series data. *VLDB*, 8(5):473–484, 2015.

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

[43] A. Eldawy, L. Alarabi, and M. F. Mokbel. Spatial partitioning techniques in spatialhadoop. In *VLDB*, volume 8, pages 1602–1605, 2015.

[44] A. Eldawy, Y. Li, M. F. Mokbel, and R. Janardan. Cg_hadoop: computational geometry in mapreduce. In *SIGSPATIAL*, pages 294–303. ACM, 2013.

[45] A. Eldawy, M. Mokbel, and C. Jonathan. Hadoopviz: A mapreduce framework for extensible visualization of big spatial data. In *ICDE*. IEEE, 2016.

[46] A. Eldawy and M. F. Mokbel. A demonstration of spatialhadoop: an efficient mapreduce framework for spatial data. *VLDB*, pages 1230–1233, 2013.

[47] A. Eldawy and M. F. Mokbel. Pigeon: A spatial mapreduce language. In *ICDE*, pages 1242–1245. IEEE, 2014.

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

[49] A. Eldawy, M. F. Mokbel, S. Alharthi, A. Alzaidy, K. Tarek, and S. Ghani. Shahed: A mapreduce-based system for querying and visualizing spatio-temporal satellite data. In *ICDE*, pages 1585–1596. IEEE, 2015.

[50] A. Eldawy, M. F. Mokbel, and C. Jonathan. A demonstration of hadoopviz: an extensible mapreduce system for visualizing big spatial data. *VLDB*, pages 1896–1899, 2015.

[51] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: fast data analysis using coarse-grained distributed memory. In *SIGMOD*, pages 689–692. ACM, 2012.

[52] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. *Fast subsequence matching in time-series databases*, volume 23. ACM, 1994.

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

[54] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The sap hana database–an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[55] Z. Feng and Y. Zhu. A survey on trajectory data mining: Techniques and applications. *IEEE Access*, pages 2056–2067, 2016.

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

[57] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SIGOPS*, volume 37, pages 29–43. ACM, 2003.

[58] F. Giannotti, M. Nanni, F. Pinelli, and D. Pedreschi. Trajectory pattern mining. In *SIGKDD*, pages 330–339, 2007.

[59] C. Y. Goh, J. Dauwels, N. Mitrovic, M. Asif, A. Oran, and P. Jaillet. Online map-matching based on hidden markov model for real-time traffic sensing applications. In *International Conference on Intelligent Transportation Systems (ITSC)*, pages 776–781. IEEE, 2012.

[60] R. Groot and J. D. McLaughlin. *Geospatial data infrastructure: concepts, cases, and good practice*. Oxford University Press, 2000.

[61] J. Gudmundsson, A. Thom, and J. Vahrenhold. Of motifs and goals: Mining trajectory data. In *SIGSPATIAL*, pages 129–138. ACM, 2012.

[62] L. Guo, D. Zhang, G. Li, K.-L. Tan, and Z. Bao. Location-aware pub/sub system: When continuous moving queries meet dynamic event streams. In *SIGMOD*, pages 843–857, 2015.

[63] H. Gupta, B. Chawda, S. Negi, T. A. Faruquie, L. V. Subramaniam, and M. Mohania. Processing multi-way spatial joins on map-reduce. In *EDBT*, pages 113–124, 2013.

[64] R. H. Güting, T. Behr, and C. Düntgen. Secondo: A platform for moving objects database research and for publishing and integrating research implementations. *IEEE Data Eng. Bull.*, pages 56–63, 2010.

[65] R. H. Güting, T. de Almeida, and Z. Ding. Modeling and querying moving objects in networks. *The VLDB Journal*, 15(2):165–190, 2006.

[66] A. Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[67] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient indexing of spatiotemporal objects. In *EDBT*, pages 251–268. Springer, 2002.

[68] Hadoop. https://hadoop.apache.org/.

[69] HBase. https://hbase.apache.org/.

[70] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *VLDB*, 4(11):1111–1122, 2011.

