University of Padova

Department of Mathematics "Tullio Levi-Civita"

Master Thesis in Erasmus Mundus Joint Master's Degree in Big Data Management & Analytics

MobilitySpark: Big Mobility Data Management with PySpark and PyMEOS

Supervisor
Prof. Geppino Pucci
University of Padova

Co-supervisor
Prof. Esteban Zimányi
Université Libre de Bruxelles

Master Candidate
Luis Alfredo León Villapún

Student ID
2105044

Academic Year
2023-2024

"With what shall I go? Will I leave nothing behind on this earth? How should my heart act? Did we come here for nothing, to sprout upon the earth? Let us at least leave flowers, Let us at least leave songs."

— Nezahualcóyotl, poet and tlatoani of Texcoco

Abstract

Mobility data management has become increasingly significant with technologies requiring precise and optimal spatiotemporal data processing. In the context of Big Data, distributed processing is essential. This thesis proposes the implementation of a binding for the Mobility Engine, Open Source (MEOS) library using PySpark, named MobilitySpark. The developed tool comprises four main modules: User-Defined Types (UDT), User-Defined Functions (UDF), User-Defined Table Functions (UDTF), and Partitioning, alongside an additional Utilities module. The Partitioning module includes several techniques for mobility data partitioning, specifically Regular Grid, KD-Tree, Adaptive Bins, and Bisective K-Means. Both in-memory and Spark-based implementations are provided for KD-Tree and Adaptive Bins. Additionally, a MapReduce-based approach, Approximate Adaptive Bins, is introduced. Each partitioning technique is evaluated in a distributed environment using data from OpenSky and BerlinMOD datasets.

The experimental results demonstrate that a balance between processing time and load distribution is achieved with the Approximate Adaptive Bins and the Spark-based implementation of Adaptive Bins. For scenarios requiring optimal load balancing, the KD-Tree implementation is recommended. Whenever a simple partitioning is needed, the Regular Grid may be used. Moreover, the thesis shows that using these grid-based partitioners as an artificial index significantly optimizes mobility query performance, as evidenced by the BerlinMOD dataset adapted to the city of Brussels.

Keywords: *Mobility Data, MEOS, Spark, Partitioning, MobilityDB, Spatiotemporal*.

Contents

Listing of figures

Listing of tables

Listing of acronyms

API . . . . . Application Programming Interface
CAP . . . . . Consistency, Availability, Partition Tolerance
DBMS . . . . . Database Management System
DMS . . . . . Data Management System
GFS . . . . . Google File System
HDFS . . . . . Hadoop Distributed File System
IoT . . . . . Internet of Things
JVM . . . . . Java Virtual Machine
MDBMS . . . . . Mobility Database Management System
MEOS . . . . . Mobility Engine Open Source
OOP . . . . . Object Oriented Programming
PIP . . . . . Point-in-Polygon
RDD . . . . . Resilient Distributed Dataset
UDT . . . . . User-defined Type
UDF . . . . . User-defined Function
UDTF . . . . . User-defined Table Function

1 Introduction

We cannot change the wind, but we can adjust the sails differently.
— Aristotle

1.1 A World of Data and Mobility

The increasing demand for robust big data frameworks requires the development and utilization of systems capable of efficiently processing specific data types. These systems aim to enhance the transformation and enrichment of data into meaningful insights. Mobility data emerges as a critical element in this context, significantly contributing to the advancement of smart cities, energy conservation, and improved transportation monitoring.

Concurrently, the evolution of big data analytics systems has highlighted the importance of an effective distribution strategy. Such a strategy is crucial for ensuring the fulfillment of two of the CAP theorem's three principles (consistency, availability, partition tolerance), depending on the system's nature. The necessity for a fast and reliable system has accelerated a shift in perspective from a purely software-centric approach to a data-centric one. This change underscores the impact of data distribution on the system's overall performance and the quality of the analysis, affecting factors such as speed and accuracy.

Adopting a data-centric approach needs considering requirements to each specific use case, acknowledging that a solution optimal for one scenario may not suit another. While there is value in generalizing these systems to an extent, they often require a degree of customization to achieve peak performance. In the context of mobility data, this involves considering the unique characteristics of moving objects, or *spatiotemporal* entities, which undergo changes over time [3, 4].

Trajectory analysis presents a particularly intriguing challenge within mobility data. This analysis can range from individual to *aggregated* crowd levels [5]. Data aggregation inherently involves operations that can quickly become complex in any database or analytics tool if not managed carefully. Moreover, in distributed systems,

the overhead of data transmission between nodes must be considered. Efficiently distributing trajectory data is therefore crucial to significantly reduce data movement across nodes.

Recent initiatives like MobilityDB [6] and MEOS [2] address the challenges associated with the storage, processing, and analysis of mobility data within a specialized framework. Functioning as an extension to PostgreSQL and PostGIS, MobilityDB introduces data types and functions specifically designed for handling mobility data, offering numerous benefits. These advantages encompass not only a significant acceleration in query processing times and a reduction in storage requirements but also a more streamlined process for defining queries at the user level, enhancing user experience and efficiency.

Building upon the foundation laid by MobilityDB, further advancements are explored in more recent contributions, such as [7, 8]. These studies propose and illustrate the expansion of MobilityDB into a distributed setting through the integration with Citus, an extension that distributes PostgreSQL databases. This extension is examined in subsequent sections of this thesis, highlighting its potential to enhance data management in distributed environments.

However, before diving into MobilityDB, one must review what is referred to as *mobility data*, its components, and structure. Understanding the intricacies of mobility data is crucial, as it plays an important role in diverse applications, including urban planning, traffic management, logistics optimization, and even public health monitoring. Furthermore, mobility data addresses several critical challenges, such as ensuring data accuracy, handling large volumes of data efficiently, and providing insights into movement patterns and behaviors. By exploring these aspects, we can better appreciate the value and complexities of managing and analyzing mobility data within frameworks like MobilityDB.

1.2 Brief Introduction to Mobility Data

The exploration of dynamic changes within various systems has long captivated human curiosity. Especially over the past century, technological advancements have empowered societies to delve into the geographical positioning of objects not merely from a static viewpoint but through a dynamic lens as well. This interdisciplinary endeavor, focusing on the *spatiotemporal* tracking of objects and their evolving attributes, is encapsulated in what we term *mobility data*.

Mobility data stands as a multifaceted and intricate type of data, offering insights into the temporal and spatial progression of objects, systems, and individuals. Its utility spans a diverse array of domains; for instance, it plays a crucial role in urban planning by enabling a nuanced analysis of city traffic systems. Similarly, in aviation, mobility data facilitates the optimization of flight paths, enhancing efficiency from one airport to another. In the arising field of the Internet of Things (IoT), devices such as smartwatches serve as rich data repositories, capturing detailed information about individual and collective behaviors. These devices, by correlating sensor data with specific times and locations, open avenues for analyzing and predicting human activity patterns, ranging from identifying periods of calm or heightened activity to optimizing public transportation stops and schedules based on collective mobility trends.

The complexity of mobility data cannot be overstated. It is subject to constant evolution, influenced by changes in systems, modifications to geographical maps, shifts in political boundaries, and the inexorable march of time. The addition of a temporal dimension to mobility data significantly enriches its analytical potential, allowing for

a comprehensive examination of multifaceted entities. Consider, for example, a bus integrated into a smart city network. Such a scenario presents a myriad of variables for analysis, including velocity, battery or gas indicators, passenger statistics, etc. A specific area of interest could be to chart the velocity of the bus across both time and space, providing valuable insights into urban mobility and infrastructure efficacy.

That being said, the spatiotemporal domain of mobility data is often conceptualized within a three-dimensional (*x, y, and time*) or four-dimensional (*x, y, z, and time*) framework. However, such a representation risks oversimplifying the complexity of mobility data, reducing it to a mere vectorial form. Mobility datasets encompass far more than just geometric information; they include semantic aspects that are crucial for interpreting each data point. For instance, questions like "Is the airplane above the 10,000 feet threshold?" can be addressed more effectively by incorporating enhanced data types that more accurately manage and interpret such contextual information. For instance, an enriched mobility data framework would provide what is known as a temporal boolean to answer the airplane question, as it will be reviewed in further chapters.

Therefore, mobility data can be defined as a domain that comprises multidimensional vectors, which include spatial dimensions and a temporal component, each imbued with specific semantic values that vary across data points. As explored in greater depth throughout this thesis, it is crucial to manage this data in a smart and efficient manner in order to fully exploit the valuable insights this fascinating domain can provide.

1.3 Challenges in Mobility Data Management

In the previous subsection, we emphasized the necessity of viewing mobility data not merely as a vector field but as a rich source of semantics that can be extracted and queried spatially and temporally. The fundamental objective of any data science endeavor is to convert raw data into insightful information, and the treatment of mobility data is no exception. Accordingly, it is imperative that mobility data is approached with the same rigor and thoroughness as any other data domain. This entails not only extracting valuable insights from raw datasets but also managing these transformations and their outcomes effectively.

From a data science perspective, mobility data provides insights into patterns of movement and usage across different spatial and temporal scales. Such data allows for the exploration of complex questions, such as the impact of urban planning on traffic congestion or the effectiveness of public transport systems over time. Moreover, with the incorporation of spatiotemporal awareness, predictive models can be significantly enhanced. For instance, by understanding the dynamics of location and time, models can predict future traffic conditions, optimize routes in real time, and foresee potential areas of congestion before they occur. This spatiotemporal dimension not only enriches the predictive accuracy but also tailors the applications to be more responsive to real-world scenarios.

From the standpoint of data management and engineering, choosing the right logical or conceptual model is critical. A model that supports the intrinsic spatial and temporal characteristics of mobility data, is often more suitable. Efficient storage of mobility data, while aligning with the chosen framework, can be achieved through techniques like data partitioning, which organizes data in a way that aligns with its most frequent queries, thus enhancing retrieval speeds. Furthermore, the ability to quickly access, query, and read mobility data is a priority. Using indexing strategies that cater to both spatial and temporal queries, like geospatial indexing, can dramatically reduce the time needed to retrieve relevant data, facilitating faster decision-making and real-time data analysis.

Another critical consideration, especially when dealing with large volumes of data, is the selection of an ap-

propriate Database Management System (DBMS) for mobility data. In modern setups, where data distribution across multiple nodes is common, it is essential to study and optimize this distribution before deploying any solution. Here, partitioning strategies become crucial as they determine how data is distributed across clusters, directly impacting the performance of queries, data science tasks, and pipeline implementations within the framework.

Based on the previous points, it becomes evident that choosing the right data model and DBMS, along with selecting effective data distribution techniques, are essential for the optimal implementation of mobility data use cases. These choices influence not only the efficiency of data processing but also the scalability and responsiveness of the entire system, ensuring that mobility data is leveraged to its full potential.

1.4 Research Objectives

The principal aim of this research is to implement and evaluate effective data partitioning methodologies tailored for mobility data, with a special focus on trajectory data analysis. The evaluation will concentrate on how each partitioning approach optimizes different mobility data tasks, such as query execution, and the preprocessing phase of trajectory analysis operations, including the handling of aggregated data [5]. In particular, the objective is to evaluate these strategies under the Spark environment, using the Mobility Engine Open Source (MEOS) framework to handle Mobility Data in a distributed scenario. Taking this into consideration, the research objectives are now outlined below.

1.4.1 Implementing a MEOS Binding Using Spark

The primary objective of this research is to develop a binding for the Mobility Engine Open Source[1] (MEOS) using Spark, aiming to enhance the processing capabilities and scalability of mobility data analysis. Spark, a framework for handling large-scale data processing, is chosen for its ability to perform complex data transformations and analyses efficiently across distributed systems. By integrating Spark with MEOS, this binding seeks to leverage the robust analytical features of Spark and its seamless scalability to improve the handling, querying, and real-time processing of spatiotemporal mobility data. This enhancement is expected to significantly advance the performance of MEOS in large-scale environments, facilitating more dynamic and responsive mobility data applications. The election of which Spark API to use (Java, Scala, Python, R), as well as which MEOS subfamily to pick (MEOS, JMEOS, PyMEOS), will be of critical importance for this thesis.

1.4.2 Identifying Trajectory Data Partitioning Strategies

Explore and assess existing data partitioning strategies that are particularly suited to handling trajectory data. Given the unique spatiotemporal characteristics of trajectory data, conventional partitioning methodologies often fall short, especially in complex operations like joins, which are essential for the synthesis and analysis of movement across various datasets. The primary goal is to implement partitioning strategies that achieve a more balanced distribution of data across computing nodes, thus improving the computational efficiency of fundamental operations.

[1] `https://www.libmeos.org/`

These strategies will specifically target enhancements in data access patterns and computational load balancing, aiming to significantly boost processing speed and overall efficiency in handling large-scale trajectory data. The findings will contribute to better scalability and operational responsiveness, advancing big data analytics practices tailored to mobility data.

1.4.3 Adaptation of a Spark Environment to Distribute Trajectory Data

This research objective focuses on customizing Apache Spark environments to optimize the distribution of trajectory data, leveraging Spark's capabilities as a leading platform in big data analytics [2]. The objective is to implement a specialized distribution strategy that addresses the spatial and temporal aspects inherent to trajectory data. This project will compare various distribution strategies within the Spark framework, aiming to identify and implement the most effective approach for managing and analyzing large datasets of trajectory data. Special attention will be given to the scalability, fault tolerance, and real-time processing capabilities of these strategies, with the aim of enhancing the performance and operational efficiency of Spark-based applications. This comprehensive evaluation will provide critical insights, supporting the development of advanced applications in areas like urban planning and transportation logistics, thus enhancing the dynamic use of trajectory data in mobility-related fields.

1.4.4 Demonstrate How a Partitioning Strategy Impacts Data Science Tasks

This research objective is centered on evaluating the impact of various trajectory data partitioning strategies on different data science tasks across multiple use cases. The aim is to systematically assess how these strategies influence the performance and outcomes at various stages of the data science lifecycle, from data ingestion and preprocessing to analysis and modeling. Special emphasis will be placed on quantifying improvements in data accessibility, processing speed, and analytical accuracy. Additionally, the project will explore how enhanced partitioning strategies can facilitate more complex analytical tasks such as predictive modeling and real-time data streaming. By identifying which aspects of the data science workflow are most improved by optimal partitioning, this research will provide actionable insights that can lead to substantial enhancements in the efficiency and effectiveness of trajectory data analysis. This in turn could support advancements in sectors reliant on dynamic data interpretation, such as traffic management, urban planning, and location-based services.

1.5 Thesis Structure

This section outlines the organizational framework of the thesis, offering a detailed overview of each chapter to clarify the objectives and subjects tackled. The thesis is dedicated to exploring key considerations in developing

[2] `https://spark.apache.org/`

an optimal strategy for analyzing trajectory data within a distributed computing environment. The overall structure of the thesis is designed to guide the reader through a systematic exploration of this topic, as detailed in the following chapters:

In *Chapter 2*, we introduce the essential definitions and concepts necessary to understand the mobility data model utilized in this thesis. This chapter begins with an exploration of the model and definitions associated with mobility data, setting the stage for subsequent discussions. It then delves into a comprehensive review of the MEOS framework, discussing its architecture, capabilities, and limitations. This is followed by **Section 2.2**, which explores various data partitioning techniques suitable for spatiotemporal data. This section starts with a discussion of classic partitioning techniques, providing a foundational understanding of traditional methods used in data management systems. It progresses to examine spatiotemporal partitioning techniques like Regular Grid Partitioning, KD Tree methods, and Adaptive Bins Tiling, evaluating each for its effectiveness in enhancing computational efficiency and scalability for large volumes of trajectory data.

Next, *Section 2.3* focuses on mobility data management systems, particularly in distributed environments handling large-scale spatiotemporal data. It begins with a review of distributed relational algebra to lay the theoretical groundwork for understanding complex data operations across distributed systems. This section transitions into a detailed examination of MobilityDB and other related database management systems, highlighting the functionalities and roles they play in managing mobility data. The text advances to discuss the state-of-the-art Distributed Mobility DB and its advantages. Next, we explore other mobility big data solutions that are implemented within Hadoop and Spark. Special attention is given to examining big data strategies and native join strategies in Spark, as these are critical for managing large-scale data efficiently. This analysis aims to assess the performance and scalability of various big data solutions in the context of mobility data management, thereby facilitating innovative approaches for processing and analyzing massive datasets.

Chapter 3 presents a detailed exploration of the solutions developed to address the core challenges discussed in the thesis. This chapter extends the capabilities of PyMEOS, a Python library for moving object data, into the PySpark environment, thus enhancing its functionality within distributed computing frameworks. The integration aims to facilitate more efficient processing and analysis of large-scale trajectory data by leveraging Spark's robust data handling and processing capabilities. Additionally, the chapter details the implementation of a data partitioning scheme specifically optimized for trajectory data analysis. A practical application of these methodologies is demonstrated through the BerlinMOD scenario, a benchmarking framework that simulates realistic urban mobility patterns. This use case serves to illustrate the effectiveness and practical impact of the proposed solutions in a real-world setting, showcasing improvements in processing speeds and analytical precision.

Chapter 4 summarizes the significant findings and contributions of the thesis, reflecting on the enhanced capabilities and performance improvements brought about by the integration of advanced data partitioning strategies and the extension of PyMEOS into PySpark. The chapter discusses the broader implications of these advancements for the field of mobility data analysis and distributed data systems. Looking forward, it proposes several avenues for future research, including potential improvements to the partitioning algorithms, further integration with other big data technologies, and the exploration of new use cases in different domains. The conclusion underscores the necessity for ongoing innovation and adaptation in data management strategies to keep pace with the evolving demands of big data analytics.

2

Mobility Data

If you have knowledge, let others light their candles in it.
— Margaret Fuller

2.1 Mobility Data

As briefly discussed in the introductory section, mobility data is a critical domain and a powerful tool that supports various applications, encompassing everything from smart cities and the Internet of Things (IoT) to transportation networks and animal tracking systems. This data, rich with spatiotemporal attributes, plays a pivotal role in optimizing traffic flow, enhancing public safety, monitoring environmental conditions, and studying wildlife migration patterns. Its unique characteristics allow stakeholders to make informed decisions based on real-time and historical movement patterns, thereby significantly contributing to advancements in these diverse fields.

This chapter offers a formal exploration of the mobility data domain. It begins with a thorough review of the data model employed in this thesis, detailing the structural elements and explaining key data types that are crucial for understanding the complex implementations discussed in subsequent sections. This is followed by an introduction to the MEOS library, which serves as a foundational component for handling mobility data across different programming languages. Each binding's unique features are discussed to illustrate how MEOS can be integrated and utilized in various computational environments. The chapter concludes with a critical comparison of different mobility data solution implementations, assessing their respective features and capabilities. This analysis not only highlights the functional differences and advantages of each implementation but also sheds light on how these can be leveraged to enhance data processing and analysis in real-world applications.

2.1.1 Introductory Definitions

Let us start by introducing the basic definitions for what is understood as mobility data and trajectory, as these concepts are the core to understanding the domain of mobility data science. These definitions provide a framework that helps in analyzing and interpreting the movement patterns of various entities.

Definition 2.1.1 (Mobility Data)**.** A dataset D that details the spatiotemporal trajectories of entities. Essentially, this data includes at least one *temporal* dimension and a set of *spatial* dimensions, typically represented as coordinates (x, y) or (x, y, z). Commonly, mobility data is generated through tracking devices that record location points, each tagged with a timestamp [9].

Definition 2.1.2 (Trajectory)**.** As defined in [4], represents the path traced by an entity's movement through a specific space S during a time interval $[t_{begin}, t_{end}]$. This path documents the continuous evolution or movement of the entity over time.

These initial definitions of mobility data and trajectory are intrinsically linked. Mobility data comprises one or more trajectories. A significant challenge arises in treating these trajectories as continuous functions. Although theoretically continuous, trajectories are practically captured by discretizing the path into a series of points.

Definition 2.1.3 (Discrete Trajectory)**.** Finite set of spatiotemporal points that approximates the trajectory within a specified temporal interval $[t_{start}, t_{end}]$. These points capture snapshots of the trajectory's evolution, but they do not convey the full continuous nature of the trajectory itself.

The process of capturing trajectory mobility data can be illustrated with a practical example. Consider a scenario where scientists equip a group of geese with GPS devices to study their flight patterns[1]. It is impractical to record GPS locations too frequently due to resource constraints, yet infrequent updates risk losing the accuracy needed to approximate the continuous trajectory accurately. As a compromise, the GPS devices are configured to record the location every 5 minutes. The resulting data, sent back to the scientists, might look something like Table 2.1.

Table 2.1 presents the dataset received by the scientists in the aforementioned example. This dataset exemplifies mobility data, as it incorporates a spatiotemporal component, featuring three distinct trajectories—one for each tracked animal. Notably, the spatial component of this data comprises three dimensions: *latitude*, *longitude*, and *altitude*. The geographic space, S, that these coordinates map is defined within the boundaries formed by the coordinate extremes: $[lat_{min}, lon_{min}]$ and $[lat_{max}, lon_{max}]$. The temporal space, T, that this table maps is defined by the range formed by the maximum and minimum timestamps t: $[t_{max}, t_{min}]$.