[71] G. Hu, J. Shao, F. Liu, Y. Wang, and H. T. Shen. If-matching: Towards accurate map-matching with information fusion. *TKDE*, 29(1):114–127, 2017.

[72] J. Huang, S. Qiao, H. Yu, J. Qie, and C. Liu. Parallel map matching on massive vehicle gps data using mapreduce. In *International Conference on Embedded and Ubiquitous Computing, & International Conference on High Performance Computing and Communications*, pages 1498–1503. IEEE, 2013.

[73] P. Huang and B. Yuan. Mining massive-scale spatiotemporal trajectories in parallel: A survey. In *Trends and Applications in Knowledge Discovery and Data Mining*, pages 41–52. Springer, 2015.

[74] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *SIGOPS*, volume 41, pages 59–72. ACM, 2007.

[75] A. Javanmard, M. Haridasan, and L. Zhang. Multi-track map matching. In *SIGSPATIAL*, pages 394–397. ACM, 2012.

[76] R. Jinno, K. Seki, and K. Uehara. Parallel distributed trajectory pattern mining using mapreduce. In *International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 269–273. IEEE, 2012.

[77] J. Jo, Y. Tsunoda, B. Stantic, and A. W.-C. Liew. A likelihood-based data fusion model for the integration of multiple sensor data: A case study with vision and lidar sensors. pages 489–500, 2017.

[78] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *VLDB*, 1(2):1496–1499, 2008.

[79] E. J. Keogh and M. J. Pazzani. Scaling up dynamic time warping for datamining applications. In *SIGKDD*, pages 285–289. ACM, 2000.

[80] E. J. Keogh and M. J. Pazzani. Scaling up dynamic time warping for datamining applications. In *SIGKDD*, pages 285–289. ACM, 2000.

[81] S. Kim and J.-H. Kim. Adaptive fuzzy-network-based c-measure map-matching algorithm for car navigation system. *IEEE Transactions on Industrial Electronics*, 48(2):432–441, 2001.

[82] E. M. Knox and R. T. Ng. Algorithms for mining distancebased outliers in large datasets. In *VLDB*, pages 392–403, 1998.

[83] M. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *VLDB*, pages 840–851, 2004.

[84] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *VLDB*, 5(4):298–309, 2011.

[85] S.-L. Lee, S.-J. Chun, D.-H. Kim, J.-H. Lee, and C.-W. Chung. Similarity search for multidimensional data sequences. In *ICDE*, pages 599–608. IEEE, 2000.

[86] C. Li, Y. Gu, J. Qi, G. Yu, R. Zhang, and W. Yi. Processing moving k nn queries using influential neighbor sets. In *VLDB*, pages 113–124, 2014.

[87] D. Li, Y. Haitao, X. Zhou, and M. Gao. Map-reduce for calibrating massive bus trajectory data. In *International Conference on ITS Telecommunications*, pages 44–49. IEEE, 2013.

[88] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu. Distributed data management using mapreduce. *ACM Computing Surveys*, 46(3):31, 2014.

[89] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *ACM Symposium on Cloud Computing*, pages 1–15, 2014.

[90] Y. Li, Q. Huang, M. Kerber, L. Zhang, and L. Guibas. Large-scale joint map matching of gps traces. In *SIGSPATIAL*, pages 214–223. ACM, 2013.

[91] B. Lin and J. Su. Shapes based trajectory queries for moving objects. In *SIGSPATIAL*, pages 21–30. ACM, 2005.

[92] Y. Lou, C. Zhang, Y. Zheng, X. Xie, W. Wang, and Y. Huang. Map-matching for low-sampling-rate gps trajectories. In *SIGSPATIAL*, pages 352–361. ACM, 2009.

[93] J. Lu and R. H. Güting. Parallel secondo: a practical system for large-scale processing of moving objects. In *ICDE*, pages 1190–1193. IEEE, 2014.

[94] P. Lu, G. Chen, B. C. Ooi, H. T. Vo, and S. Wu. Scalagist: scalable generalized search trees for mapreduce systems [innovative systems paper]. *VLDB*, pages 1797–1808, 2014.

[95] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. In *VLDB*, pages 1016–1027, 2012.