From a preliminary viewpoint, this data setup allows scientists to accurately trace the locations of each goose at any given timestamp, facilitating detailed behavioral studies. However, from a data management perspective, there are additional considerations to address. If the dataset grows significantly large, managing it on a point-by-point basis may become impractical. Therefore, it could be advantageous to develop an enhanced data model that manages data at the trajectory level. Such a model would not only streamline data handling but also enable the integration of additional relevant features into the dataset. This approach and its potential benefits will be

[1] This example can be followed in the `DuckExample.ipynb` Jupyter notebook in the project repository, see Chapter 3.

Timestamp	AnimalID	Latitude	Longitude	Altitude (m)
2024-06-01 08:00:00	A1	52.3761	4.9089	15
2024-06-01 08:00:00	A2	52.3700	4.9070	12
2024-06-01 08:00:00	A3	52.3650	4.9050	10
2024-06-01 08:05:00	A1	52.3780	4.9095	20
2024-06-01 08:05:00	A2	52.3720	4.9075	15
2024-06-01 08:05:00	A3	52.3670	4.9055	14
2024-06-01 08:10:00	A1	52.3798	4.9101	18
2024-06-01 08:10:00	A2	52.3740	4.9080	16
2024-06-01 08:10:00	A3	52.3690	4.9060	12

Table 2.1: Example of mobility data.

thoroughly explored in the subsequent section, aiming to equip scientists with more effective tools for analyzing complex mobility data.

2.1.2 Data Model

In developing the data model for this thesis, we will adhere closely to the structural frameworks established in [10] and [4], as well as the practical implementations observed in MobilityDB and MEOS [2]. These frameworks are not only foundational but also heavily influence the architecture of contemporary mobility data systems.

The necessity for specialized mobility data types stems from the complex requirements of encoding *basic* data types (such as integers, strings, booleans, and floats) within a *spatiotemporal* context. This model integrates *temporal* types (e.g., instants and spans) with *spatial* types (points, geometries, boxes) to form a comprehensive mobility data schema. The design of these data types is crucial as it supports transformations, aggregations, and analytical functions, enhancing the data's utility in various applications. When implemented within an Object-Oriented paradigm, these data types benefit from the principles of inheritance and abstraction, significantly improving usability and flexibility for end-users. This structured approach ensures that the mobility data model is both robust and adaptable. Now, the *entities* which compose a trajectory can be *points*, *lines*, or *regions* that we denote as *geometries*. Naturally, these are spatial data types that typically are described in euclidean terms.

Definition 2.1.4 (Instant)**.** A data type representing a specific point in time $t \in \mathbb{R}$, where $\mathbb{R}$ denotes the set of real numbers.

Definition 2.1.5 (Span)**.** Continuous interval of time, represented by two instants i_0 and i_1, where $i_0, i_1 \in \mathbb{R}$ and $i_0 < i_1$. The span can be denoted as $S = [i_0, i_1]$.

Definition 2.1.6 (Spatial)**.** Data type representing an object or phenomenon with a defined location in a given space. This can be defined in a coordinate system $\mathbb{R}^n$, where n is the number of dimensions, typically $n = 2$ for 2D space (latitude, longitude) and $n = 3$ for 3D space (latitude, longitude, altitude).

Definition 2.1.7 (Temporal)**.** Data type describing the variation or evolution of a value v over time t. The value v can be any basic data type (such as integer, float) or a spatial data type, and the temporal data type captures how v changes with t, typically expressed as a function $v(t)$.

Having temporal data types is useful in the context of mobility data, as it allows to understand the characteristics of data under different time or spatial-restricted bounds. It is useful to also derive three subclasses for the `Temporal` data type: `TemporalInstant`, `TemporalSequence`, `TemporalSequenceSet`.

Definition 2.1.8 (TemporalInstant)**.** Data type representing a value v observed at a specific point in time $t \in \mathbb{R}$. Formally, a TemporalInstant can be denoted as a tuple (t, v), where t indicates the time of the observation and v denotes the value of the temporal attribute at that instant.

Definition 2.1.9 (TemporalSequence)**.** Data type representing an ordered collection of `TemporalInstant`, (t_i, v_i), where $i = 0, 1, \ldots, n$ and $t_0 < t_1 < \ldots < t_n$. This sequence defines the evolution of a value over a continuous time interval $[t_0, t_n]$, capturing the changes in the value v as a function of time t.

Definition 2.1.10 (TemporalSequenceSet)**.** Data type comprising a set of `TemporalSequence`, $\{S_i\}$, where each S_i represents a distinct `TemporalSequence` defined over its own time interval $[t_{0_i}, t_{n_i}]$. This set can include sequences that are disjoint, overlapping, or contiguous, thus allowing for the representation of multiple, potentially non-continuous, segments of temporal data.

A trajectory, for instance, can be understood in terms of a collection of `TemporalInstant`, a `TemporalSequence`, or a `TemporalSequenceSet`. In Chapter 3, trajectories are transformed from sequences of instants into `TemporalSequence` or `TemporalSequenceSet` objects, for example.

Another important family of data types that are important for a mobility data model include the concept of *boxes*. Boxes can be spatial or spatiotemporal.

Definition 2.1.11 (Box)**.** Multi-dimensional geometric object characterized by a range $[r_{\min}, r_{\max}]$ for each dimension D. Formally, a `Box` in n-dimensional space can be represented as a set of intervals $\{[r_{\min_i}, r_{\max_i}]\}_{i=1}^{n}$, where $r_{\min_i}$ and $r_{\max_i}$ denote the minimum and maximum bounds of the box along the i-th dimension, respectively.

Definition 2.1.12 (Spatiotemporal Box)**.** Extends the concept of a box to include both spatial and temporal dimensions. It is defined by a set of ranges $\{[r_{\min_i}, r_{\max_i}]\}$, where i spans the spatial dimensions (e.g., x, y, z) and the temporal dimension t. It can be described as $\{[r_{\min_x}, r_{\max_x}], [r_{\min_y}, r_{\max_y}], [r_{\min_z}, r_{\max_z}], [t_{\min}, t_{\max}]\}$, encapsulating the spatial extent and the time interval over which the box exists.

Figure 2.1 shows three of the concepts reviewed in this section. Note how a trajectory is continous and therefore, it is difficult to describe in computational terms. A discrete trajectory can be defined, though, more easily with the use of temporal types. Finally, a spatiotemporal box is shown as well under the three spatiotemporal dimensions x, y, and t.

In the context of this project, and particularly pertaining the partitioning of data, it is important to have topological functions which describe the interaction between spatiotemporal objects. This thesis makes a wide use of the `ever_intersects`, `at`, and `contains` functions. Chapter 3 shows how these functions are used mostly to determine if a trajectory, often represented as a `TemporalSequence` object, is `at`, `intersects`, or `contains` another trajectory or another spatiotemporal box.

Definition 2.1.13 (Function `ever_intersects`)**.** Determines whether two spatial or temporal objects A and B intersect at any point in their respective domains. Let $A(t)$ and $B(t)$ denote the state of objects A and B at time

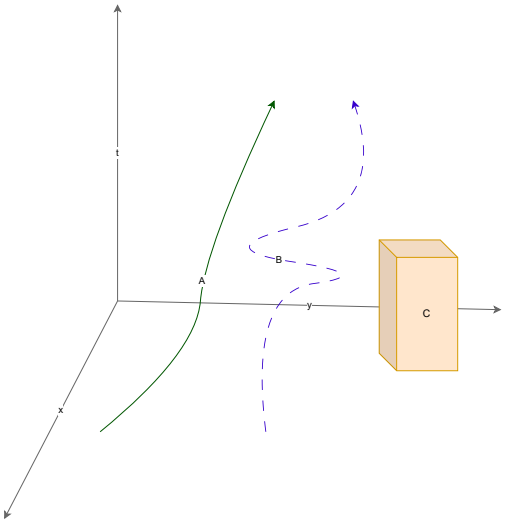

Figure 2.1: Common mobility data types. A trajectory (A), a segmented discrete trajectory (B), and a spatiotemporal box (C).

$t \in \mathbb{R}$ for temporal objects, or their respective spatial representations for spatial objects. The function returns *True* if there exists a time $t \in \mathbb{R}$ such that $A(t) \cap B(t) \neq \emptyset$, indicating a non-empty intersection, and *False* otherwise.

Definition 2.1.14 (Function `at`)**.** Restricts the spatial or temporal domain of object A to the portion where it intersects with object B. Let $A(t)$ and $B(t)$ represent the state of A and B at time t for temporal objects, or their respective spatial components for spatial objects. The function returns $A \cap B$, which includes all elements x in A such that $x \in A$ and $x \in B$. In the temporal domain, this is equivalent to finding $A \cap B$ over all t where $A(t)$ and $B(t)$ are defined and intersect.

Definition 2.1.15 (Function `contains`)**.** Verifies whether the entire spatial or temporal extent of object B is contained within object A. For all x in B, if $x \in B \implies x \in A$, then A is said to contain B, and the function returns *True*. Otherwise, it returns *False*. This is expressed as $B \subseteq A$, meaning every point or interval defining B lies entirely within the bounds of A.

2.1.3 The MEOS Family

The Mobility Engine Open Source (MEOS)[2] C library, as cited in [2], is an implementation that adheres to the *ISO 19141:2008* standard, titled *Geographic information — Schema for moving features*. This standard provides a framework for describing features whose geographical locations change over time, as noted in [11]. MEOS effectively realizes this framework through a conceptual model that includes a comprehensive suite of classes, operators, definitions, and attributes tailored for tracking the trajectories of moving objects.

[2] `https://libmeos.org/`

One of the primary benefits of MEOS is its foundation on an internationally recognized standard, coupled with its implementation in the C programming language. This combination provokes a robust and efficient environment for high-performance applications. The architecture of MEOS not only facilitates rapid processing but also supports easy extensions to other programming languages. This versatility makes MEOS an ideal core for various mobility database management systems (DBMS), such as MobilityDB[3].

Languages	Python, Java, Javascript, C
Systems	MEOS, MobilityDB, PostGIS, PostgreSQL
Operating Systems	MacOS, Ubuntu, Windows
Hardware	ARM, Intel, M1

Table 2.2: Overview of The MEOS family. Taken from [2].

Table 2.2 summarizes the languages, systems, operating systems, and hardware platforms compatible with MEOS, showcasing its flexibility and extensibility. Within the scope of this thesis, the PyMEOS (Python) [2] and JMEOS (Java) [4] bindings are particularly significant. The availability of Python and Java bindings for MEOS offers distinct advantages inherent to each language. Python, renowned for its simplicity and extensive libraries, facilitates the integration of MEOS functions and data types with popular packages such as *matplotlib*, enabling straightforward plotting of trajectories. Java, with its robust object-oriented architecture, provides access to enterprise-level frameworks like Spark, which are ideal for processing large-scale data across distributed systems. We compare MEOS and these two bindings in table 2.3.

Criteria	**MEOS**	**PyMEOS**	**JMEOS**
Language/Framework	C	Python	Java
MEOS Version Support	1.1.0+	1.1.0+	1.1.0-beta1
Abstraction Level	Low/Medium	High	High
Paradigm	Function-based	OOP	OOP

Table 2.3: Comparison of MEOS, PyMEOS, and JMEOS.

Both PyMEOS and JMEOS leverage the principles of Object-Oriented Programming (OOP) to develop a robust data model and achieve a high level of abstraction for managing mobility data. A significant benefit of PyMEOS is its compatibility with the latest versions of MEOS, ensuring that users have access to the most recent features and improvements. In contrast, JMEOS compatibility is somewhat restricted, supporting only up to version 1.1.0-beta1. This version introduces minor changes in the naming and structure of the classes and functions that map to MEOS. Given that MEOS serves as the foundational layer upon which these bindings operate, merely as interfaces, it is crucial to use the most current version of MEOS to fully leverage the capabilities provided by the object-oriented structure of the bindings. On the other hand, JMEOS offers distinct advantages for integration with systems like Spark, enabling the implementation of mobility-specific rules, data types, and functions not only at the user level but also within Spark's optimizer. This aspect will be explored in detail in Chapter 3.

[3] For additional details on projects involving MobilityDB and MEOS, visit `https://github.com/MobilityDB`. Discussions on MobilityDB and Distributed MobilityDB are elaborated further in Section 2.3.

[4] `https://github.com/MobilityDB/JMEOS`

The MEOS family supports a range of formats, including WKT (Well-Known-Text) and WKB (Well-Known-Binary), which are integral for the efficient handling of input-output operations, whether instantiating or persisting data types. Additionally, the data model within MEOS embraces a suite of temporal data types such as `tint`, `tbool`, `tfloat`, and `tpoint`, each tailored to capture and represent temporal aspects of data. These data types are designed to work seamlessly with a variety of aggregation functions that the framework offers, enhancing the analytical capabilities of the system.

The aforementioned types are integrally associated with the `TemporalInstant`, `TemporalSequence`, and `TemporalSequenceSet` classes, each serving distinct roles in the spatiotemporal analysis. The `Instant` class represents events occurring at a singular point in time and space, providing a snapshot of the spatiotemporal dimensions at a specific moment. The `TemporalSequence` class, on the other hand, encapsulates trajectories that evolve continuously over time and space, thereby capturing the dynamic nature of spatiotemporal phenomena. Lastly, the `TemporalSequenceSet` class is designed to handle complex trajectories that are partitioned into multiple sequences, enabling the representation of disjointed or segmented spatiotemporal paths. This differentiation allows for a comprehensive and nuanced modeling of various spatiotemporal behaviors and interactions.

Therefore, the fundamental PyMEOS datatypes, combined with `TemporalInstant`, `TemporalSequence`, and `TemporalSequenceSet` classes, offer robust frameworks for representing mobility data. For example, within the `TPoint` class, PyMEOS includes `TGeomPointInst`, `TGeomPointSeq`, and `TGeomPointSeqSet`. This pattern is consistent across other data types, such as `TFloatInst`, `TFloatSeq`, and `TFloatSeqSet`, among others.

Beyond basic data handling, MEOS provides an array of analytical, topological, and transformation functions that enrich data with meaningful context, thereby facilitating more complex analyses and interpretations. A particularly significant data type in MEOS is the `STBox`, or spatiotemporal box, which is crucial for operations such as calculating the boundaries of a dataset, segmenting data into manageable pieces, or representing the multidimensional aspects of the data domain. This functionality not only simplifies spatial and temporal queries but also optimizes the performance and scalability of applications utilizing MEOS for advanced geospatial and temporal data manipulation.

With the concepts seen in the previous section, and the MEOS/PyMEOS datatypes, building a mobility data use case is easy and convenient. Let's return to our geese example and compile the data into PyMEOS `TPointSeq` and `TInstSeq`, as shown in Table 2.4. Here, note that the altitude variable could be considered as part of the spatiotemporal attributes describing the `TPointInst`. However, it can be interpreted as well as a semantic attribute related to the points. Therefore, in this example the altitudes are created as `TInstSeq` objects that show the evolution of the altitudes over time. Note, as well, how the dataset is reduced drastically in size to only 3 rows. We can also calculate an `STBox` representing the spatiotemporal extent of the geese dataset:

```
STBOX XT(
    ((52.365,4.905),(52.3834,4.9115)),
    [2024-06-01 08:00:00+00, 2024-06-01 08:25:00+00
)
```

Until now, the discussion around MEOS and the associated mobility data model has operated under the premise that the data volumes are manageable enough to be stored on a single system. However, what happens when the trajectories are too voluminous to be accommodated collectively? In scenarios where such data is managed within a database, could the transactions involving spatiotemporal relationships become prohibitively costly?

AnimalID	TPointSeq	TAltitudeSeq
A1	[POINT(52.3761 4.9089)@2024-06-01 08:00:00+00, ...]	{15@2024-06-01 08:00:00+00, 20@2024-06-01 08:05:00+00, ...}
A2	[POINT(52.37 4.907)@2024-06-01 08:00:00+00, POINT(52.371 4.908)@2024-06-01 08:10:00+00, ...]	{12@2024-06-01 08:00:00+00, 15@2024-06-01 08:05:00+00, ...}
A3	[POINT(52.365 4.905)@2024-06-01 08:00:00+00, POINT(52.366 4.906)@2024-06-01 08:10:00+00, ...]	{10@2024-06-01 08:00:00+00, 14@2024-06-01 08:05:00+00, ...}

Table 2.4: Geese movement and altitude data.

It may be more pragmatic to *partition* these extensive trajectories and the related data into smaller, more manageable segments. A database management system (DBMS) that utilizes MEOS as its mobility data engine could see significant performance improvements by strategically organizing these trajectory partitions according to their spatiotemporal contexts. This approach not only makes sense from a data management perspective but also enhances processing efficiency by enabling parallelization across various stages of the data handling pipeline. In Section 2.2, we will delve into prevalent partitioning techniques specifically tailored for mobility data, exploring their methodologies and assessing their impact on system performance and scalability.

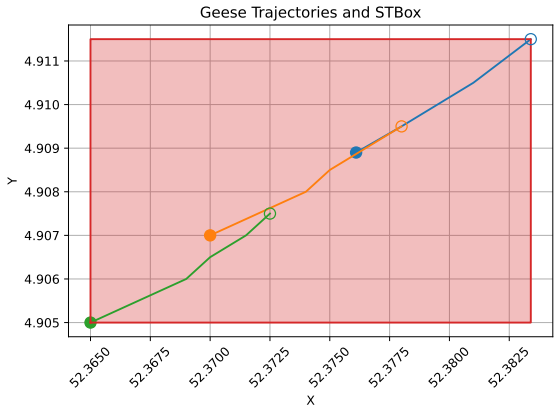

Figure 2.2: Geese example trajectories and spatiotemporal box using PyMEOS.

2.2 Mobility Data and Partitioning Concepts

As previously discussed, utilizing PyMEOS provides significant advantages by establishing a structured framework with specialized data types and functions tailored for mobility data analysis. This framework is particularly

effective when working with small and manageable datasets that can be easily stored in memory or within a single database system. However, the challenge arises when dealing with larger datasets or trajectories that cannot be processed as a whole due to their size or complexity. In such cases, optimizing data processing becomes crucial.

The central question to consider in this section is: What is an efficient method to partition a mobility dataset to optimize the processing of all associated data? In other words, we seek to identify strategies to divide and organize trajectory data, and mobility data in general, in a manner that groups similar items together. This process is referred to as *partitioning*.

This section will begin by exploring key definitions and classical techniques for partitioning large datasets, with a particular focus on mobility data. Following this, we will introduce the concept of point-in-polygon queries, which forms part of the methodology applied in the practical implementation discussed in Chapter 3.

2.2.1 Partitioning Mobility Data

Consider a scenario where, in our geese tracking example, the trajectory data becomes so voluminous, with thousands of instants per trajectory, that it becomes impractical to manage on a single machine. The question then arises: how should the data be partitioned?

Definition 2.2.1 (Partitioning)**.** Process of dividing a *dataset*—in the context of data science—or a *relation*—from a database perspective—into distinct segments. Formally, it can be described as splitting a dataset D into n partitions, $D = \bigcup_{i=1}^{n} D_i$, where each D_i is a subset of D, and together they reconstitute the entire dataset [12].

This division offers significant advantages in both storage and processing, as data can be distributed across different nodes in a distributed system, allowing each worker node to process its portion independently, without the need for extensive data exchange between nodes. The primary objective is to ensure that data colocated on the same node shares inherent relationships, thereby enhancing the efficiency of operations performed on that data.

In the context of trajectory data, as explored in this thesis, partitioning becomes particularly critical because it must consider the spatiotemporal attributes intrinsic to the data. For instance, a query that seeks information on geese passing through a specific point (x, y) during the time interval $[t_s, t_f]$ can be processed more efficiently if the system can exclude irrelevant data not pertaining to the specified location and timeframe.

Traditional DBMS frameworks often employ straightforward partitioning techniques based on a single attribute or a composite key. A broader application of this can be seen in traditional data warehousing systems, where fact tables are partitioned such that related transactions are stored in close proximity. This partitioning might be based on dimensions such as time intervals, user demographics, price ranges, or a combination thereof. The selection of appropriate features for partitioning is crucial, as it significantly influences the effectiveness and efficiency of the data partitions, tailored to the specific use case and the characteristics of the dataset.

Numerous surveys evaluate big data partitioning techniques [13, 14]. The survey conducted in [15] evaluates various big data partitioning techniques from a traditional standpoint, distinguishing between *horizontal* and *vertical* partitioning.

Definition 2.2.2 (Horizontal Partitioning)**.** Division of a relation into disjoint subsets, where each subset contains a complete set of attributes, but only a subset of the tuples. This technique segments the data into rows, such that $R = \bigcup_{i=1}^{n} R_i$.

Definition 2.2.3 (Vertical Partitioning)**.** Dividing a relation into subrelations, where each subrelation contains a subset of the attributes and all the tuples. A relation R with attributes $(A_1, A_2, \ldots, A_m)$ can be partitioned into $R_1, R_2, \ldots, R_k$ such that each $R_i = \pi_{A_i}(R)$ where A_i is a subset of the attributes. This technique is useful when queries frequently access only a subset of attributes.

A hybrid approach between vertical and horizontal partitioning is known as *functional partitioning*, similar to Data Marts in data warehousing design. In functional partitioning, data is divided based on the intended application or workload, enhancing data processing and retrieval efficiency. For example, data can be segregated into distinct categories such as invoice data and product inventory, optimizing the processing of queries specific to each category.

Definition 2.2.4 (Simple Random Partitioning)**.** A method where each record $r_i \in R$ from a dataset R is assigned to one of n partitions $P = \{P_1, P_2, \ldots, P_n\}$ based on a random function $f(r_i)$. This function maps each record to a partition without considering any attribute of r_i, potentially leading to imbalanced distributions. While this method can achieve approximate balance, it lacks guarantees on distribution uniformity.

Definition 2.2.5 (Hash Partitioning)**.** A method which allocates records $r_i \in R$ to partitions $P = \{P_1, P_2, \ldots, P_n\}$ by computing a hash function $h(a)$ on a key attribute a of each record. This method ensures that records with the same key value are consistently placed in the same partition, promoting data locality.

Definition 2.2.6 (Range Partitioning)**.** A method which divides a dataset R into partitions $P = \{P_1, P_2, \ldots, P_n\}$ based on the range of values in a specified attribute a. For each partition P_j, there exists a range $[r^j_{min}, r^j_{max}]$ where a_i is the value of the attribute for record r_i. This method is effective for datasets with naturally ordered attributes, facilitating efficient range queries.

The mentioned techniques are suitable for general use cases, where the nature of the data permits a standard solution. The distribution and partitioning of tables can be determined by a range or by the ID of the relation. Several systems natively support these partitioning options, which are crucial for optimizing query planning and execution[5].

Until now, we have explored data partitioning in generic scenarios where simple horizontal or vertical partitioning policies substantially enhance DBMS operations. However, these methods primarily suit tabular-like data and may fall short for specific use cases that demand specialized partitioning schemes, such as those involving mobility data.

The inherent complexity of spatiotemporal mobility data presents significant challenges. For example, consider the scenario of distributing taxi trajectory data in New York: Should the data be split based on the geographical areas most frequented, or should it be divided according to the times when taxis are most active? The answer is not straightforward and varies according to specific use case requirements. The task of partitioning datasets with spatial and/or temporal components has been extensively studied, and recent advancements have approached this issue with innovative methods. Here, we review the most pertinent research.

[5] An excellent practical tutorial on basic partitioning types in PostgreSQL can be found at `https://tinyurl.com/PostgreSQL-partitioning`. These techniques are system-agnostic and applicable across various DBMS platforms.

Grid-based partitioning techniques segment spatiotemporal data into various slices, typically existing in two dimensions (spatial (x, y)), three dimensions (spatiotemporal (x, y, t)), or with additional dimensions such as altitude (x, y, z, t). *Regular grid* partitioning defines the number of partitions per dimension, segmenting the dataset into regular intervals along each dimension, akin to multidimensional range partitioning. Techniques such as KD-Tree [16] and R-Tree [17] apply this principle, allowing for non-regular intervals to preserve proximity among stored objects—a concept known as *multidimensional tiling*. Recent works like [7] and [8] have implemented this as the foundational partitioning scheme in Distributed MobilityDB. Grid-based approaches are of particular importance within this thesis, with a focus on the Regular Grid and KD-Tree partitioning techniques.

Definition 2.2.7 (Regular Grid Partitioning)**.** Divides a dataset into a uniform grid by partitioning each dimension into equal-sized intervals. For a dataset D with dimensions $d_1, d_2, \ldots, d_n$, each dimension d_i is divided into k_i equal parts, creating grid cells where each cell contains data points falling within the corresponding intervals.

Definition 2.2.8 (KD-Tree Partitioning)**.** Recursively partitions a dataset along its dimensions, using median values to create balanced, hierarchical splits. For a dataset D with dimensions $d_1, d_2, \ldots, d_n$, KD-Tree Partitioning splits the data at the median of the current dimension, then recursively applies the same process to the resulting subspaces, alternating dimensions at each level.

Graph-based strategies conceptualize spatiotemporal data as evolving graphs, with moving objects and locations represented as vertices connected by edges that encode the timestamp of each interaction. These graphs, often comprising millions or trillions of edges, pose unique challenges. A distributed spatiotemporal graph model designed for edge partitioning across a cluster of nodes called PAST is detailed in [18], leveraging data replication for efficiency. Additionally, [19] uses graph partitioning to enhance the training of a Deep Convolutional Recurrent Neural Network (DCRNN) for traffic prediction, partitioning the dataset into proximate spatiotemporal chunks for independent training using an ensemble methodology.

Another category for partitioning techniques with mobility data is that of *learned partitioning*. This emerging category employs neural networks to optimize existing partitioning configurations or develop new ones from scratch. For example, [20] utilizes Deep Reinforcement Learning to derive an optimal partitioning scheme for spatial data, training based on established partitioning methods to minimize training duration.

On the other hand, clustering-based partitioning leverages the proximity between related spatiotemporal points or trajectories to effectively partition data. Although simple clustering methods may not always ensure balanced load distribution, they are beneficial when load balancing is not the primary concern. The work in [21] discusses a clustering-based method using nearest neighbors to optimize partitioning for spatial data queries within Spark environments. Research from [22] integrates a DBSCAN-based approach with K-means to partition trajectory data effectively. This illustrates how virtually any clustering algorithm can be employed to partition mobility data, exploiting its multidimensional nature. In this project, a similar strategy is implemented using *Bisective K-Means* as a partitioning method for mobility data.

Definition 2.2.9 (Bisective K-Means Partitioning)**.** Applies recursively the K-Means clustering algorithm to divide data into two clusters at each step, creating a binary partitioning tree. For a dataset D, this method first applies K-Means to split D into two clusters C_1 and C_2, and then recursively applies K-Means to each resulting cluster, continuing until a desired number of clusters is reached or further partitioning is considered unnecessary.

Finally, deep-learning partitioning techniques can also be employed. This method leverages deep learning to identify the most appropriate partitioning strategy based on the characteristics of the data. The approach involves sampling data to approximate its distribution, with the deep learning model then selecting the optimal partitioning strategy, which may include Grid, KD-Tree, or cluster partitioning methods [23]. This approach effectively learns which clustering technique is best suited for various scenarios.

This thesis adheres to the philosophy that simplicity is preferable. Grid-based approaches, in particular, are straightforward and faster compared to other partitioning methods. Techniques like deep-learning and graph-based approaches often require a restructuring of the mobility dataset. Moreover, deep-learning and cluster-based approaches involve training a model to determine the partition keys for each record. Grid-based methods, on the other hand, offer the advantage of simplicity and provide a good balance between computational efficiency and load balancing. They often consider the spatiotemporal densities of the data points, making them effective for mobility data.

2.2.2 Point-in-Polygon Queries and Partitioning

Understanding the impact of efficient mobility query processing is crucial for this project. We have explored various partitioning techniques for splitting data, but it's also important to consider how partitioning affects the usage and querying of this data. Efficient partitioning can significantly enhance the performance of mobility queries, which often aim to answer questions such as: Did mobility object A interact with mobility object B? Here, the term *interact* encompasses any spatiotemporal function f. A key concept in this context is the *point-in-polygon* query [24] (also known as *PIP query*), a term initially termed in the computer graphics field, which can be extended to include spatiotemporal data.

Definition 2.2.10 (Point-in-Polygon Query)**.** Determines whether a given point (x, y) lies within the boundaries of a polygon P. For a point $p = (x, y)$ and a polygon P, the query returns true if contains(p, P), and false otherwise.

Definition 2.2.11 (Spatiotemporal Point-in-Polygon Query)**.** Extends the traditional point-in-polygon query to include a temporal dimension. It determines whether a point (x, y, t) at time t is within a moving polygon $P(t)$. For a point $p = (x, y, t)$ and a polygon $P(t)$ at time t, the query returns true if contains(p, P) at time t, and false otherwise.

NestID	Geometry
Nest1	`POLYGON ((52.362 4.90, ...`
Nest2	`POLYGON ((52.374 4.90, ...`
Nest3	`POLYGON ((52.379 4.90, ...`

Table 2.5: Nest dataset.

Both definitions are applicable to any mobility data type included in the previously discussed data model, and the `contains` function can be adapted as needed. However, we adhere to the general definition of *point-in-polygon* to showcase the concepts.

To illustrate the relationship between point-in-polygon queries, partitioning, and mobility data, let us revisit our geese example. In Figure 2.2, we demonstrated how an in-memory instance of mobility data functions. The figure displays the geese trajectories and the bounding box encompassing the entire set of data points. Now, consider introducing a separate dataset containing nest data, as represented in Table 2.5. Each row corresponds to a known nest used by the geese. Plotting these nests yields Figure 2.3. In this example, only one trajectory intersects with one of the nests.

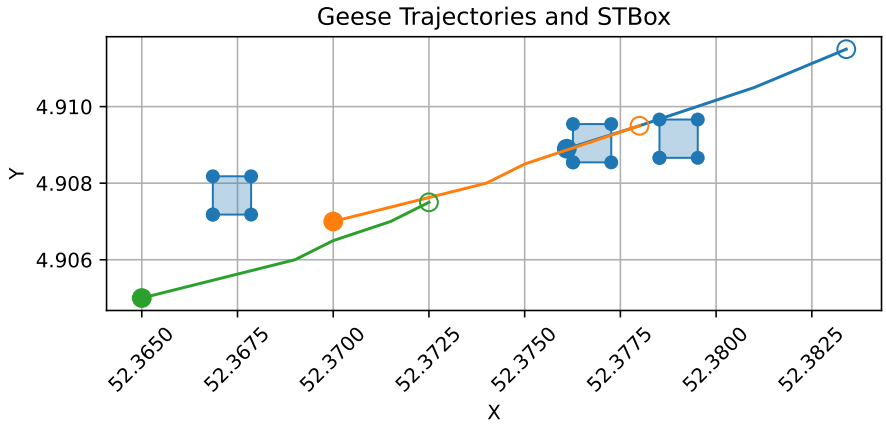

Figure 2.3: Geese trajectories and nests.

A typical *point-in-polygon query* seeks to identify which geese passed by the nest, and if they did, to determine the specific instants of intersection. This can be efficiently addressed using the previously defined *at* function.

The brute-force approach involves comparing all trajectory instants against all nests, known as a *cartesian product*. Given that the search space is currently constrained only by the overall spatiotemporal bounds of the trajectories (as illustrated in Figure 2.2), the question arises: would partitioning this space in a way that limits the comparisons to relevant objects be beneficial?

This exact approach was employed by the Databricks team in [25], particularly in [26]. They introduced PIP queries as a method for identifying areas of interest for querying and joining spatial datasets. Spatiotemporal datasets, such as mobility data, can similarly benefit from this technique. The concept involves constructing a *grid* using any of the grid-partitioning techniques previously discussed and assigning an identifier to each *tile* within the grid. By pre-calculating whether a tile *contains* a mobility object, the query can be further refined to consider only objects within the overall bounds if and only if both A and B are contained within tile T.

This method effectively creates an artificial index by using the grid as a system for partitioning and tiling the data. Chapter 3 delves into and demonstrates the advantages of using this system. For a quick example using the geese dataset, we can create a simple regular grid around the bounds of the trajectories and assign tile identifiers to the corresponding objects within each dataset. The resulting Tables 2.6 and 2.7 illustrate this process. Note that we are pre-computing the PIP query over the tile and then performing a join operation based on the *tileid* for the objects contained within the tiles. Depending on the use case, one may choose to retain the entire dataset or split it. For trajectories, splitting can be facilitated using the *at* topological function.

By initially filtering with the PIP query on the tiled bounds, the number of comparisons between the nest

tileid	NestID	Geometry
0	Nest1	POLYGON ((52.366 4.907,...
1	Nest1	POLYGON ((52.366 4.907,...
6	Nest2	POLYGON ((52.378 4.908,...
6	Nest3	POLYGON ((52.376 4.908,...
7	Nest2	POLYGON ((52.378 4.908,...
7	Nest3	POLYGON ((52.376 4.908,...

Table 2.6: Tiled nests.

tileid	AnimalID	TPointAt
0	A2	{[POINT(52.37 4.907)@2024-06-01 08:00:00+00, P...
0	A3	{[POINT(52.365 4.95)@2024-06-01 08:00:00+00, ...
1	A3	{[POINT(52.3695 4.96)@2024-06-01 08:12:30+0...

Table 2.7: Tiled geese trajectories (sample).

geometries and trajectory *tpoints* is significantly reduced, making the process more efficient and feasible [26]. This concept is clearly illustrated in Figure 2.4, where the executed operations are the ones highlighted in the orange tile. The gray geometries indicate trajectories or geometries which are discarded through the PIP filtering. Note the portion of interest of the trajectory is only considered, and nests which are not crossed by any goose are also discarded.

This section has introduced key terminology and concepts utilized in this project to implement *partitioning*, a critical component of mobility data management in a distributed environment. We will now explore essential concepts in mobility data management systems and examine the state-of-the-art in distributed mobility data management systems, with a particular focus on Spark-based systems.

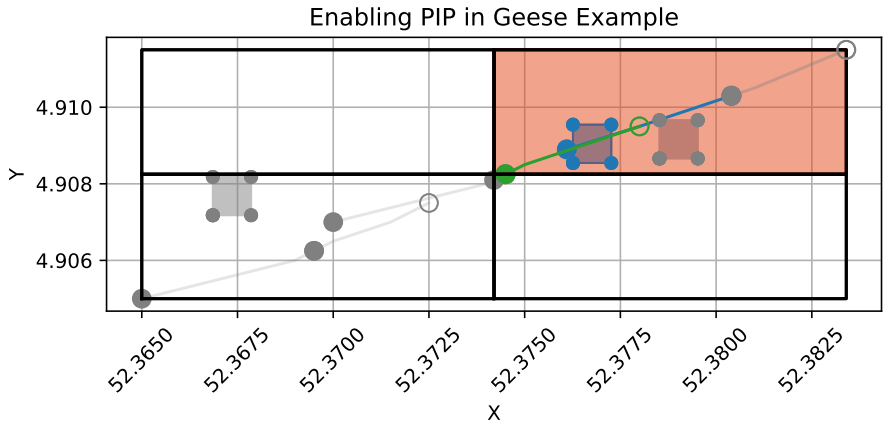

Figure 2.4: Restricting search space using PIP query.

2.3 Mobility Data Management Systems

Thus far, we have introduced the data model utilized in the MEOS system, along with the partitioning techniques frequently employed to segment mobility data, encompassing both spatial and spatiotemporal aspects. The next focus is to define and examine key concepts and systems used to manage mobility data, known as Mobility Data Management Systems (MDMS) or Mobility Database Management Systems (MDBMS).

With the coming of Big Data, many MDBMS have evolved to become *distributed*. This implies that data is managed and stored across multiple machines, collectively referred to as a *cluster*. In such a cluster, the data processing tasks are typically coordinated by a driver node, which oversees the distribution and execution of tasks, while the worker nodes perform the actual data processing and computation. One such example of a Distributed DBMS is Spark [27], which will be examined in detail in Subsection 2.3.2.

A central focus of this thesis is to explore how a Distributed DBMS can be effectively adapted to manage mobility data. This involves leveraging the data model defined earlier alongside the benefits of a distributed environment. The integration of these tools can be achieved through the application of the partitioning techniques outlined earlier in this chapter, ensuring efficient data distribution across the cluster.

This section will initially review key concepts pertinent to the utilization of distributed database management systems. Subsequently, it will delve into the inner workings of Spark, and conclude with an overview of significant MDBMS, beginning with MobilityDB.

2.3.1 Distributed Database Management Systems

Since the inception of relational databases and relational algebra [28], the management and storage of datasets have been significantly influenced by their abstraction into tables, referred to as *relations*. These relations consist of a defined set of columns and rows, enabling various operations such as selection, projection, join, and aggregation, which are fundamental to the manipulation and analysis of data in relational databases.

Definition 2.3.1 (Relation)**.** Set of tuples that share the same attributes. Formally, a relation R can be defined as a subset of the Cartesian product of a list of domains $D_1, D_2, \ldots, D_n$, such that $R \subseteq D_1 \times D_2 \times \cdots \times D_n$. Each attribute corresponds to a column in the table, and each tuple represents a row.

Definition 2.3.2 (Selection)**.** Denoted as σ, retrieves tuples from a relation that satisfy a given predicate. Formally, if R is a relation and θ is a predicate, then $\sigma_\theta(R)$ represents the set of all tuples $t \in R$ for which $\theta(t)$ is true.

Definition 2.3.3 (Projection)**.** Denoted as π, extracts specific columns from a relation, eliminating duplicates to ensure the result is a set. For a relation R and a set of attributes A, the projection $\pi_A(R)$ yields a relation consisting of unique tuples with only the attributes in A.

Definition 2.3.4 (Join)**.** Combines tuples from two relations based on a common attribute or condition. Denoted by $\bowtie$, if R and S are relations and θ is a condition, then $R \bowtie_\theta S$ represents the set of all concatenated tuples (r, s) where $r \in R$ and $s \in S$ such that $\theta(r, s)$ holds.

Definition 2.3.5 (Aggregation)**.** Involves summarizing data from a relation by applying a function, denoted *aggregate function*, to a group of tuples. Common aggregate functions include SUM, COUNT, AVG, MIN, and

MAX. For a relation R, an aggregation operation might be denoted as $\gamma_{A,f}(R)$, where A is a set of attributes to group by, and f is the aggregate function applied to each group.

What happens when a relation R cannot be set within a single machine due to its size? This is where a distributed architecture becomes essential. Distributed multirelational algebra, as introduced in [12], extends the traditional relational algebra to support operations across multiple machines. This extension requires modifications to the standard definitions of relational operations to accommodate the distribution of data.

Definition 2.3.6 (Distributed Relation)**.** A relation whose tuples are partitioned across multiple machines or nodes in a distributed system. Each partition can be processed independently, and collectively they represent the entire dataset. Formally, if R is a relation, it can be expressed as a union of disjoint subsets $R = \bigcup_{i=1}^{n} R_i$, where each R_i is stored on a different node. Note how this definition is equal to that of Partitioning (Definition 2.2.1), applied to a relational, distributed context.

Definition 2.3.7 (Distributed Selection)**.** Applies a selection operation to each partition of a distributed relation independently. Denoted as $\sigma_\theta(R)$, where R is distributed as $R = \bigcup_{i=1}^{n} R_i$, the selection operation is applied to each partition R_i to produce $R' = \bigcup_{i=1}^{n} \sigma_\theta(R_i)$. This parallel application allows for efficient filtering of data across the distributed system.

Definition 2.3.8 (Distributed Projection)**.** Involves applying the projection operation to each partition of the distributed relation. For a relation R partitioned as $R = \bigcup_{i=1}^{n} R_i$, and a set of attributes A, the projection $\pi_A(R)$ is computed as $R' = \bigcup_{i=1}^{n} \pi_A(R_i)$. The operation reduces each partition to the specified attributes, ensuring that the result remains a set of unique tuples across all partitions.

Definition 2.3.9 (Distributed Join)**.** Involves joining tuples from two distributed relations R and S, which may be partitioned across different nodes. The join condition θ is applied across partitions, and the results are combined. Formally, if R and S are partitioned as $R = \bigcup_{i=1}^{n} R_i$ and $S = \bigcup_{j=1}^{m} S_j$, the join $R \bowtie_\theta S$ is performed by joining each R_i with each S_j that satisfies the join condition, resulting in $R' = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} (R_i \bowtie_\theta S_j)$.

Definition 2.3.10 (Distributed Aggregation)**.** Applies aggregate functions to partitions of a distributed relation, followed by a final aggregation step to combine results. If R is partitioned into $R = \bigcup_{i=1}^{n} R_i$, and f is an aggregate function (e.g., SUM, COUNT), the aggregation is performed in two stages: first, locally on each partition, yielding $f(R_i)$ for each R_i, and then a global aggregation, $f(R) = f(\bigcup_{i=1}^{n} f(R_i))$, to produce the final result.

Distributed joins and aggregations in a database system pose significant challenges due to the necessity of *data exchange* between partitions to compute the final result. For example, consider relations A and B, each divided into two partitions, A_1, A_2 and B_1, B_2. To compute a join between these relations, it is necessary to pair each partition across nodes, such as pairing A_1 with B_1 and B_2. This data exchange process across nodes is computationally expensive and introduces additional overhead to query processing. This thesis particularly focuses on two concepts that address these challenges: colocation and the MapReduce framework [29].

Definition 2.3.11 (Colocation)**.** Refers to the strategy of placing tuples from different relations, which are likely to be accessed together, on the same node or partition. Given two relations R and S with schemas $R(X, Y)$ and $S(Y, Z)$, colocation ensures that tuples $r \in R$ and $s \in S$, where $r_Y = s_Y$, are stored on the same partition. This strategy minimizes the need for data shuffling during join operations, reducing network communication and improving query performance.