[96] X. Lu, C. Wang, J.-M. Yang, Y. Pang, and L. Zhang. Photo2trip: generating travel routes from geo-tagged photos for trip planning. In *ACM Multimedia*, pages 143–152. ACM, 2010.

[97] W. Luo, H. Tan, L. Chen, and L. M. Ni. Finding time period-based most frequent path in big trajectory data. In *SIGMOD*, pages 713–724, 2013.

[98] Q. Ma, B. Yang, W. Qian, and A. Zhou. Query processing of massive trajectory data based on mapreduce. In *International workshop on Cloud data management*, pages 9–16. ACM, 2009.

[99] MLlib. http://spark.apache.org/docs/latest/mllib-guide.html.

[100] MongoDB. https://www.mongodb.com/.

[101] P. Newson and J. Krumm. Hidden markov map matching through noise and sparseness. In *SIGSPATIAL*, pages 336–343. ACM, 2009.

[102] J. Ni and C. V. Ravishankar. Indexing spatio-temporal trajectories with efficient polynomial approximations. *TKDE*, pages 663–678, 2007.

[103] S. Nishimura, S. Das, D. Agrawal, and A. El Abbadi. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In *International Conference on Mobile Data Management*, volume 1, pages 7–16. IEEE, 2011.

[104] S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan, and A. Ailamaki. Touch: In-memory spatial join by hierarchical data-oriented partitioning. In *SIGMOD*, pages 701–712. ACM, 2013.

[105] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, et al. Rethinking data-intensive science using scalable analytics systems. In *SIGMOD*, pages 631–646, 2015.

[106] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial tessellations: concepts and applications of Voronoi diagrams*, volume 501. John Wiley & Sons, 2009.

[107] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110. ACM, 2008.

[108] OpenStreetMap. https://www.openstreetmap.org/.

[109] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proceedings of the VLDB Endowment*, 3(1-2):928–939, 2010.

[110] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient olap operations in spatial data warehouses. In *International Symposium on Spatial and Temporal Databases*, pages 443–459. Springer, 2001.

[111] Y. Park, J.-K. Min, and K. Shim. Parallel computation of skyline and reverse skyline queries using mapreduce. *VLDB*, pages 2002–2013, 2013.

[112] Y. Park, J.-K. Min, and K. Shim. Processing of probabilistic skyline queries using mapreduce. *VLDB*, pages 1406–1417, 2015.

[113] J. M. Patel, Y. Chen, and V. P. Chakka. Stripes: an efficient index for predicted trajectories. In *SIGMOD*, pages 635–646. ACM, 2004.

[114] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178, 2009.

[115] D. A. Peixoto and N. Q. V. Hung. Scalable and fast top-k most similar trajectories search using mapreduce in-memory. In *Australasian Database Conference*, 2016.

[116] D. A. Peixoto, H. Su, N. Q. V. Hung, B. Stantic, B. Zheng, and X. Zhou. Concept for evaluation of techniques for trajectory distance measures. In *IEEE International Conference on Mobile Data Management (MDM)*, 2018.

[117] D. A. Peixoto, X. Zhou, N. Q. V. Hung, D. He, and B. Stantic. A system for spatial-temporal trajectory data integration and representation. In *International Conference on Database Systems for Advanced Applications (DASFAA)*, 2018.

[118] D. A. Peixoto, X. Zhou, N. Q. V. Hung, and B. Stantic. Workload-aware system for trajectory data storage and retrieval using spark. In *Submitted to International Conference on Very Large Data Bases (VLDB)*, 2018.

[119] D. A. Peixoto, X. Zhou, N. Q. V. Hung, and B. Zheng. A framework for parallel map-matching at scale using spark. In *Submitted to Distributed and Parallel Databases Journal (DAPD)*, 2017.

[120] N. Pelekis, I. Kopanakis, G. Marketos, I. Ntoutsi, G. Andrienko, and Y. Theodoridis. Similarity search in trajectory databases. In *International Symposium on Temporal Representation and Reasoning*, pages 129–140. IEEE, 2007.