Definition 2.3.12 (MapReduce)**.** Programming model for processing and generating large datasets through parallel and distributed computation. It consists of two primary functions:

- *Map:* The map function, $m : (K_1, V_1) \to \{(K_2, V_2)\}$, takes an input pair (K_1, V_1) and produces a list of intermediate key-value pairs (K_2, V_2). Here, K_1 and K_2 are the input and output keys, respectively, while V_1 and V_2 are the corresponding values.
- *Reduce:* The reduce function, $r : (K_2, \{V_2\}) \to \{(K_3, V_3)\}$, aggregates the intermediate data from the map phase. For each key K_2, the reduce function processes the associated list of values $\{V_2\}$ to produce a final set of key-value pairs (K_3, V_3), where K_3 is often the same as K_2 or a derived value.

These concepts are exploited by systems such as Hadoop [30] and Spark [27]. In particular the case of Spark provides a complete suite of APIs that leverage distributed SQL [31] to perform operations, with MapReduce executing in the background, as we will study in the next subsection.

These concepts are effectively utilized by systems such as Hadoop [30] and Spark [27]. Specifically, Spark offers a comprehensive suite of APIs designed to enable distributed data processing. It leverages distributed SQL [31] to perform various operations, with MapReduce functioning as the underlying computational model. This integration allows Spark to handle large-scale data efficiently, providing capabilities for both batch and real-time processing. In Section 2.3.2, we will delve deeper into Spark's architecture.

2.3.2 Spark

The Apache Spark system [27, 32] is a powerful tool for Big Data analytics and processing. It extends and generalizes the MapReduce [29] framework, which is foundational to Hadoop [30], and enhances it for more comprehensive Big Data processing. Spark is not just limited to data processing; it also includes a library that supports efficient big data transactions and implements its own query language, Spark SQL [31]. Spark SQL is particularly notable for its use of the Catalyst optimizer, a robust query optimization engine that helps in efficiently executing SQL queries on large datasets.

Although Spark is primarily built on Java and Scala, it offers API extensions for several programming languages, including R and Python. These APIs act as gateways to Spark's functionalities, allowing users to write code in these languages while the underlying operations are serialized and executed within the Java Virtual Machine (JVM) associated with the Spark core. This architecture ensures that Spark can leverage the performance advantages of the JVM while providing flexibility to users familiar with different programming environments.

Spark's data processing capabilities are centered around two key data abstractions: DataFrames and Resilient Distributed Datasets (RDDs). DataFrames are a higher-level abstraction, similar to a table in a relational database, and provide a more expressive and user-friendly interface for data manipulation. They support a wide range of data types and operations, making them ideal for complex data analysis tasks. DataFrames are built on top of RDDs and offer optimizations that RDDs do not, such as columnar storage and more efficient memory usage.

Definition 2.3.13 (Spark DataFrame)**.** Distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R or Python, but with richer optimizations. DataFrames can be constructed from a wide array of sources such as structured data files, tables in Hive, external databases, or existing RDDs.

On the other hand, RDDs are the core abstraction of Spark, representing a fault-tolerant, distributed collection of elements that can be operated on in parallel. RDDs provide the foundational data structure from which DataFrames are built and offer more control over low-level operations. They are immutable, meaning that once an RDD is created, it cannot be changed. This immutability enables a robust fault tolerance mechanism, as RDDs can automatically recompute lost data using the lineage information of transformations applied to the data. Figure 2.5 shows the abstraction levels for the Spark architecture.

Definition 2.3.14 (Resilient Distributed Dataset)**.** Distributed data structure that allows for parallel processing of large datasets across a cluster of computers. RDDs are immutable and can be built through deterministic operations on data either in stable storage or in other RDDs. They are the fundamental data structure of Spark, providing a low-level API for data manipulation and transformation.

Definition 2.3.15 (Spark SQL)**.** A module in Apache Spark that integrates relational processing with Spark's functional programming API. Spark SQL supports both SQL and the Hive Query Language (HQL), and it can read data from various sources including Hive tables, JSON files, and Parquet files.

Definition 2.3.16 (Catalyst)**.** Query optimization engine in Spark SQL. It is designed to handle both SQL queries and complex data transformations. Catalyst provides a rich framework for building query plans and optimizations, including rule-based and cost-based optimizations. It leverages a pluggable architecture, allowing for custom rules and optimizations to be easily integrated into the query planning process.

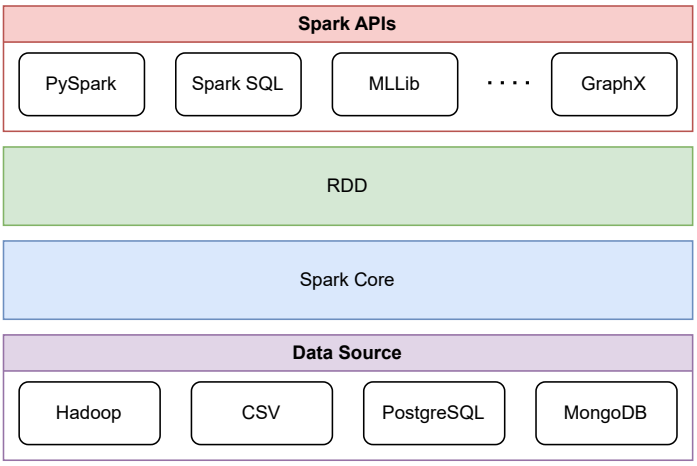

Figure 2.5: Apache Spark abstraction levels.

Spark partitions input datasets by default, a mechanism that divides large datasets into smaller, manageable pieces known as partitions. This partitioning is based on the data's hash value or, in the case of sorted data, the range of the partition keys. Each partition is processed independently, which allows Spark to perform operations in parallel across a distributed computing cluster. This approach, often referred to as *divide and conquer*, is fundamental to Spark's ability to handle Big Data efficiently. The partitions are managed using MapReduce operations, enhanced by the optimized query plans generated by the Catalyst engine.

To further optimize the distribution of data, Spark allows users to specify strategies for partitioning and bucketing. These methods improve the efficiency of data access and query performance by organizing the data in a way that reduces the amount of data shuffling required during processing.

Definition 2.3.17 (Spark Partitioning)**.** Method of dividing a dataset into smaller partitions across the cluster nodes. This can be done based on a hash of a column's value (hash partitioning) or by range (range partitioning), where the dataset is divided into partitions based on a defined range of column values. Partitioning helps in distributing the workload evenly across the nodes, thus enhancing parallelism and reducing computation time.

Definition 2.3.18 (Spark Bucketing)**.** Technique used to organize data into fixed-size buckets based on a specified column. This is particularly useful when joining two large datasets, as it ensures that rows with the same bucket key are colocated, reducing the shuffle phase and optimizing query performance. Unlike partitioning, which can result in a varying number of rows per partition, bucketing provides a more uniform distribution of data.

These strategies—partitioning and bucketing—are crucial in Spark for optimizing data access patterns and minimizing data movement across the cluster. By carefully choosing the partitioning and bucketing keys, users can significantly enhance the performance of their Spark jobs, making the system more efficient and scalable.

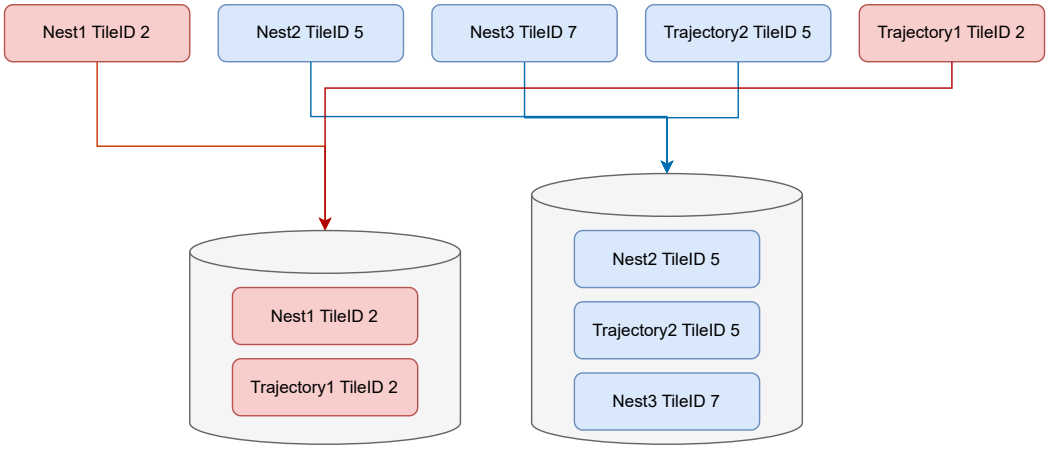

Figure 2.6: Bucketing the geese example.

Figure 2.6 illustrates bucketing under our geese example. Imagine the geese data also contains a separate dataset with information about the geese nests. This dataset is also considered a mobility dataset as it contains the spatial location of the nest. If we divided the spatiotemporal data into a grid of tiles, and assigned each trajectory and nest a *tileId* value correspondent to the tile in which each trajectory or nest is located, we can easily colocate the data by bucketing. In the example, we assume two buckets, meaning that Spark would allocate the data to each bucket using a hash function $h(r) = r_{val} \text{mod}(b)$, where r is the current row, the *val* is the value to bucket by, and b is the number of buckets. For this example the formula would be over 2 buckets, meaning that pair *tileid* values would go to one bucket and odd values would go to the other one.

Ensuring colocation is crucial when two or more relations are frequently queried together, especially if a join operation is involved. Colocation can significantly enhance query performance by reducing the need for data

shuffling across nodes in a distributed system. Spark SQL, through the Catalyst optimizer, implements several join strategies to efficiently handle join operations. These strategies include Broadcast Nested Loop Join, Broadcast Hash Join, Sort Merge Join, Shuffle Hash Join, and Cartesian Join. The choice of join strategy depends on various factors such as the size of the datasets, the availability of indices, and the data distribution.

Definition 2.3.19 (Broadcast Nested Loop Join)**.** Join strategy where Spark broadcasts a small dataset to all nodes, and each node then performs a nested loop join with its partition of the larger dataset. This method is efficient for very small datasets but can be costly if the broadcasted dataset is large. Common use cases include joining a small reference table with a much larger fact table. For example, joining a small lookup table containing country codes with a large table of user data.

Definition 2.3.20 (Broadcast Hash Join)**.** Join strategy which involves broadcasting a small dataset to all worker nodes and performing a hash join on the distributed data. This is particularly efficient when one of the tables is small enough to fit in memory. Catalyst automatically chooses this join if the dataset is small. A typical use case would be joining a small configuration or lookup table with a larger dataset, such as joining user demographic data with a large transaction dataset.

Definition 2.3.21 (Sort Merge Join)**.** Join strategy used when both datasets are large and sorted. It involves sorting the data on the join key and then merging the sorted datasets. This join strategy is efficient for large datasets where the sort operation can be optimized. It is commonly used in scenarios where the datasets are already sorted or can be easily sorted on the join keys, such as joining large log files on timestamp or user ID.

Definition 2.3.22 (Shuffle Hash Join)**.** Join strategy where Spark shuffles the data to ensure that matching keys from both datasets are colocated in the same partition, followed by a hash join on the shuffled partitions. This strategy is beneficial when dealing with large datasets where neither can be broadcasted, but the data needs to be shuffled to ensure colocation. An example use case would be joining two large tables on a common key where both tables are distributed across different partitions.

Definition 2.3.23 (Cartesian Join)**.** Join strategy, also known as a cross join, where each row from the first dataset is combined with each row from the second dataset. This join strategy results in a Cartesian product of the two datasets. It is generally avoided due to its high computational cost unless explicitly required for generating all combinations of two datasets, such as in scenarios where cross-validation data needs to be prepared.

The use of bucketed tables can significantly improve join performance in Spark SQL. Bucketing organizes data into fixed-size buckets, which ensures that rows with the same bucket key are colocated in the same partition. This colocation reduces the amount of data shuffling needed during join operations, making joins more efficient. For example, if both tables in a join are bucketed on the join key, Spark can leverage the bucketed layout to perform joins directly within each bucket, thereby minimizing the shuffle phase and speeding up the query. This is especially beneficial in large-scale data processing where reducing data movement can lead to significant performance gains. Chapter 3 covers a demonstration of these gains through the *BerlinMOD* experiments.

2.3.3 MobilityDB and Other Solutions

MobilityDB [33, 34] is a DBMS specifically designed to efficiently store and manage trajectories, moving objects, and related data types. The system optimizes data access through the use of indexes, specialized data types, and data cleaning techniques. Built upon PostgreSQL and extended with PostGIS, which provides spatial data support, MobilityDB integrates the processing of geospatial data within a temporal framework. As mentioned earlier, MobilityDB works with MEOS as an engine to extend its functionalities and data types into PostgreSQL.

Recent developments have aimed at expanding MobilityDB into a distributed setting [7, 8] using Citus to scale across a distributed PostgreSQL architecture. This transition to a distributed environment is managed through a classic master/slave model, where a coordinator node maintains essential metadata about data distribution across worker nodes.

Figure 2.7 illustrates the architecture of Distributed MobilityDB. At the foundation, each node hosts a complete MobilityDB instance. The coordinator stores critical metadata, essential for managing data distribution to worker nodes. The Distributed Storage layer is responsible for implementing global and local data partitioning and indexing strategies, enabling efficient management, localization, and processing of data queries and ingestion. The Distributed Query Processing layer oversees the query planner and optimizer, producing tailored plans based on user queries. At the highest level, users interact with the system as if it were not distributed, maintaining an illusion of a singular, unified database environment.

It is insightful to compare the structure of Distributed MobilityDB with that of Spark, as both systems utilize a master/worker cluster configuration. While Spark processes queries through the Catalyst optimizer, this can be viewed as a form of *Distributed Query Processing*, similar to what is depicted in Figure 2.7. This structural similarity suggests that the Distributed MobilityDB architecture could be adapted for a Spark environment. Given that MEOS surrounds each installation of MobilityDB, this architecture can potentially be recreated across different distributed systems, thereby extending beyond the PostgreSQL-dependent framework that MobilityDB currently employs.

For the purposes of this thesis, it is also relevant to explore other Hadoop and Spark-based systems that focus on mobility data. *ST-Hadoop*, as introduced in [35], is an extensible MapReduce framework designed specifically for processing spatiotemporal data, incorporating various optimizations at different levels. At the core of this project is the partitioning level, where ST-Hadoop first partitions the data by time ranges (e.g., daily, monthly) as specified by the user. These temporal partitions are then spatially partitioned into segments of approximately 64MB each. Similarly, *Summit* [36] builds upon ST-Hadoop by introducing a sampling phase in the partitioning process. This sampling phase helps to account for the data distribution during partitioning, ensuring a more balanced workload. After sampling, the partitioning is executed in the same manner as in ST-Hadoop.

Additionally, *HadoopTrajectory*, described in [37], integrates spatiotemporal-aware operations and data types, effectively handling the partitioning of large data files while preserving the semantic integrity and structure of moving objects. It utilizes two key partitioning techniques: a regular grid and an R-Tree structure. An R-Tree is a tree data structure used for spatial access methods, i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles, or polygons. This method enables efficient querying and data retrieval by organizing the data in a way that reflects its spatial properties.

For Spark-based systems focused on mobility data, some notable examples include *TrajSpark* [38], *UITrajMan* [39], *Simba* [40], *Dita* [41], and *Beast* [42].

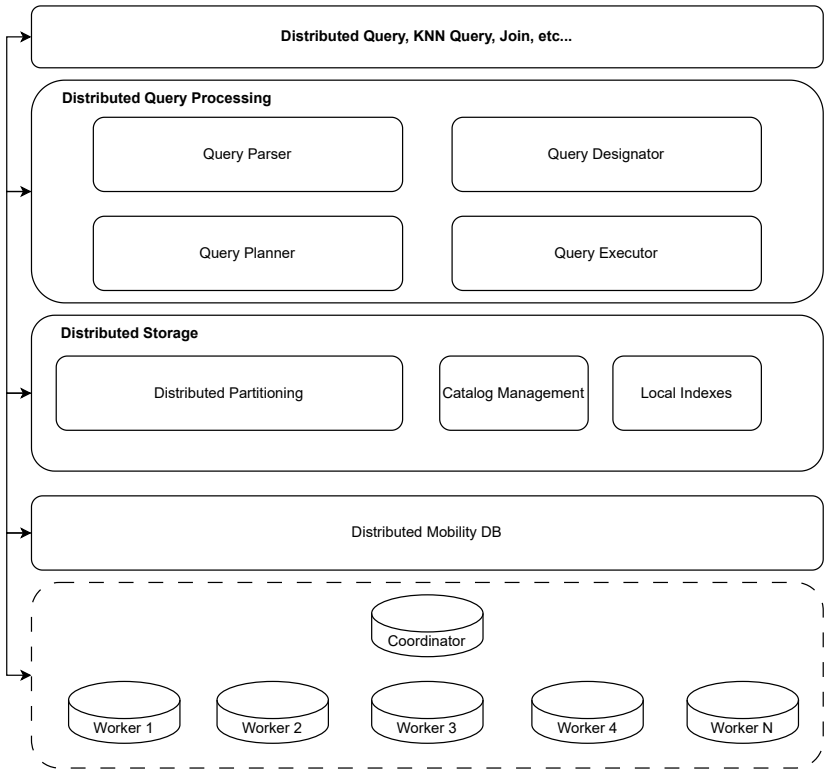

Figure 2.7: Distributed Mobility DB architecture. Both partitioning and indexing work at the distributed storage logical level. Taken from [1].

TrajSpark [38] employs both local and global indexing techniques to efficiently process trajectory queries. It features adaptive partitioning that dynamically recalculates data distribution as new data batches are ingested. Initially, the data points are partitioned using a KD-Tree structure, where each trajectory index is mapped locally to a leaf node generated by the KD-Tree. Subsequently, these mapped indexes are indexed locally, following a similar procedure to the one outlined in [35]. This involves first partitioning the indexes by their temporal attributes, then further subdividing them based on spatial characteristics, and finally constructing a B-tree over these levels to facilitate efficient querying.

It is important to recognize the close relationship between partitioning and indexing. As seen before, partitioning involves dividing a large dataset into smaller, more manageable segments based on certain criteria, such as time or spatial location. Indexing, on the other hand, involves creating a data structure that allows for quick retrieval of data points based on specific attributes. For example, consider a transportation dataset containing GPS logs of vehicles. By partitioning the data by time (e.g., daily logs) and then indexing each partition by spatial location (e.g., city), one can quickly retrieve all logs for a particular day in a specific city. This dual strategy of partitioning and indexing not only organizes the data more effectively but also enhances query performance by reducing the search space and enabling faster data access.

This relationship is explored in Chapter 3, particularly in the *BerlinMOD experiments*, where the partitioning ids are interpreted as an artificial index and are employed to optimize data processing and query execution. By leveraging these strategies, the system can handle large-scale mobility datasets more efficiently, providing rapid access to relevant data subsets.

Dita [41] employs a particularly intriguing method for partitioning trajectories, utilizing a two-level strategy. In the initial level, trajectories are bucketed based on their starting instant, ensuring that trajectories commencing at similar positions are colocated. Subsequently, in the second level, these bucketed trajectories are further partitioned according to their ending instant, thereby grouping trajectories with similar start and end points. This approach effectively organizes trajectories with comparable spatial and temporal characteristics. However, a potential limitation of this method is that it might not adequately account for trajectories that significantly diverge between their starting and ending points. For instance, consider two trajectories that originate from the same location but end at vastly different destinations; the initial bucketing would group them together despite their divergent paths, potentially leading to inefficiencies in data processing and analysis.

2.4 Final Remarks

This chapter encompasses the essential theoretical concepts and pertinent literature necessary for comprehending two primary aspects: first, the challenge of effectively processing mobility data within a distributed environment; and second, the proposed solution outlined in Chapter 3. Initially, we introduced fundamental concepts to establish a mobility data model, subsequently demonstrated and elucidated through MEOS, a spatiotemporal data management library. The geese example served as a practical illustration of the usage and instantiation of various mobility data types and functions.

Furthermore, we reviewed relevant partitioning techniques. Beginning with general methods such as horizontal and vertical partitioning, we demonstrated that these approaches are insufficient for handling spatiotemporal mobility data. This section also presented techniques and literature pertinent to partitioning these specific data types. The discussion concluded with an examination of PIP queries, which are crucial for understanding the *BerlinMOD* experiments discussed in Section 3.

Additionally, a comprehensive overview of key database management systems for mobility data was provided. We first analyzed MobilityDB from a distributed database perspective, followed by an introduction to Spark as a system. The Section concluded with a discussion of notable DBMS systems within Hadoop and Spark ecosystems that specialize in mobility or spatial data.

Understanding these concepts is vital, as they are interrelated in the context of managing mobility data. For example, effective mobility data partitioning requires an understanding of the attributes or properties commonly used for partitioning. Likewise, without familiarity with the data model or the specific use case, selecting the appropriate framework to implement a solution can become a complex endeavor.

Having established this foundational knowledge, we now proceed to explore the proposed solution for a distributed mobility data system, operating on top of Spark, capable of efficiently processing and distributing mobility data: Mobility Spark.

3

MobilitySpark

When the solution is simple, God is answering.
— Albert Einstein

Thus far, we have delved into the theoretical underpinnings and practical considerations of utilizing mobility data, alongside implementing robust big data solutions such as Hadoop and Spark for its efficient management. Our discussions have included in-depth explorations of the MobilityDB and MEOS frameworks.

It is evident that integrating a Big Data Management System with a specialized Mobility Data framework facilitates efficient and seamless analysis, enabling Data Scientists to effectively study and assess urban traffic patterns. Moreover, such an integration empowers Data Engineers to construct streamlined data pipelines that are not only faster and easier to process but also more comprehensible. Additionally, the development of data products like business dashboards and machine learning models benefits significantly from a fully integrated solution, such as the PyMEOS/PySpark binding proposed in this thesis.

Initially, we will explore the rationale for choosing the PyMEOS/PySpark integration over the JMEOS/Spark alternative, discussing the benefits and constraints of each option. Subsequently, we will address the integration of complex mobility data types from PyMEOS into PySpark, examining both its advantages and limitations. Following this, we will delve into the critical task of implementing partitioning schemes that enhance data processing efficiency through a use case with flight mobility data, *OpenSky*. The section will conclude with a benchmark analysis of the *BerlinMOD* queries [43] within this framework, providing insights into the performance enhancements achieved through our technical choices.

The codebase for this project can be followed through the project repository[1]. The experimental results can also be reviewed there.

[1] `https://github.com/Action52/MobilityPySpark`

3.1 Extending PyMEOS into PySpark

As studied in the previous chapter, the MEOS environment allows to create bindings in different programming languages or frameworks. In the context of Spark, this gives two options: Spark/JMEOS or PySpark/PyMEOS, each with a set of advantages and disadvantages. These are discussed in Table 3.1.

Advantages	
PyMEOS/PySpark	1. Facilitates easy prototyping. 2. Integrates seamlessly with Python's extensive package ecosystem. 3. Versioning consistent with MEOS. 4. High maintenance and regular updates.
JMEOS/Spark	1. Allows complex configurations using Spark's Catalyst Optimizer. 2. Native Java implementation facilitates integration with Spark's JVM.
Disadvantages	
PyMEOS/PySpark	1. Limited configuration capabilities; Catalyst Optimizer cannot be utilized. 2. Python functions can be opaque to Spark's JVM, potentially leading to performance issues.
JMEOS/Spark	1. Relies on an outdated version of MEOS; lower maintenance. 2. Lacks the comprehensive environment offered by Python. 3. API may be challenging for complex data configurations.

Table 3.1: Comparison of PyMEOS/PySpark and JMEOS/Spark.

It is evident that the current state of JMEOS introduces significant limitations to the system: its lack of regular maintenance and dependency on an outdated version of MEOS are major drawbacks to consider. While JMEOS facilitates the utilization of Spark's powerful Catalyst optimizer, allowing for complex configurations, it is hindered by its less friendly API and the absence of the rich Python environment, which limits user accessibility and flexibility. On the other hand, PyMEOS offers advantages such as seamless integration with Python's extensive package ecosystem and a versioning that is consistent with the latest MEOS, ensuring that users have access to the most current features and updates. Furthermore, PyMEOS is highly maintained, which contributes to its reliability and the potential for ongoing development. Although PyMEOS and PySpark may not fully leverage Spark's Catalyst optimizer, they still enable the creation of a robust, introductory binding that serves as an effec-

tive proof-of-concept. This setup not only proves viable for initial integrations but also lays the groundwork for future, more intricate integrations that could be developed within Spark. Therefore, we opt to implement a solution using PyMEOS/PySpark, with the anticipation of eventually extending these capabilities into JMEOS/Spark. This strategic choice allows us to harness immediate benefits while keeping the door open for leveraging JMEOS's full potential in subsequent phases of development.

With the choice of PyMEOS/PySpark as a framework, we can now explain the definition of the PyMEOS binding into PySpark. Under the context of this project, we will name the PyMEOS/PySpark binding as *MobilitySpark*.

3.1.1 System Architecture

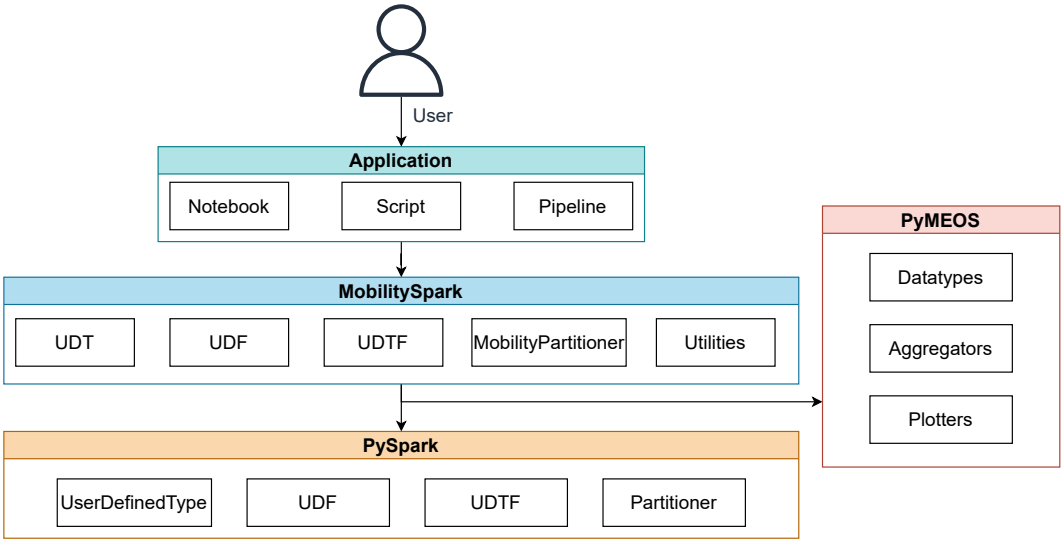

Figure 3.1: MobilitySpark architecture.

Figure 3.1 illustrates the general architecture of the proposed system, which integrates modules from both the PySpark and PyMEOS layers into the MobilitySpark layer. This integration forms the backbone of the application layer, which is accessed by end-users such as Data Scientists, Data Engineers, and Business Experts. These professionals utilize the system to develop a variety of end products, including dashboards, automated scripts, robust data pipelines, and interactive Jupyter notebooks, adjusted to their specific operational needs and analytical tasks.

MobilitySpark's design segregates its functionality into distinct modules that adhere to best practices in software design, enhancing usability and maintainability. This section outlines the initial set of modules—UDT, UDF, UDTF, and Utilities—each designed to streamline operations within the MobilitySpark framework. The User-Defined Types (UDT) module enables the creation of custom PyMEOS data types, making them compatible and readable within the PySpark environment. User-Defined Functions (UDF) encapsulate MEOS functions and support the integration of custom operations that involve MEOS function calls, facilitating complex data

manipulations. Concurrently, User-Defined Table Functions (UDTF) provide mechanisms for generating Spark tables or dataframes from MEOS-based functions. Additionally, the Utilities module incorporates supplementary helper functions that enhance the functionality and ease-of-use of MobilitySpark. The detailed discussion of the Partitioning module, which plays a crucial role in optimizing data processing and management, is reserved for Section 3.2.

3.1.2 User-Defined Types (UDT)

The core functionality that PySpark must manage is the ability to read complex mobility data types, such as those discussed in Chapter 2. PyMEOS provides classes that represent these data types, but there are challenges to overcome. Firstly, since these classes are written in Python, they must be made interpretable to Spark's JVM. Secondly, it is necessary to define to PySpark which base type from Spark each class represents, and how to serialize and deserialize these types effectively. The solutions to these challenges are depicted in Figure 3.2.

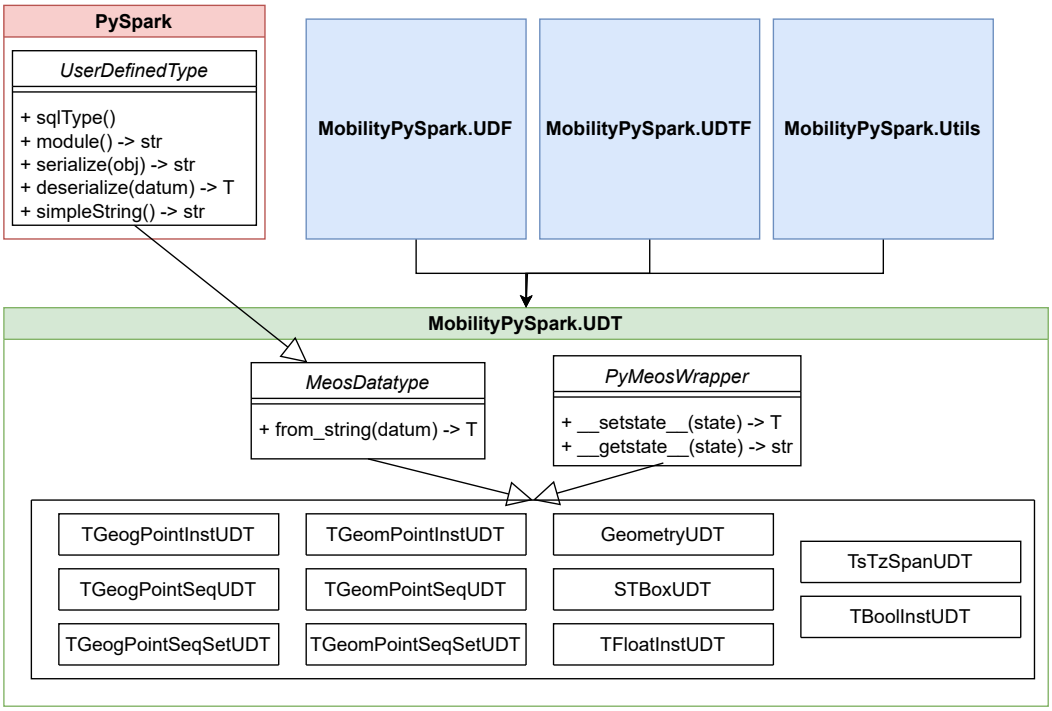

Figure 3.2: MobilitySpark UDT module class diagram.

We begin by outlining the requirements to create a user-defined type (UDT) in PySpark. PySpark's API offers a `UserDefinedType` class which includes methods to encapsulate a complex data type. Any class inheriting from `UserDefinedType` must implement the following key functions to function correctly within the PySpark framework:

1. `sqlType()`: This method must return the Spark SQL data type that the UDT wraps. In MobilitySpark, all UDTs are mapped to `StringType` to facilitate representation of PyMEOS classes within PySpark.

2. `serialize(obj)`: This function should define the custom serialization logic. For MobilitySpark, it is designed to serialize just the string representation of the object.

3. `deserialize(datum)`: This function should define the custom deserialization logic. In MobilitySpark, this function invokes the UDT's `from_string()` method, which is tailored for each UDT.

4. `simpleString()`: This method returns a string used internally to identify the data type succinctly.

5. `module()`: This should return a string indicating the Python module from which the UDT originates.

The `MeosDatatype` class, inheriting from `UserDefinedType`, serves as the foundational data type in MobilitySpark. All concrete UDTs such as `STBoxUDT`, `TBoolInstUDT`, and `TFloatInstUDT` derive from this class. Additionally, the `PyMeosWrapper` class is implemented to handle the `__setstate__` and `__getstate__` methods. These methods are essential because they are invoked by Spark's serializer during read and write operations on the UDTs. It is important to note that in certain operations, the Python *pickle* module calls the `__getstate__` and `__setstate__` methods to serialize the data, as opposed to the `serialize` and `deserialize` functions defined in `UserDefinedType`. Thus, each UDT in MobilitySpark must inherit from two classes: `MeosDatatype` and its corresponding `PyMeosWrapper`, to ensure full compatibility and functionality within the PySpark framework.

```
schema = StructType([
    StructField("icao24", StringType()),
    StructField("PointInst", TGeogPointInstUDT())
])
```

```
+------+--------------------+
|icao24|           PointInst|
+------+--------------------+
|34718e|POINT(1.922973632...|
|ac6364|POINT(-85.5262662...|
|406471|POINT(1.838302612...|
|a04417|POINT(-83.4583702...|
|c04aa1|POINT(-79.3079393...|
+------+--------------------+
```

Figure 3.3: Example MobilitySpark UDT.

Finally, this module is consumed by other modules within MobilitySpark. In particular, as it will be reviewed, the UDF and UDTF modules exploit all these available data types with custom functionalities. Implementing these functions ensures that the PyMEOS classes are fully compatible and operational within the PySpark ecosystem, thereby enhancing the functionality and usability of the MobilitySpark framework[2].

[2] The `UDTDemo.ipynb` notebook, available in the project repository, showcases a simple example of read/write operations with MobilitySpark UDTs.

3.1.3 User-Defined Functions (UDF)

A User-Defined Function (UDF) in Spark allows users to extend the native capabilities of Spark by implementing custom behaviors within Spark pipelines. These are particularly valuable when the existing Spark or PySpark APIs fall short in performing specific transformations. In the MobilitySpark framework, UDFs serve as direct interfaces to PyMEOS functions, enabling the integration of complex functionalities that might not be natively supported by PyMEOS but still rely on its operations.

However, UDFs in PySpark and Spark are treated as black boxes, meaning that Spark's optimizer is unable to apply rule-based optimizations to these functions. This limitation can introduce significant performance bottlenecks. Moreover, PySpark's UDFs execute within a Python process external to Spark's JVM; the results must be encoded and transmitted to the JVM, then decoded back, adding overhead. An alternative approach to mitigate some of these inefficiencies involves using Pandas UDFs, which employ Pandas to parallelize execution internally. Although promising, this approach is not covered in this thesis and remains an area for future exploration. The usage of Pandas UDFs in MobilitySpark is briefly discussed in 3.5.3. For the purposes of this project, the acceptance of the inherent drawbacks of simple UDFs is necessary when they are employed as wrappers for PyMEOS functions.

The architecture of a UDF in MobilitySpark is illustrated in Figure 3.4. This example demonstrates the implementation of a PyMEOS function within a PySpark UDF framework. The `@F.udf` decorator specifies that the function is a UDF and `returnType` defines the expected return data type of the function. It is crucial to initiate every MobilitySpark UDF that interacts with PyMEOS operations with the `pymeos_initialize(utc)` command. This initialization is mandatory in any MEOS application due to the extensive parallelization in Spark, which often triggers multiple new processes that each require initialization of PyMEOS. Here, the function returns a `STBox` object, which are represented in MobilitySpark as `STBoxUDT`.

```
@F.udf(returnType=STBoxUDT())
def point_to_stbox(tpoint: TPoint, utc: str = "UTC") -> STBox:
    """
    Returns the STBox associated to a TPoint.
    :param tpoint:
    :param utc:
    :return:
    """
    pymeos_initialize(utc)
    return tpoint.bounding_box()

spark.udf.register("point_to_stbox", point_to_stbox)
```

Figure 3.4: Backbone of a MobilitySpark UDF definition.

A UDF defined with the decorator can be directly passed to a PySpark DataFrame. For instance, the code execution in Figure 3.5.

To make a function accessible from Spark SQL, it must first be registered with the `SparkContext`, which represents the Spark instantiation within a PySpark application. This registration is accomplished using the `udf.regi-`

```
df.withColumnRenamed("PointStr", "PointInst") \
    .withColumn("STBox", point_to_stbox("PointInst")).show(3)

    +------+--------------------+--------------------+
    |icao24|           PointInst|               STBox|
    +------+--------------------+--------------------+
    |34718e|POINT(1.922973632...|SRID=4326;GEODSTB...|
    |ac6364|POINT(-85.5262662...|SRID=4326;GEODSTB...|
    |c04aa1|POINT(-79.3079393...|SRID=4326;GEODSTB...|
    +------+--------------------+--------------------+
```

Figure 3.5: Applying a UDF in MobilitySpark Example.

`ster` command, as illustrated in Figure 3.4. This step ensures that the function can be seamlessly integrated and utilized within the broader Spark SQL query environment.

For this project, a diverse array of User-Defined Functions (UDFs) have been developed. Some of these functions serve as direct wrappers to existing PyMEOS functionalities, thereby extending their utility within the PySpark ecosystem. Others incorporate additional operations to fulfill specific requirements of the MobilitySpark framework. A detailed enumeration of these functions, along with their descriptions and use cases, is available in Appendix 4.5, providing a resource for users to explore and utilize according to their project needs.

Within the MobilitySpark architecture, all UDFs are organized under the UDF module, enabling easy integration and deployment within user applications. For instances where automated registration of these functions is necessary, the Utilities module offers a `register_udfs_under_spark_sql(spark)` function. This utility function simplifies the process of registering all UDFs with a given `SparkContext`, ensuring they are available for SQL operations without requiring manual intervention for each function. This streamlined approach facilitates a more efficient setup and operational workflow, enhancing the user experience and system performance.

3.1.4 User-Defined Table Functions (UDTF)

User-Defined Functions (UDFs) are particularly useful in PySpark for column-wise, one-to-one transformations, offering remarkable extensibility to users. However, UDFs are not suited for operations that require outputs different from the inputs in size, such as *aggregations* or *explode* functions that generate a variable number of rows based on the input. To handle aggregations, PySpark employs User-Defined Aggregation Functions (UDAFs), which, while important, are not implemented in this project.

Another powerful extension of PySpark's functionality are the User-Defined Table Functions (UDTFs). Unlike UDFs, UDTFs can generate multiple output rows from a single input row, effectively working as generators. For a given input i, a UDTF can produce a number of output rows o, where o may differ from i. This capability makes UDTFs ideal for tasks that require expanding or transforming input data into more complex outputs.

UDTFs are structured distinctly from UDFs, as they must be encapsulated within classes. The functionality of a UDTF is executed within the `eval` method of the class. Since UDTFs output a new table, a schema must be defined and passed as the `returnType` parameter, specifying the structure of the output table. The provided example in Figure 3.6 illustrates a UDTF that takes a list of geographic tiles and outputs a new table row using

MobilitySpark's `STBoxUDT` data type for each tile, along with its identifier. Note how, as in UDFs, the `pymeos_initialize` instruction is enforced.

```
@F.udtf(returnType=StructType([
    StructField("tileid", IntegerType()),
    StructField("tile", STBoxUDT())
]))
class GridUDTF:
    def eval(
            self,
            tilesstr: List[str],
            col_dict={},
            utc_time: str = "UTC"
    ):
        pymeos_initialize(utc_time)
        for tileid, tile in enumerate(tilesstr):
            yield tileid, STBox(tile)
```

Figure 3.6: Simple User-Defined Table Function (UDTF) in MobilitySpark.

In MobilitySpark, the UDTF module is heavily used to implement the functionality of Partitioners, as will be reviewed in Sections 3.2 and 3.5.

3.1.5 Utilities

The Utilities module in MobilitySpark features a collection of helper functions that are essential for the operational efficiency of the project and can be utilized by users as needed. A pivotal function within this module is the `udt_append` function, designed to facilitate integration between the UDT module and PyMEOS. This function employs monkey patching—a technique where modifications are made to a class or module at runtime without altering the source code—to assign a `__UDT__` variable to PyMEOS classes. For example:

```
TGeogPointInst.__UDT__ = TGeogPointInstUDT()
```

This `__UDT__` variable is essential as it is appended to all PySpark User Defined Types, effectively linking them with corresponding PyMEOS classes. Monkey patching allows for dynamic adjustments to the class behaviors, which is crucial for maintaining compatibility and synchronization between MobilitySpark and PyMEOS without modifying their underlying source codes directly.

It is critical to execute the `udt_append` function immediately after initiating the `SparkContext` and before deploying any MobilitySpark functionalities. This ensures that all modifications are in place before the framework is actively used in a PySpark project. The standard procedure for initializing a project using MobilitySpark is depicted in Figure 3.7. Note how the `pymeos_initialize`, from PyMEOS, as well as some functions from the Utilities module (`udt_append`, `register_udfs_under_spark_sql`, `register_udfs_under_spark_sql`), are called.

So far, the different core modules in the system architecture of MobilitySpark have been explored. The remaining module, Partition, is explored in Section 3.2.

```
# Initialize PyMEOS
pymeos_initialize("UTC")

# Initialize a Spark session
spark = SparkSession.builder \
    .appName("Typical Init in MobilitySpark") \
    .master("local[3]") \
    .getOrCreate()

# Append the UDT mapping to the PyMEOS classes
udt_append()

# Register the UDFs in spark
register_udfs_under_spark_sql(spark)

# Register the UDFs in spark
register_udtfs_under_spark_sql(spark)
```

Figure 3.7: Example initialization in MobilitySpark.

3.2 MobilitySpark Partitioning Framework

The philosophy of horizontal scaling needs that data be efficiently distributed across the system. As discussed in Chapter 2, a mobility data application significantly benefits from incorporating *spatiotemporal awareness*. This involves the system considering the proximity of data in both space and time to optimize operations. For instance, by grouping data points that are geographically and temporally close, the system can reduce the overhead of data shuffling and improve query performance. This approach enhances the efficiency of operations such as trajectory analysis, real-time tracking, and event detection, thereby leveraging the full potential of systems such as Spark.