[121] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. Gorilla: a fast, scalable, in-memory time series database. *VLDB*, 8(12):1816–1827, 2015.

[122] M. Perry and J. Herring. Ogc geosparql-a geographic query language for rdf data. *OGC Implementation Standard, ref: OGC*, 2011.

[123] D. Pfoser and C. Jensen. Capturing the uncertainty of moving-object representations. In *Advances in Spatial Databases*, pages 111–131. Springer, 1999.

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

[125] O. Pink and B. Hummel. A statistical approach to map matching using road network geometry, topology and vehicular motion constraints. In *International Conference on Intelligent Transportation Systems (ITSC)*, pages 862–867. IEEE, 2008.

[126] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, volume 10, pages 1–14, 2010.

[127] S. Puri, D. Agarwal, X. He, and S. K. Prasad. Mapreduce algorithms for gis polygonal overlay processing. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pages 1009–1016. IEEE, 2013.

[128] J. V. D. V. R. Jonkery, G. De Leve and A. Volgenant. Rounding symmetric travelling salesman problems with an asymmetric assignment problem. *Operations Research*, pages 623–627, 1980.

[129] A. Rajabifard and I. P. Williamson. Spatial data infrastructures: concept, sdi hierarchy and future directions. 2001.

[130] S. Ranu, P. Deepak, A. D. Telang, P. Deshpande, and S. Raghavan. Indexing and matching trajectories under inconsistent sampling rates. In *ICDE*, pages 999–1010, 2015.

[131] S. Rasetic, J. Sander, J. Elding, and M. A. Nascimento. A trajectory splitting model for efficient spatio-temporal indexing. In *VLDB*, pages 934–945, 2005.

[132] Y. Sakurai, M. Yoshikawa, and C. Faloutsos. Ftw: fast similarity search under the time warping distance. In *SIGMOD-SIGACT-SIGART*, pages 326–337. ACM, 2005.

[133] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. *Indexing the positions of continuously moving objects*, volume 29. ACM, 2000.

[134] M. Sharifzadeh and C. Shahabi. Vor-tree: R-trees with voronoi diagrams for efficient processing of spatial nearest neighbor queries. *VLDB*, pages 1231–1242, 2010.

[135] 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. In *VLDB*, pages 2110–2121, 2015.

[136] V. Sikka, F. Färber, A. Goel, and W. Lehner. Sap hana: The evolution from a modern main-memory data platform to an enterprise application platform. *VLDB*, 6(11):1184–1185, 2013.

[137] Z. Song and N. Roussopoulos. Seb-tree: An approach to index continuously moving objects. In *Mobile Data Management*, pages 340–344. Springer, 2003.

[138] S. Spaccapietra, C. Parent, M. L. Damiani, J. A. de Macedo, F. Porto, and C. Vangenot. A conceptual view on trajectories. *Data & Knowledge Engineering*, 65(1):126–146, 2008.

[139] Spark. https://spark.apache.org/.

[140] Spark-JobServer. https://github.com/spark-jobserver/spark-jobserver.

[141] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[142] M. Stonebraker and A. Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[143] Storm. https://storm.apache.org/.

[144] H. Su, K. Zheng, K. Zeng, J. Huang, S. Sadiq, N. J. Yuan, and X. Zhou. Making sense of trajectory data: A partition-and-summarization approach. In *ICDE*, pages 963–974. IEEE, 2015.

[145] H. Su, K. Zheng, K. Zeng, J. Huang, and X. Zhou. Stmaker–a system to make sense of trajectory data. *VLDB*, 7(13):1701–1704, 2014.

[146] H. Tan, W. Luo, and L. M. Ni. Clost: a hadoop-based storage system for big spatio-temporal data analytics. In *CIKM*, pages 2139–2143. ACM, 2012.

[147] Y. Tang, A. D. Zhu, and X. Xiao. An efficient algorithm for mapping vehicle trajectories onto road networks. In *SIGSPATIAL*, pages 601–604. ACM, 2012.