Achieving spatiotemporal awareness fundamentally relies on effective data partitioning. In the context of mobility data, this entails executing partitioning routines that group data based on their spatiotemporal characteristics. This section will present the Partition module of MobilitySpark, detailing the partitioning techniques employed and the framework used to integrate these schemes into MobilitySpark applications [3].

3.2.1 Partitioning Framework Overview

Figure 3.8 presents the classes used to model the Partition module in MobilitySpark, and how this module interacts with the UDTF module, and then with the Application layer.

First, we present the `MobilityPartitioner` class, which inherits from PySpark's `Partitioner` class. Even though no overriding is enacted, the idea is to extend the functionality from the Partitioner to MobilitySpark. The MobilityPartitioner implements, in general, two helper functions: `as_spark_table` and `plot`, which are used to be parsed into a spark table through a UDTF and to plot the tile system, accordingly. MobilitySpark incorporates

[3] As a complement to Section 3.2, the `PartitioningSchemesWithPyMEOS.ipynb` notebook in the project repository provides a practical approach.

natively 4 options to partition MobilityData: `RegularPartitioner`, `KDTreePartitioner`, `AdaptiveBinsPartitioner`, and `BisectiveKMeansPartitioner`. The KD-Tree implementation includes in-memory and Spark-based options, while the Adaptive Bins implementation includes in-memory, Spark-based, and a MapReduce approximation strategy.

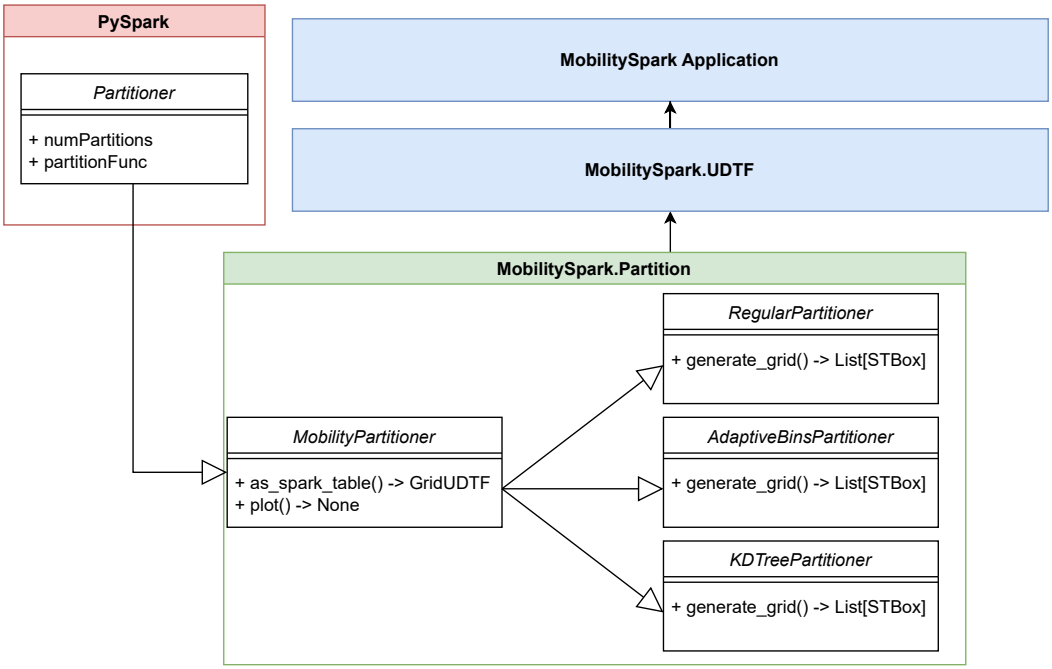

Figure 3.8: MobilitySpark partitioning framework.

When a `MobilityPartitioner` is instantiated, an internal method, typically `generate_grid`, is invoked. This method is responsible for tiling the provided space, which should encompass the spatiotemporal bounds of the dataset, represented by an `STBox` object. The `generate_grid` method processes this `STBox` and returns a list of tiles that represent the spatiotemporal partitions of the data.

These tiles can then be retrieved using the UDTF module to generate a new table with the partitioned mobility data. MobilitySpark utilizes a specialized UDTF, `BasePartitionUDTF`, to create the corresponding Spark table of partitioned data. This class accepts three parameters: `response_extra_cols`, a list of strings specifying additional columns to include in the new table; `check_function`, the topological function, typically from PyMEOS, that determines if a point belongs to a partition; and `return_full_traj`, which indicates whether the trajectories should be split by partitions. By default, the `at` topological function is used to ascertain if a point is part of the partition.

A basic `PartitionUDTF` can be created as follows:

```
partition_schema = StructType(
    [
```

```
        StructField("movingobjectid", StringType()),
        StructField("tileid", IntegerType()),
        StructField("movingobject", TGeomPointSeqSetUDT()),
    ]
)

@F.udtf(returnType=partition_schema)
class PartitionUDTF(BasePartitionUDTF):
    def __init__(self):
        check_function = lambda traj, tile: traj.at(tile)
        super().__init__(
            response_extra_cols=[],
            check_function=check_function,
            return_full_traj=True
        )

    def eval(self, row: Row):
        for val in super().eval_wrap(row):
            yield val

spark.udtf.register("PartitionUDTF", PartitionUDTF)
```

In this example, the `PartitionUDTF` is a straightforward UDTF function that uses the `at` function to evaluate the temporal points at tiles. Given an existing grid table in Spark, the `PartitionUDTF` can be called to perform its evaluation:

```
SELECT *
FROM PartitionUDTF(
    TABLE(
        SELECT
            seqId AS trajectory_id,
            PointSeq AS movingobject,
            (SELECT collect_list(tile) FROM grid) AS tiles,
            (SELECT collect_list(tileid) FROM grid) AS tileids
        FROM trajectories
    )
)
```

Here, the *trajectories* table is used as input for the `PartitionUDTF`. For each record in the dataframe, the UDTF will evaluate using the default function on each of the tiles passed within the SQL query. The output would look as follows, note the name of the columns follow the schema defined:

```
+---------------+------------+----------------------+
| movingobjectid|      tileid|          movingobject|
+---------------+------------+----------------------+
|   274877906945|          15|  {[POINT(8.01 48.9...|
|   274877906945|          16|  {[POINT(8.013 48....|
|   274877906945|          17|  {[POINT(8.016 48....|
+---------------+------------+----------------------+
```

Assuming the grid is reasonably small, the tiles can be collected in a Spark SQL `SELECT` statement and passed as arguments to the `PartitionUDTF`. The UDTF will iterate over each row, partitioning the data and creating a new table.

3.2.2 Calculating the dataset bounds

An important step required by the partitioners in MobilitySpark is to precompute the dataset spatiotemporal bounds. This step ensures that the partitioner will split the tiles considering the whole space available. To perform the operations quickly, a classic MapReduce function is proposed in Algorithms 3.1 and 3.2.

Algorithm 3.1 Bounds Calculation Map Function

Require: An iterable of partition rows R, each containing a spatiotemporal column, a column name *colname*, a time zone *utc*
Ensure: A list of spatiotemporal bounding boxes T
1: Start a new spatiotemporal extent aggregation $\mathcal{A} \leftarrow \varnothing$
2: **for all** row $r \in R$
3: **if** r is valid
4: $\mathcal{A} \leftarrow \mathcal{A} + r_{colname}$
5: **end if**
6: **end for**
7: **if** $\mathcal{A} \neq \varnothing$
8: Return $\mathcal{A}$
9: **else**
10: Return $\varnothing$
11: **end if**

The provided pseudocode outlines a MapReduce approach for calculating the spatiotemporal bounds of a dataset. This approach is divided into two primary functions: the map function and the reduce function.

The `bounds_calculate_map` function is responsible for processing a partition of rows, each containing a series of spatiotemporal points represented as a sequence. The function initializes PyMEOS with the specified UTC time zone and begins an aggregation process. For each row in the partition, if the row is valid, it extracts the spatiotemporal sequence from the specified column and adds it to the aggregator. After processing all rows, the function returns the aggregated spatiotemporal bounding box or `None` if an error occurs. This function should be applied to a PySpark RDD using the `mapPartitions` function. Unlike the `map` function, which processes

Algorithm 3.2 Bounds Calculation Reduce Function

Require: Two spatiotemporal bounding boxes B_1 and B_2, a time zone *utc*
Ensure: The combined spatiotemporal bounding box B
1: Initialize $B \leftarrow \varnothing$
2: **if** $B_1 \neq \varnothing$ and $B_2 \neq \varnothing$
3: $\quad B \leftarrow B_1 + B_2$
4: **else if** $B_1 = \varnothing$ and $B_2 \neq \varnothing$
5: $\quad B \leftarrow B_2$
6: **else if** $B_1 \neq \varnothing$ and $B_2 = \varnothing$
7: $\quad B \leftarrow B_1$
8: **end if**
9: Return B

one element at a time, `mapPartitions` operates on entire partitions of data. This approach is more efficient in this context because it reduces the overhead of repeated function calls, allows for better use of resources within each partition, and minimizes the amount of data shuffling between nodes. Consequently, `mapPartitions` can significantly enhance the performance of large-scale spatiotemporal data processing.

The `bounds_calculate_reduce` function is designed to combine two spatiotemporal bounding boxes into a single bounding box that encompasses the spatiotemporal extents of both. The function initializes the spatiotemporal library with the specified UTC time zone and checks the validity of the input bounding boxes. If both bounding boxes are valid, it combines them. If only one bounding box is valid, it returns that bounding box. The resulting bounding box is then returned. If no valid combination can be made, the function returns `None`.

This MapReduce approach allows for efficient and scalable computation of spatiotemporal bounds by distributing the computation across multiple partitions and then combining the results. The functions can be imported from the Utilities module by the user with MobilitySpark:

```
from pysparkmeos.utils.utils import bounds_calculate_map, bounds_calculate_reduce
trajectories.rdd.mapPartitions(bounds_calculate_map).reduce(bounds_calculate_reduce)
```

Here, the `trajectories` RDD uses the proposed framework to calculate the overall spatiotemporal bounds of the dataset, producing a STBox as output, for instance:

```
STBOX XT(
    ((-125.8,25.1),(-75,52.9)),[2022-06-27 00:00:00+00, 2022-06-27 00:00:40+00])
```

3.3 Partitioning Schemes in MobilitySpark

Now that we have outlined the overall process of partitioning mobility data within MobilitySpark, we will delve into the native partitioning schemes included in the framework. After calculating the global bounds of the dataset, the user must choose a partitioning scheme. MobilitySpark offers implementations for 4 partitioning techniques: *Regular Grid*, *KD-Tree*, *Adaptive Bins*, and *Bisective K-Means*.

It is important to emphasize that MobilitySpark adheres to a *simplicity-first* philosophy. This approach ensures that the recommended partitioning strategies for mobility data are straightforward enough to integrate seamlessly with the horizontal-scaling capabilities of Spark. More advanced methods, such as those reviewed in Chapter 2, are generally discouraged, particularly in scenarios where practical considerations are imperative. Complex partitioners employing techniques like machine learning are assumed to be less common and are recommended only for specific, demanding use cases, and are not covered in this thesis.

In the following sections, we will review the implemented partitioning strategies in MobilitySpark. This discussion will cover the individual implementation details of each partitioner, while Section 3.4 will provide a comprehensive comparison of these methods using *OpenSky* data.

3.3.1 Regular Grid Partitioner

The most basic partitioner in MobilitySpark is the *Regular Grid* Partitioner. The idea is to split the spatiotemporal bounds of the dataset into equally-sized tiles. Creating a regular grid in MobilitySpark is depicted in the code below. Furthermore, Algorithm 3.3 shows how the `GridPartition` class works internally to create the tiles.

```
gp = GridPartition(cells_per_side=4, bounds=bounds)
grid = gp.as_spark_table()
```

```
+------+--------------------+
|tileid|                tile|
+------+--------------------+
|     0|STBOX XT(((-125.8...|
|     1|STBOX XT(((-125.8...|
|     2|STBOX XT(((-125.8...|
|     3|STBOX XT(((-125.8...|
+------+--------------------+
```

When creating the class, the user must specify the number of cells per dimension and the `STBox` representing the bounds of the dataset. The total number of output tiles will therefore be calculated as n^d, where n is the number of cells per side and d is the number of dimensions (including the time dimension). For example, a mobility dataset with dimensions x, y, t and 5 partitions per side would result in $5^3 = 125$ equally-sized tiles.

The `as_spark_table` function, which is common to all `MobilityPartitioner` classes in MobilitySpark, triggers internally a UDTF which results in the partitioned boxes being loaded into a Spark DataFrame.

The spatial representation of the grid can easily be plotted using MobilitySpark's wrapper to the PyMEOS `STBox.plot` function. MobilitySpark ensures this function is called for each tile in the grid. Figure 3.9 shows the *xy* projection of the partition tiles using MobilitySpark. Naturally, some tiles will overlap, as the slice over the time dimension is not projected.

Algorithm 3.3 Regular Grid Partitioner Algorithm

Require: Spatiotemporal bounds B represented by `STBox`, number of cells per dimension n
Ensure: A list of equally-sized spatiotemporal tiles T

1: Initialize an empty list $T \leftarrow \varnothing$
2: Compute the number of dimensions d from B
3: Calculate total number of tiles $N \leftarrow n^d$
4: **for** each dimension i in d
5: Calculate the size s of each tile in dimension i: $s \leftarrow \frac{B(i)_{max} - B(i)_{min}}{n}$
6: Create a list of split indices for dimension i based on s
7: **end for**
8: **for** each possible combination of tile indices
9: Create a new tile t with the calculated sizes for each dimension
10: Add t to T
11: **end for**
12: Return T

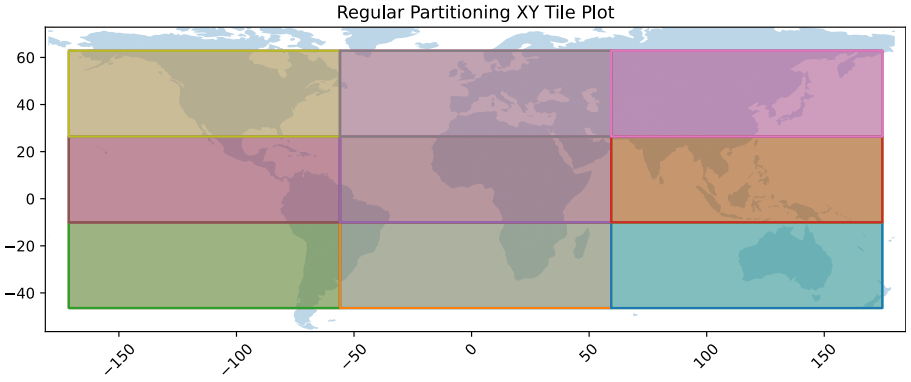

Figure 3.9: Regular Grid Partitioner XY projection using MobilitySpark.

3.3.2 KD-Tree Partitioner

MobilitySpark includes two different implementations for a KD-Tree Partitioner: `KDTreePartition` and also `KDTreePartitionSpark`. The former computes the grid by loading the data points, or a subset of them, into the master PySpark node, while the latter leverages Spark as an engine to perform the computation. Despite the differences in execution, the underlying concept remains the same.

As discussed in Chapter 2, a KD-Tree is an algorithm often used for indexing and partitioning data with multiple dimensions. In MobilitySpark, this is achieved by exploding each trajectory data point into an instant. This means that if each trajectory is saved as a `TGeomPointSeq` or `TGeomPointSeqSet` instance, the partitioner will retrieve each `TGeomPointInst` object and use it for measurement. Once the instants are retrieved, the *median* instant is calculated based on one of the dimensions x, y, t, or z. The data points are then split by this median, and the recursion is applied to each new group of data points with a new spatiotemporal bound considering the median as a boundary. Algorithm 3.4 illustrates this process in MobilitySpark.

The partitioning process continues until the tree reaches the specified maximum depth or until all branches run out of data. Given a maximum depth d_{max}, the expected number of tiles T is given by $T_{length} = 2^{d_{max}}$, as it is a binary tree.

Another key aspect of the KD-Tree partitioning in MobilitySpark is the alternating split dimensions in each iteration. For instance, starting with the x dimension, it will then split by y and subsequently by t. This order can be configured by the user. While alternating dimensions is assumed to be a reasonable approach, the efficacy of this method as the optimal splitting strategy is not evaluated within this project.

A `KDTreePartitioner` or a `KDTreePartitionerSpark` can be instantiated as follows:

```
kd = KDTreePartition(pointsample, ['x', 'y', 't'], bounds, 4, "UTC")
kd_spark = KDTreePartitionSpark(
    spark, instants, 'instants', 'instant',
    'seqId', bounds, ['x', 'y', 't'], 4, "UTC"
)
```

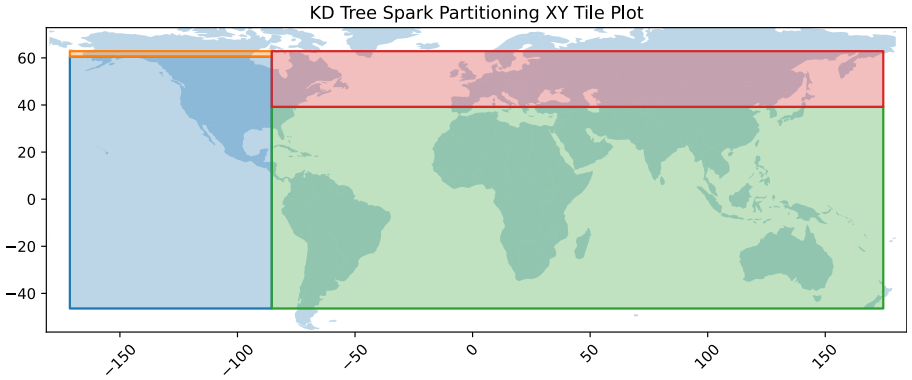

Figure 3.10: KD-Tree Partitioner XY projection using MobilitySpark.

The decision of which KD-Tree implementation to use is left to the user. `KDTreePartitioner` is recommended if the dataset is guaranteed to fit in the driver node. If not, an approximation can be achieved by passing a representative sample of the dataset to the partitioner.

Calling the `as_spark_table` method will load the tiles as a DataFrame into Spark, as demonstrated in the previous section. The `plot` method generates an image similar to the one shown in Figure 3.10.

The `KDTreePartitionerSpark` is recommended when the dataset is too large to fit in the driver node and sampling is not feasible. This implementation leverages PySpark DataFrames and RDDs to calculate the tree, implementing the same algorithm as the `KDTreePartitioner`. The main difference lies in its use of distributed computing resources, making it suitable for very large datasets. Calculating the median in a distributed dataset is particularly challenging. In Spark, Dataframes have no associated row number, and even assigning an identifier through the `monotonically_increasing_id` method doesn't ensure contiguous numbers through the partitions of the RDD/Dataframe. In `KDTreePartitionerSpark`, the median is approximated using the *median of medians* approach, where the median is first calculated within each partition, and then the intermediate results are aggregated at the driver to compute the final median. Under this context, the median calculation step in the

Algorithm 3.4 KD-Tree Partitioner Algorithm

Require: Trajectory points P, spatiotemporal bounds B represented by `STBox`, dimensions $dims$, current depth $depth$, max depth $depth_{max}$
Ensure: A list of spatiotemporal tiles T

$curdim \leftarrow dims[depth \% dims.length]$
$P \leftarrow \text{sort}(P, curdim)$
Initialize an empty list $T \leftarrow \varnothing$
if $depth = depth_{max}$ or $P.length = 0$
    $T \leftarrow T + [B]$
else
    $median \leftarrow P[\text{floor}(P.length/2)]$
    $B_l \leftarrow \text{bounds}(B, median, left)$
    $B_r \leftarrow \text{bounds}(B, median, right)$
    Split P into P_l and P_r based on $median$
    $left \leftarrow \text{kdtree}(P_l, B_l, dims, depth + 1, depth_{max})$
    $right \leftarrow \text{kdtree}(P_r, B_r, dims, depth + 1, depth_{max})$
    $T \leftarrow T + left$
    $T \leftarrow T + right$
end if
Return T

`KDTreePartitionerSpark` can be reviewed in Algorithm 3.5. This algorithm is easily parallelized through the partitions in PySpark, as done before, using `mapPartitionsWithIndex`:

```
rdd = instants_at.select(dim,"rowNo").rdd.mapPartitionsWithIndex(
    lambda i, x:KDTreePartitionSpark.partition_median_index(
        i,x,dim,rowcol='rowNo')
)
```

The Algorithm 3.5 is mapped to each partition of the dataset, and then, the local median is calculated. Afterwards, the partial results are collected and then the final median is approximated. The index of the original partition is kept for reference purposes.

All the previously mentioned operations using RDD/DataFrames are costly, and the recursive nature of Algorithm 3.4 makes it very expensive in time-restricted scenarios. Therefore, a recommendation to the final user is to assess the need of calculating the KD-Tree using the whole dataset, or leveraging it with a representative sample.

A comparison between `KDTreePartitioner` and `KDTreePartitionerSpark` is provided in Section 3.4.

3.3.3 Adaptive Bins Partitioner

A regular grid approach, as previously discussed, assumes that data is uniformly distributed across the spatiotemporal bounds of the dataset. An improvement to this approach is to consider the concentration of data points when constructing the grid.

Algorithm 3.5 Median of Medians `KDTreePartitionerSpark`

Require: A set of partition iterators P, current dimension dim
Ensure: The estimated median row of the RDD/DataFrame
1: Initialize $median \leftarrow \varnothing$
2: Initialize $medians_{partial} \leftarrow \varnothing$
3: **for all** (idx, p) in enumerate(P)
4: Initialize $values \leftarrow \varnothing$
5: **for** (i, row) in enumerate(p)
6: $median_{index} \leftarrow \text{round}(\frac{i}{2})$
7: $values \leftarrow values + [(row[dim], row[id])$
8: **end for**
9: $values \leftarrow \text{sort}(values, \text{by} = f(x) : x[0])$
10: $medians_{partial} \leftarrow medians_{partial} + [values[median_{index}]]$
11: **end for**
12: $median \leftarrow \text{calculate\_final\_median}(medians_{partial})$
13: Return $median$

The proposed `AdaptiveBinsPartitioner` adopts this philosophy. Instead of building uniform tiles over the bounds, it creates tiles based on the data concentration by performing splits over equally-distributed slices of data rather than uniform tiles. This results in tiles that are more *compact* compared to others. These splits can be performed step-wise per dimension.

In MobilitySpark, we introduce three partitioner implementations using the Adaptive Bins approach: `AdaptiveBinsPartitioner`, `AdaptiveBinsPartitionerSpark` and `ApproximateAdaptiveBins`. The first implementation runs the algorithm in-memory, while the second one leverages Spark as an engine. The `ApproximateAdaptiveBins` estimates the bin size using a MapReduce approach on the tiles.

In general, the Adaptive Bins algorithm is outlined in Algorithm 3.6. The algorithm first extracts all instants from any trajectory data type. In MobilitySpark, this means creating a set of `TPointInst` objects for each `TPointSeq` or `TPointSeqSet` in the dataset. Once the instants are extracted, the maximum is obtained to consider during the bounds calculation. The `ntile` function is then used to divide the sorted values into n equally distributed splits, called $bins_1$. For each split in $bins_1$, the data within each bin is sorted, and row numbers are assigned to each value. Finally, the new bounds for each bin are calculated by *pasting* the last value of the previous bin with the first value of the current bin. The resulting bounds are then assigned to a set of tiles T. In MobilitySpark, the three implementations of this algorithm perform this step once per dimension. As in the `RegularGridPartitioner`, splits occur in x, y, t, and/or z dimensions. Therefore, the number of tiles is calculated as n^{dims}.

The in-memory `AdaptiveBinsPartitioner` can be instantiated as follows:

```
agp = AdaptiveBinsPartitioner(
    bounds=bounds, movingobjects=pointsample, num_tiles=4,
    dimensions=['x', 'y', 't'], utc="UTC")
gridagp = agp.as_spark_table()
```

Algorithm 3.6 Adaptive Bins Algorithm

Require: A set of spatiotemporal trajectories D, spatiotemporal bounds B, current dimension dim, number of tiles n
Ensure: A list of equally-sized spatiotemporal tiles T

Initialize $T \leftarrow \varnothing$
if dim is spatial
    $values \leftarrow \text{explode}(\text{spatial\_values}(D, dim))$
else
    $values \leftarrow \text{explode}(\text{timestamps}(D, dim))$
end if
$max_{vals} \leftarrow \max(values)$
$bins1 \leftarrow \text{ntile}(n, \text{sort}(values))$
$bins2 \leftarrow \text{sort}((id, val, \text{rownumber}), \text{by} = \text{rownumber}) \; \forall \; bin \in bins1$
$new_{bounds} \leftarrow \text{sort}((val, \text{lead}(val, 1)), \text{by} = val) \; \forall \; val \in bins2, \; \text{rownumber} = 1$
for $nb \in new\_bounds$
    $T \leftarrow T + \text{calculate\_bounds}(B, nb)$
end for
Return T

This would create a grid of $4^3 = 64$ tiles, with 4 splits per dimension. The `as_spark_table` method works as described previously: it saves the grid as a Spark DataFrame. As with other in-memory methods, it is recommended to pass a sample of the trajectories or instants to avoid memory issues with the driver node.

In contrast, the `AdaptiveBinsPartitionerSpark` leverages Spark SQL to calculate the bins. It can be instantiated as follows:

```
sagp = AdaptiveBinsPartitionerSpark(
    spark, bounds, 'trajectories', 'PointSeq', num_tiles=4,
    dimensions=['x', 'y', 't'], utc="UTC"
)
```

This command requires the Spark instance, the dataset bounds, the name of the table to partition, and the column containing the spatiotemporal values. It also specifies the number of splits per dimension, the dimensions to consider, and the UTC configuration.

Both implementations of the Adaptive Bins algorithm have a drawback: they require additional memory from the driver node to compute certain operations. The in-memory Adaptive Bins loads all points or a sample into the driver, while the Spark-based version needs to perform intermediate `collect` operations, which are sent to the driver to estimate the result of the query.

In this project, we propose an approximation algorithm that trades some accuracy for speed in calculating the grid. The `ApproximateAdaptiveBins` leverages the use of MapReduce to estimate the bins. Here, each input partition of the Spark DataFrame is interpreted as a sample of the dataset, and the bin size is computed considering the average bin size calculated for each partition.

Algorithm 3.7 illustrates the first step. This step is passed through a `mapPartitions` instruction to Spark, resulting in a set of candidate bounds for each bin from each partition. The second step, shown in Algorithm 3.8, is applied with a `reduceByKey` function. It calculates the mean values of each pair of partial bins. This approach ensures that the bins from the mapped step will be combined in the reduce step, and that the mean value approximates the real split point.

The calculated bin bounds with the mean do not ensure contiguous split values throughout the mobility dataset. Therefore, a final step to make the bounds contiguous is triggered. In this step, the reduced bounds of each bin are collected and sorted, and the bounds between each bin are adjusted to be contiguous by selecting the maximum value of the previous bin as the adjusted starting point of the current bin. Finally, the bounding box of each bin is calculated based on these adjusted bins. This is all reflected on Algorithm 3.9.

Algorithm 3.7 Map Approximate Adaptive Bins Algorithm

Require: A set of spatiotemporal trajectories P, spatiotemporal bounds B, current dimension dim, number of tiles n
Ensure: A list of pairs $[(i, (bin_0, bin_1))] \forall bin \in bins_P$
1: $bins_P \leftarrow \text{AdaptiveBins}(P, B, dim, n)$
2: Initialize an empty list $values \leftarrow \varnothing$
3: **for all** (i, bin) in enumerate($bins_P$)
4: $\quad values \leftarrow values + [(i, (bin_0, bin_1))]$
5: **end for**
6: Return $values$

Algorithm 3.8 Reduce Approximate Adaptive Bins Algorithm

Require: A pair of tuples (bin_{A_0}, bin_{A_1}) and (bin_{B_0}, bin_{B_1})
Ensure: A mean pair $(mean_{min}, mean_{max})$
1: $mean_{min} \leftarrow \text{mean}(bin_{A_0}, bin_{B_0})$
2: $mean_{max} \leftarrow \text{mean}(bin_{A_1}, bin_{B_1})$
3: Return $(mean_{min}, mean_{max})$

Creating an `ApproximateAdaptiveBinsPartitioner` is done as follows:

```
aagp = ApproximateAdaptiveBinsPartitioner(
    spark=spark, df=trajectories, colname='PointSeq',
    bounds=bounds, num_tiles=4, dimensions=['x', 'y', 't'], utc="UTC"
)
```

This class requires the Spark instance, the DataFrame containing the trajectory data, and the column name where this data is stored. It also takes common parameters such as the number of tiles per dimension and the dimensions to consider during the split.

Section 3.4 provides a practical evaluation of these three implementations.

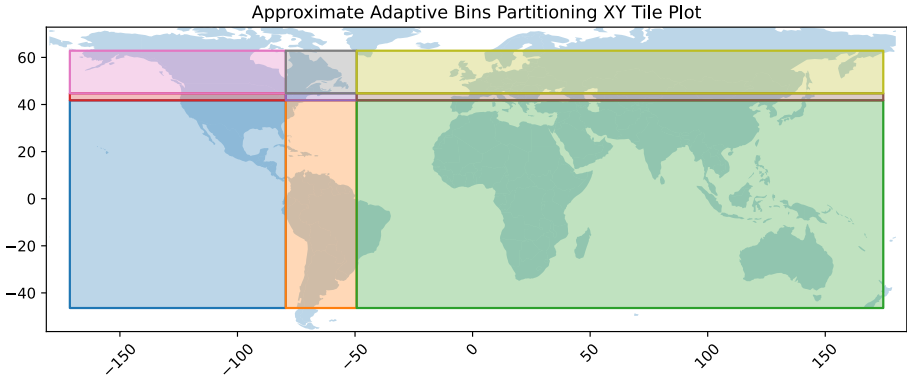

Figure 3.11: Approximate Adaptive Bins Partitioner XY projection using MobilitySpark.

Algorithm 3.9 Make Bounds Contiguous Algorithm

Require: A list of bounds *bounds_list*, real bounds *real_bounds*, dimension *dim*

Ensure: A list of contiguous bounds *final_bounds*

```
Sort bounds_list by minimum bound
Initialize current_max with the first minimum bound
Initialize an empty list contiguous_bounds
for all (index, (min_bound, max_bound)) in sorted list
    Adjust min_bound ← current_max
    Update current_max ← max_bound
    Add (index, (min_bound, max_bound)) to contiguous_bounds
end for
Initialize an empty list final_bounds
for each (key, (min_bound, max_bound)) in contiguous_bounds
    Adjust bounds using real bounds if first or last element
    Create bounding box box with new bounds min_bound and max_bound
    Add box to final_bounds
end for
Return final_bounds
```

3.3.4 Bisecting K-Means Partitioner

The last partitioner proposed in MobilitySpark uses Bisecting K-Means as a partitioning scheme for spatiotemporal data.

Bisecting K-Means combines clustering with a hierarchical approach. The `BisectiveKMeansPartitioner` is implemented in MobilitySpark using Spark's native methods for machine learning: Spark ML. This submod-

ule within Spark and PySpark contains numerous classes and methods to apply distributed machine learning to DataFrames and RDDs. Algorithm 3.10 illustrates the general idea behind Bisecting K-Means. Ideally, the cluster with the highest sum of squared error (SSE) is split at each iteration until the desired number of clusters is achieved.

In MobilitySpark, this partitioner expects a DataFrame of instants with the values of each dimension in separate columns. Once this is prepared, instantiating a `BisectiveKMeansPartitioner` is straightforward:

```
bkp = BisectingKMeansPartitioner(
    bounds=bounds, df=instants, traj_colname='instant', num_tiles=16
)
```

Thus, the requirement is to first materialize the instants of the dataset, including the x, y, t, and/or z values in different columns. This is necessary because the Bisective K-Means algorithm in Spark uses these values to form a vector, which is then used for clustering. Algorithm 3.11 illustrates the MobilitySpark approach for this implementation. Once each data point is clustered to the expected number of tiles, the spatiotemporal extent of each cluster is calculated with the `bounds_calculate_map` and `bounds_calculate_reduce` functions described earlier. This results in a set of n `STBox` tiles.

Clearly, this partitioner computes the grid by aggregating the points with a common cluster label. This generates overlapping clusters as seen in Figure 3.12.

Next sections, 3.4 and 3.5 present a set of experiments built to assess the performance of MobilitySpark using the different partitioning schemes, as well as their impact on typical SQL queries.

Algorithm 3.10 Bisecting K-Means Algorithm

Require: A dataset D, number of clusters k
Ensure: A set of k clusters
1: Initialize the list of clusters $C \leftarrow \{D\}$
2: **while** $|C| < k$
3: Select the cluster $C_i \in C$ with the highest SSE (Sum of Squared Errors)
4: Bisect C_i into two clusters C_{i1} and C_{i2} using standard K-Means
5: Remove C_i from C
6: Add C_{i1} and C_{i2} to C
7: **end while**
8: Return C

Algorithm 3.11 BisectiveKMeans Mobility Spark Algorithm Implementation

Require: Instants DataFrame I, number of tiles *num_tiles*, trajectory column name *traj_colname*, time zone *utc*

Ensure: A list of spatiotemporal tiles *tiles*

1: Select dimension columns from I as *dims*
2: Create a VectorAssembler $v \leftarrow$ VectorAssembler(*dims*)
3: Scale the features with $s \leftarrow$ StandardScaler()
4: Create $b \leftarrow$ BisectingKMeans(*num_tiles*)
5: Create Pipeline $p \leftarrow$ Pipeline($[v, s, b]$)
6: Fit the model *model* $\leftarrow p.fit(I)$
7: Compute tiles T by calculating the spatiotemporal bounds of each group of points belonging to each cluster.
8: Return T

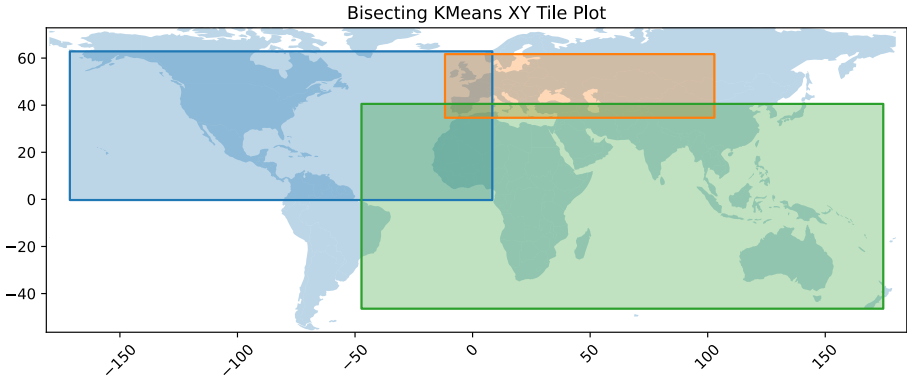

Figure 3.12: Bisective K-Means Partitioner XY projection using MobilitySpark.

3.4 MobilitySpark Partitioning on OpenSky Data

A classic mobility data use case involves tracking the status of different flights over time. OpenSky[4] provides a rich source of flight datasets that can be used to study their trajectories. The *states* dataset, in particular, offers comprehensive air-traffic data on a daily basis. In this project, we refer to this dataset as *OpenSky States*. This mobility dataset includes spatiotemporal columns such as *time*, *latitude*, *longitude*, and *altitude*. Additionally, it contains valuable information such as *velocity* and *heading*. The identifier column for each airplane is called *icao24*. According to OpenSky, this identifier is rarely changed and can be used to reliably track the evolution of flight trajectories. Each row represents an *instant* associated with a particular *icao24*.

The OpenSky dataset is an excellent use case for evaluating partitioning schemes for several reasons. Firstly, it contains high-dimensional spatiotemporal data that is representative of real-world mobility scenarios. This makes it an ideal candidate for testing the performance of different partitioning algorithms. Secondly, the data is contin-

[4] `https://opensky-network.org/`

uously updated and covers a wide geographical area, providing a diverse and comprehensive dataset for analysis. Finally, the dataset's granularity and volume present a significant challenge for data processing frameworks, making it a robust benchmark for evaluating the efficiency and scalability of partitioning schemes.

To assess the performance of MobilitySpark, particularly when using different partitioning schemes, we utilized the OpenSky States dataset to measure various aspects of MobilitySpark's partitioner performance. Our focus was primarily on the distribution of data obtained by partitioning the trajectories using the different partitioners offered by MobilitySpark.

We conducted two experiments using different data sizes from the OpenSky States dataset. The dataset used in these experiments[5] tracks flight data from June 27, 2022. Both experiments track flight locations globally, under the standard *UTC* timezone. The first experiment includes 15 minutes of data from 00:00:00hrs to 00:15:00hrs, while the second experiment spans 6 hours of data from 0 to 6AM. In the first experiment, we focused on a short time frame to examine the fine-grained details of the partitioning schemes' performance. This experiment allowed us to observe how each partitioner handles a smaller, more manageable dataset, providing insights into their efficiency and accuracy in distributing data evenly across partitions. The second experiment, with a significantly larger dataset, tested the scalability and robustness of the partitioners. By analyzing a longer period of flight data, we could evaluate how well each partitioning scheme copes with increased data volume and complexity. This experiment provided valuable information on the partitioners' ability to maintain an appropriate distribution and performance when handling larger datasets.

These experiments highlight the practical implications of using different partitioning schemes in MobilitySpark, demonstrating their strengths and limitations in processing real-world spatiotemporal data. The results offer a comprehensive evaluation of MobilitySpark's capabilities and inform the selection of appropriate partitioning strategies for various mobility data applications.

Finally, it is important to note that both experiments omit to include the *altitude* attribute as part of the spatiotemporal vector data. This is to simplify the problem and to be able to assess more fairly, in a simpler 2D plot, the spatial evolution of the trajectories.

The experimental setup for this exercise was performed using Databricks with AWS, under a cluster of 1 driver node with 32GB of memory and 4 cores, and 4 worker nodes with the same characteristics. All nodes in the system contained Spark 3.5.0, and PyMEOS. The experiments were built using a Jupyter Notebook in Databricks, and the datasets were uploaded to AWS S3 to be accessible by the cluster.

3.4.1 Data Processing

Both experiments start by reading the csv file into a Spark DataFrame and generating the instants:

```
df = df \
    .withColumn("time", F.from_unixtime(F.col("time"), "yyyy-MM-dd' 'HH:mm:ss")) \
    .withColumn("lat", F.round("lat", 2)) \
    .withColumn("lon", F.round("lon", 2)) \
    .withColumn("Point", create_point_udf("lat", "lon", "time")) \
    .withColumn("x", get_point_x("Point")) \
```

[5] Downloadable from `https://opensky-network.org/datasets/states/2022-06-27/`

```
    .withColumn("y", get_point_y("Point")) \
    .withColumn("t", get_point_timestamp("Point")) \
    .withColumn("id", F.monotonically_increasing_id()
)
```

The previous code snippet allows for the creation of a *Point* column of type `TGeomPointInstUDT`, in preparation to transform the dataset from `TInstant` to `TSequence`, or `TSequenceSet`. Next, the trajectories dataframe that is used throught the experiments is created with:

```
trajectories = df \
    .groupBy("icao24") \
    .agg(F.collect_list(F.col("Point")).alias("PointSeq")) \
    .select("icao24","PointSeq") \
    .withColumn("PointSeq", tgeompointseq_from_instant_list("PointSeq"))
```

Here, the UDF `tgeompointseq_from_instant_list` is called. This UDF sorts the instants associated to a particular *icao24* and then creates the `TSeq` object. Doing this results in a significant reduction of the row count of the DataFrame: experiment A reduces this from 530 thousand instants to 7095 trajectory sequences, while experiment B reduces this from 10.7 million instants to 20.1 thousand trajectory sequences. The final preprocessed trajectory dataset therefore looks like:

```
+------+--------------------+-----+
|icao24|            PointSeq|seqId|
+------+--------------------+-----+
|008ffa|[POINT(28.25 -26....|    0|
|009131|[POINT(28.25 -26....|    1|
|0100e7|[POINT(34.26 29.5...|    2|
+------+--------------------+-----+
```

Here, the `PointSeq` column is of type `TGeomPointSeqUDT`, and contains the instants associated to column `icao24`. The `seqId` column is an extra identifier to distinguish the trajectory at hand.

Keeping the a reference to the original instants is still useful, hence, an `instants` view is materialized, containing the *instant*, and the values of each *dimension* of the data, as seen in Figure 3.13. As explained in earlier sections, this table works as input for some of the MobilitySpark partitioners. Also, the bounds of the full dataset are calculated as seen before.

Once these steps are perfomed, the dataset is ready to be tested on the different partitioning schemes in MobilitySpark.

```
WITH instants_raw AS (
    SELECT
        movingobjectid,
        explode(instants(movingobject)) AS instant
    FROM trips
)
SELECT * FROM instants_raw
```

```
+--------------+--------------------+
|movingobjectid|             instant|
+--------------+--------------------+
|            18|POINT(497558.0253...|
|            18|POINT(497532.9560...|
|            18|POINT(497530.9870...|
|            18|POINT(497513.3023...|
|            18|POINT(497509.0459...|
+--------------+--------------------+
```

Figure 3.13: Materializing the instants from a MobilitySpark DataFrame using Spark SQL.

3.4.2 Regular Grid Partitioner

The experiments on the regular grid partitioner show how this strategy is practical given its fast instantiation time and whenever the load balancing over the tiles is not a concern. Figure 3.14 and Table 3.2 show the results of Experiment A and Experiment B using this partitioner. A sample of the partitioned trajectories is provided in Figure 3.15. Note how the tiles have the same size, and do not take into consideration the spatiotemporal context of the trajectories.

Experiment	Creation Time (s)	Mean Points per Partition	Standard Deviation
15 minutes	0.48	423.8666	510.9390
6 hours	0.53	830.5781	906.8453

Table 3.2: Results of Regular Grid Partitioning Experiments with OpenSky States Dataset.