[148] Y. Tao, D. Papadias, and J. Sun. The tpr*-tree: an optimized spatio-temporal access method for predictive queries. In *VLDB*, pages 790–801, 2003.

[149] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive – a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005. IEEE, 2010.

[150] V. S. Tiwari, A. Arya, and S. Chaturvedi. Framework for horizontal scaling of map matching: Using map-reduce. In *International Conference on Information Technology*, pages 30–34. IEEE, 2014.

[151] S. Venkataraman, Z. Yang, E. L. Davies Liu, H. Falaki, X. Meng, R. Xin, A. Ghodsi, M. Franklin, I. Stoica, and M. Zaharia. Sparkr: Scaling r programs with spark. In *SIGMOD*. ACM, 2016.

[152] M. Vlachos, D. Gunopulos, and G. Das. Rotation invariant distance measures for trajectories. In *ACM SIGKDD*, 2004.

[153] M. Vlachos, D. Gunopulos, and G. Kollios. Discovering similar multidimensional trajectories. In R. Agrawal and K. R. Dittrich, editors, *ICDE*, pages 673–684, 2002.

[154] M. Vlachos, D. Gunopulos, and G. Kollios. Robust similarity measures for mobile object trajectories. In *Database and Expert Systems Applications*. IEEE, 2002.

[155] H. Vo, A. Aji, and F. Wang. Sato: A spatial data partitioning framework for scalable query processing. In *SIGSPATIAL*, pages 545–548. ACM, 2014.

[156] H. Wang, J. Li, Z. Hou, R. Fang, W. Mei, and J. Huang. Research on parallelized real-time map matching algorithm for massive gps data. *Cluster Computing*, pages 1–12, 2017.

[157] H. Wang, H. Su, K. Zheng, S. Sadiq, and X. Zhou. An effectiveness study on trajectory similarity measures. In *Australasian Database Conference*, pages 13–22. Australian Computer Society, 2013.

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

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

[160] K. Wang, J. Han, B. Tu, J. Dai, W. Zhou, and X. Song. Accelerating spatial data processing with mapreduce. In *International Conference on Parallel and Distributed Systems (ICPADS)*, pages 229–236. IEEE, 2010.

[161] L. Wang, Y. Zheng, X. Xie, and W.-Y. Ma. A flexible spatio-temporal indexing scheme for large-scale gps track retrieval. In *MDM*, pages 1–8. IEEE, 2008.

[162] X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann, and E. Keogh. Experimental comparison of representation methods and distance measures for time series data. *Data Mining and Knowledge Discovery*, 26(2):275–309, 2013.

[163] X. Wang, X. Zhou, and S. Lu. Spatiotemporal data modelling and management: a survey. In *TOOLS-Asia*, pages 202–211. IEEE, 2000.

[164] H. Wei, Y. Wang, G. Forman, Y. Zhu, and H. Guan. Fast viterbi map matching with tunable weight functions. In *SIGSPATIAL*, pages 613–616. ACM, 2012.

[165] C. Wenk, R. Salas, and D. Pfoser. Addressing the need for map-matching speed: Localizing global curve-matching algorithms. In *International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 379–388. IEEE, 2006.

[166] E. Wu and S. Madden. Partitioning techniques for fine-grained indexing. In *ICDE*, pages 1127–1138, 2011.

[167] Y. Xia, Y. Liu, Z. Ye, W. Wu, and M. Zhu. Quadtree-based domain decomposition for parallel map-matching on gps data. In *International Conference on Intelligent Transportation Systems (ITSC)*, pages 808–813. IEEE, 2012.

[168] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *SIGMOD*. ACM, 2016.

[169] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[170] Y. Xu, P. Kostamaa, and L. Gao. Integrating hadoop and parallel dbms. In *SIGMOD*, pages 969–974, 2010.

[171] Z. Yan, S. Spaccapietra, et al. Towards semantic trajectory data analysis: A conceptual and computational approach. In *VLDB*, 2009.

[172] B. Yang, Q. Ma, W. Qian, and A. Zhou. Truster: Trajectory data processing on clusters. In *DASFAA*, pages 768–771, 2009.

[173] B.-K. Yi, H. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE*, pages 201–208. IEEE, 1998.

[174] S. You, J. Zhang, and L. Gruenwald. Large-scale spatial join query processing in cloud. In *ICDE Workshops*, pages 34–41. IEEE, 2015.

[175] J. Yu, J. Wu, and M. Sarwat. Geospark: A cluster computing framework for processing large-scale spatial data. In *SIGSPATIAL*. ACM, 2015.

[176] J. Yuan, Y. Zheng, X. Xie, and G. Sun. Driving with knowledge from the physical world. In *SIGKDD*, pages 316–324, 2011.

[177] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. T-drive: driving directions based on taxi trajectories. In *SIGSPATIAL*, pages 99–108. ACM, 2010.

[178] M. Yuan, K. Deng, J. Zeng, Y. Li, B. Ni, X. He, F. Wang, W. Dai, and Q. Yang. Oceanst: a distributed analytic system for large-scale spatiotemporal mobile broadband data. *VLDB*, 7(13):1561–1564, 2014.

[179] Y. Yuan and M. Raubal. Measuring similarity of mobile phone user trajectories–a spatio-temporal edit distance method. *International Journal of Geographical Information Science*, 2014.

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

[181] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *USENIX Conf. on Hot Topics in Cloud Comp.*, page 10, 2010.

[182] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SIGOPS*, pages 423–438, 2013.

[183] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *USENIX Conference on Hot Topics in Cloud Computing*. USENIX Association, 2012.

[184] E. Zamanian, C. Binnig, and A. Salama. Locality-aware partitioning in parallel database systems. In *SIGMOD*, pages 17–30, 2015.

[185] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica. G-ola: Generalized on-line aggregation for interactive analysis on big data. In *SIGMOD*, pages 913–918, 2015.

[186] C. Zhang, F. Li, and J. Jestes. Efficient parallel knn joins for large data in mapreduce. In *EDBT*, pages 38–49, 2012.

[187] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *TKDE*, 27(7):1920–1948, 2015.

[188] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng. Spatial queries evaluation with mapreduce. In *International Conference on Grid and Cooperative Computing*, pages 287–292. IEEE, 2009.

[189] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu. Sjmr: Parallelizing spatial join with mapreduce on clusters. In *International Conference on Cluster Computing (CLUSTER)*, pages 1–8. IEEE, 2009.

[190] B. Zheng, N. J. Yuan, K. Zheng, X. Xie, S. Sadiq, and X. Zhou. Approximate keyword search in semantic trajectory database. In *ICDE*, pages 975–986, 2015.

[191] K. Zheng, P. C. Fung, and X. Zhou. K-nearest neighbor search for fuzzy objects. In *SIGMOD*, pages 699–710. ACM, 2010.

[192] K. Zheng, H. Su, B. Zheng, S. Shang, J. Xu, J. Liu, and X. Zhou. Interactive top-k spatial keyword queries. In *ICDE*, pages 423–434, 2015.

[193] K. Zheng, Y. Zheng, X. Xie, and X. Zhou. Reducing uncertainty of low-sampling-rate trajectories. In *ICDE*, pages 1144–1155. IEEE, 2012.

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

[195] Y. Zheng. Trajectory data mining: an overview. *ACM Transactions on Intelligent Systems and Technology (TIST)*, page 29, 2015.

[196] Y. Zheng, Q. Li, Y. Chen, X. Xie, and W.-Y. Ma. Understanding mobility based on gps data. In *International Conference on Ubiquitous Computing*, pages 312–321. ACM, 2008.

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

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

[199] Y. Zheng and X. Zhou. *Computing with spatial trajectories*. Springer Science & Business Media, 2011.

[200] Y. Zhong, J. Han, T. Zhang, Z. Li, J. Fang, and G. Chen. Towards parallel spatial query processing for big spatial data. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 2085–2094. IEEE, 2012.

[201] X. Zhou, D. J. Abel, and D. Truffet. Data partitioning for parallel spatial join processing. In *Advances in Spatial Databases*, pages 178–196. Springer, 1997.