It is clear that this strategy doesn't achieve a good load balancing, as it only splits the trajectories with no context but the general bounds of the dataset. The row distribution between tiled partitions is clearly not uniform, with groups of tiles containing a lot of rows, and others containing very few. Notably, among the $4^3 = 64$ tiles in this partitioner, the groups of tiles with a low distribution of rows represent `STBox` tiles that do not contain almost any spatiotemporal data. These can be in any spatial or temporal dimension. For instance, these tiles can fall in geographical areas that typically have low traffic, at times when traffic is even lower. For example: the countryside of a city at midnight. In the *OpenSky* datasets, these empty tiles can be tiles that represent portions of the ocean where very few flights pass through, at a time when flights are scarce.

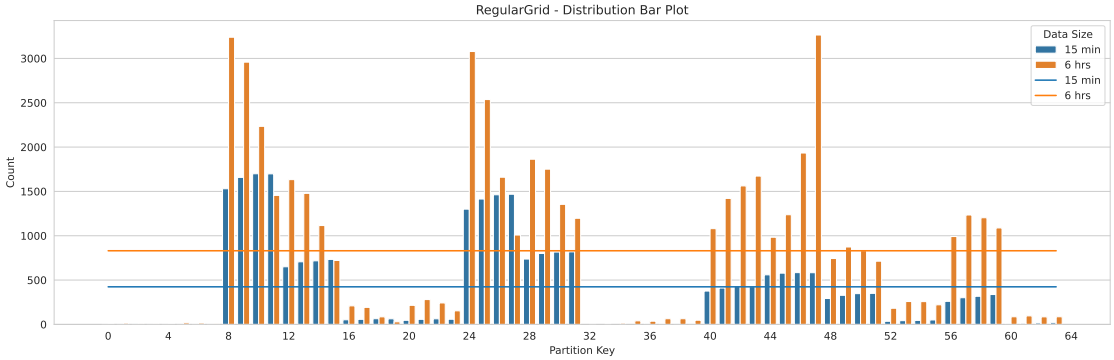

Figure 3.14: OpenSky States load balance using Regular Grid.

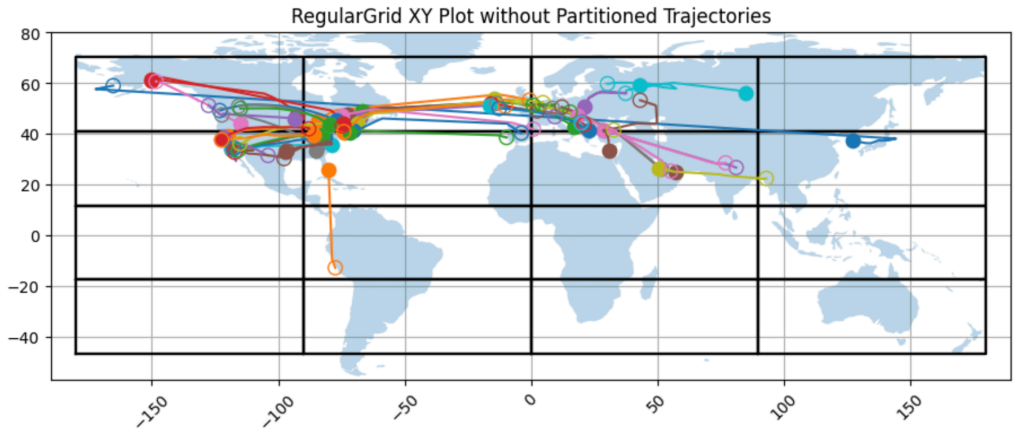

Figure 3.15: OpenSky States 6 hours data sample with Regular Grid.

3.4.3 In-Memory KD-Tree Partitioner

The first tested partitioner which considers the spatiotemporal context beyond the bounds was the In-Memory KD-Tree. As explained before, this partitioner loads the whole dataset, or a sample of it, into the driver node to calculate the tiles.

To estimate the grid of this partitioner, a sample of 5% of the total instants was taken as input in both experiments. To ensure fairness over the results, both samples were taken using the same seed number. No particular sampling strategy was used.

As observed in Table 3.3 and Figure 3.16, the achieved distribution is almost optimal for the *OpenSky States* 15 min. experiment (40.37 σ points per partition), while the other 6 hours experiment presents as well a promising load balance (174.78 σ points per partition).

Experiment	Creation Time (s)	Mean Points per Partition	Standard Deviation
15 minutes	4.56	407.3906	40.3766
6 hours	82.57	1021.8750	174.7834

Table 3.3: Results of In-Memory KD-Tree Partitioning Experiments with OpenSky States Dataset.

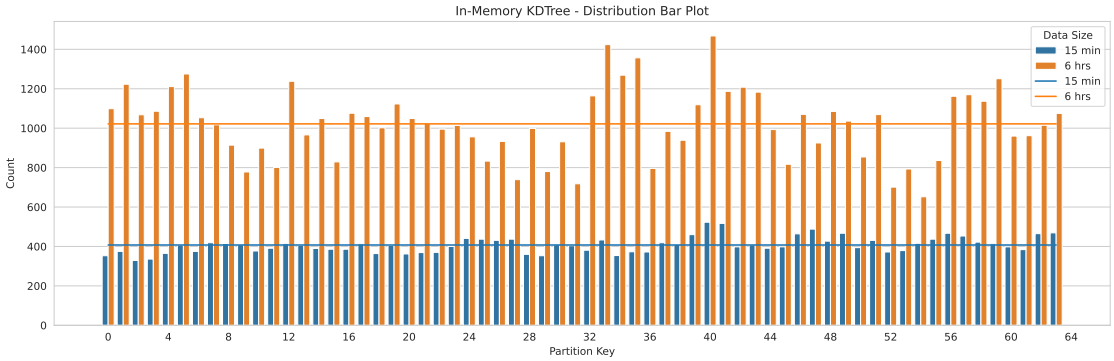

Figure 3.16: OpenSky States load balance using In-Memory KD-Tree Grid.

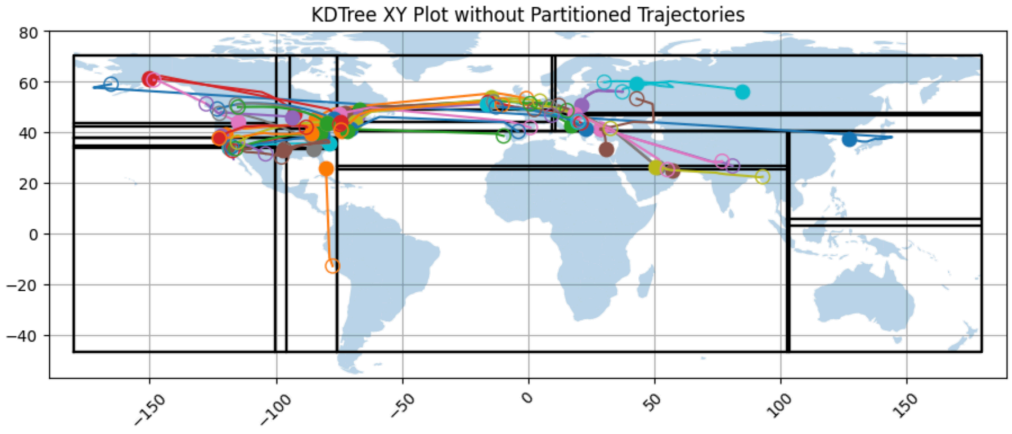

Figure 3.17: OpenSky States 6 hours data sample with In-Memory KD-Tree Grid.

Both experiments took 4.56s and 82.57s to generate the grid, respectively. In both experiments the maximum depth of the tree was established as 6, meaning that both grids resulted in $2^6 = 64$ tiles. Note how in Figure 3.17 the splits appear to happen where there is a higher concentration of points, explaining why it achieves a better load balance: spatiotemporal areas with a high count of points will be *rewarded* with a more compact tile, whereas those with a lower count of points will be *punished* with a broader tile.

The biggest disadvantage of using this strategy relies in the necessity of loading the data into the driver node, which naturally is not feasible nor recommended all the time. As mentioned, in this experiment it was needed to

pass only 5% of the instants. With this it was enough to generate a reliable enough grid, but what would happen if the dataset is just too big to be loaded in-memory, even the sample? Achieving a representative sample with less than 5% becomes a difficult task where the user should incorporate sampling techniques to leverage with this partitioner, if desired. However, it is true that once loaded in-memory, the computation time for the tree is acceptable.

3.4.4 Spark-based KD-Tree Partitioner

Experiment	Creation Time (s)	Mean Points per Partition	Standard Deviation
15 minutes	1841.67	406.1718	54.8242
6 hours	3142.92	1022.9687	164.5419

Table 3.4: Results of Spark KD-Tree Partitioning Experiments with OpenSky States Dataset.

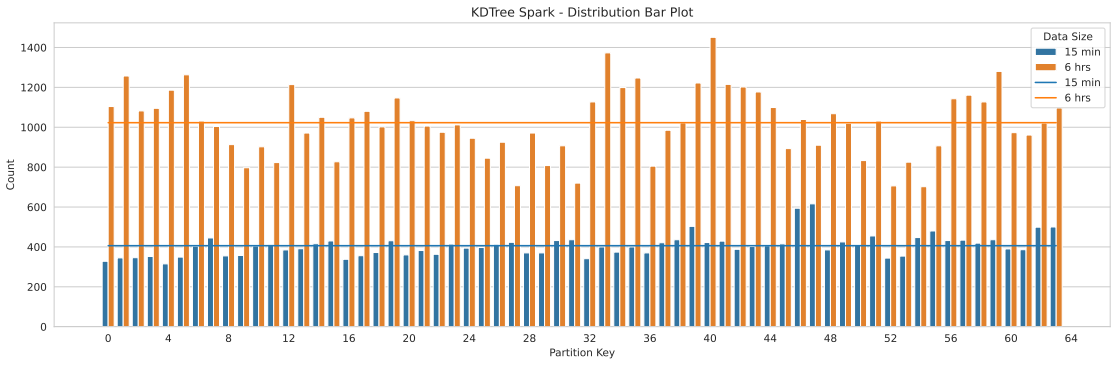

Figure 3.18: OpenSky States load balance using Spark KD-Tree Grid.

The alternative with the Spark-based KD-Tree Partitioner was tested as well with a sample of 5% of the total instants in the dataset. The algorithm has the same logic as its in-memory counterpart, however, the two main distinctions are that here, a Dataframe is received as input to be processed, and the median within the Dataframe is estimated using the *median of medians* approach.

Table 3.4 and Figures 3.18 and 3.19 show how a similar distribution is obtained over the tiles in the grid. However, for both datasets, the creation time turns out to be expensive: 1841.67 and 3142.92 seconds. This is explained because of the amount of DataFrame and RDD opeations in the algorithm, which trigger, each, separate Spark jobs, for each recursion in the tree. These experiments also obtained a grid of 64 tiles on 6 KD-Tree levels.

Therefore, the Spark-based KD-Tree solution is recommended when it is strictly necessary to consider a wider percentage of the dataset, when there are no time constraints in the creation of the grid, and when a good load balancing is required in the system.

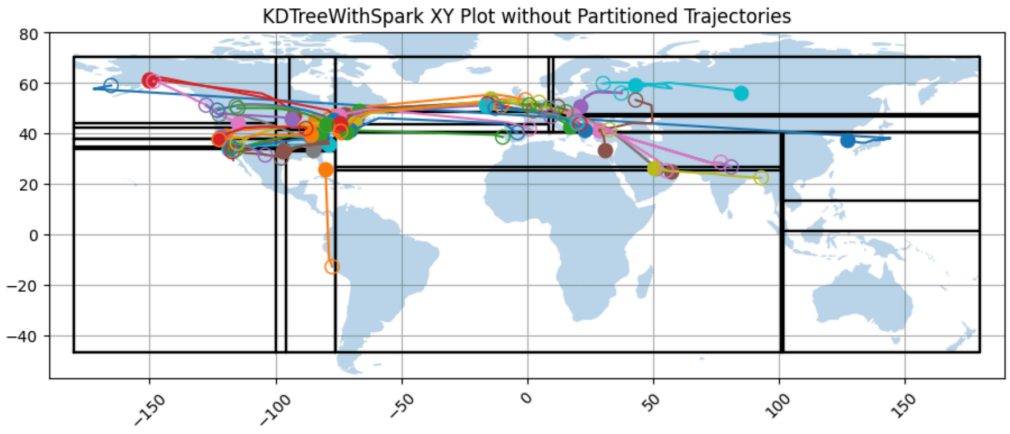

Figure 3.19: OpenSky States 6 hours data sample with Spark KD-Tree Grid.

3.4.5 In-Memory Adaptive Bins Partitioner

Experiment	Creation Time (s)	Mean Points per Partition	Standard Deviation
15 minutes	2.80	404.9687	181.2575
6 hours	46.64	943.1250	699.6410

Table 3.5: Results of In-Memory Adaptive Bins Partitioning Experiments with OpenSky States Dataset.

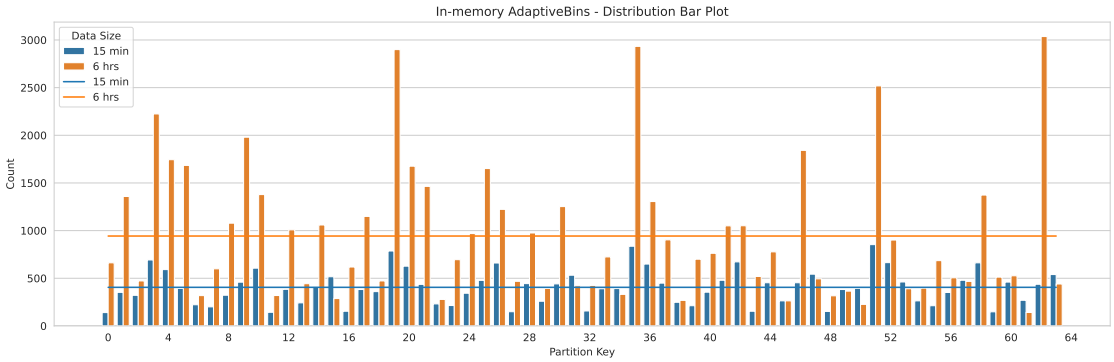

Figure 3.20: OpenSky States load balance using In-Memory Adaptive Bins Grid.

The in-memory implementation of the Adaptive Bins algorithm involves loading the instants or trajectories directly into the driver node. For the experiments conducted with this partitioner, a sample comprising 5% of the total instants was used. It is important to note that this partitioner, while it splits the space similarly to the Regular partitioner, considers the specific context and distribution of the trajectories.

Table 3.5 presents a satisfactory distribution achieved with grid creation times of 2.8 and 46.64 seconds, respectively. As illustrated in Figures 3.20 and 3.21, the tiles are more concentrated in areas with a higher density of instants. This indicates that the Adaptive Bins strategy effectively balances the simplicity of the Regular partitioner with the more nuanced approach of the KD-Tree versions, making it a reasonable mid-point solution.

The Adaptive Bins algorithm's ability to adapt to the spatiotemporal distribution of data points ensures that it efficiently manages densely populated regions while maintaining overall grid balance. This adaptive nature allows for more precise and relevant partitioning, which can significantly improve query performance and data processing efficiency. By focusing on the context of the trajectories, this partitioner minimizes data shuffling and optimizes the accessibility of related data points belonging to the same tile.

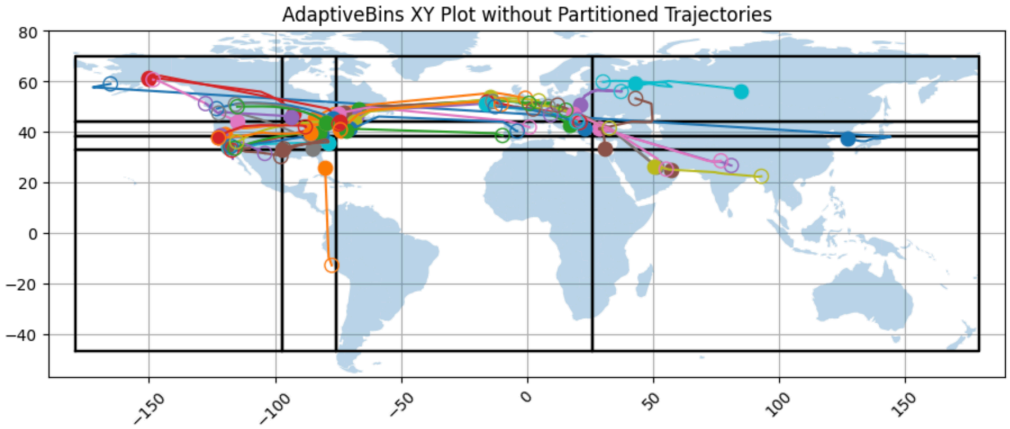

Figure 3.21: OpenSky States 6 hours data sample with In-Memory Adaptive Bins Grid.

3.4.6 Spark-based Adaptive Bins Partitioner

Using the Spark version for the Adaptive Bins partitioner allows to consider the whole dataset in reasonable times. Both experiments took as input the whole instant dataframe, and where able to obtain the grid in 18.75 and 138.03 seconds, as shown in Table 3.6.

The load balance from Figure 3.22 can be considered as the *real*, or *desired* load balance for any Adaptive Bins partitioner, as the whole dataset is considered in this scenario. It can be observed in Figure 3.23 that there is not much difference between the sampled approach taken in the in-memory version and the Spark version.

Experiment	Creation Time (s)	Mean Points per Partition	Standard Deviation
15 minutes	18.75	404.1093	184.3817
6 hours	138.03	940.75	694.8354

Table 3.6: Results of Spark Adaptive Bins Partitioning Experiments with OpenSky States Dataset.

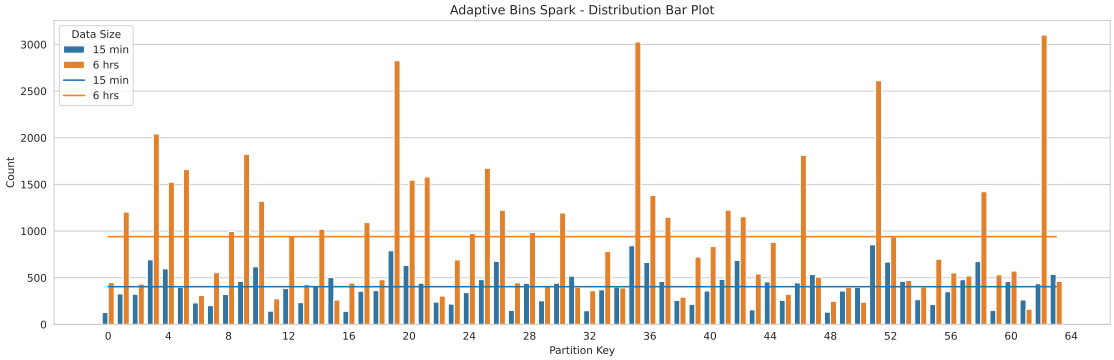

Figure 3.22: OpenSky States load balance using Spark Adaptive Bins Grid.

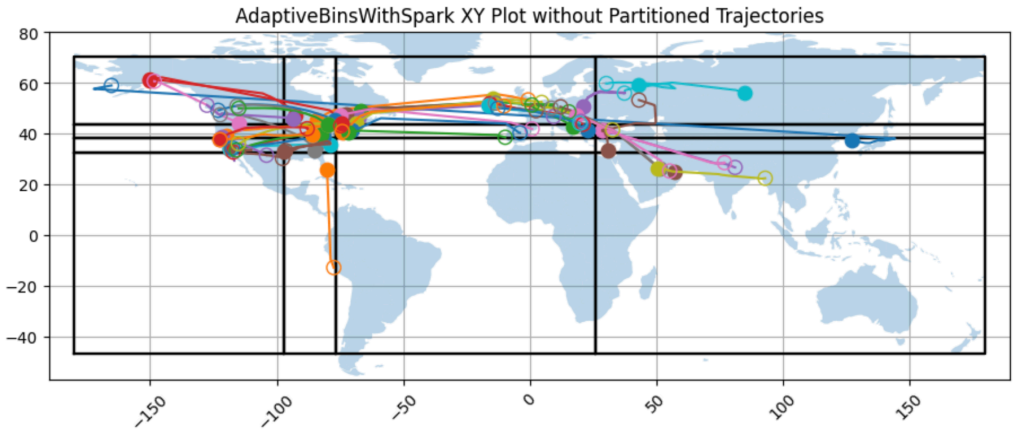

Figure 3.23: OpenSky States 6 hours data sample with Spark Adaptive Bins Grid.

This partitioner proves to be a good choice in cases where a higher instant sample needs to be considered in reasonable times. Still, some tiles will have a higher concentration of points, achieving a less balanced load.

3.4.7 MapReduce Approximate Adaptive Bins Partitioner

Experiment	Creation Time (s)	Mean Points per Partition	Standard Deviation
15 minutes	11.34	402.9531	272.6849
6 hours	73.04	944.5468	657.5073

Table 3.7: Results of Approximate Adaptive Bins Partitioning Experiments with OpenSky States Dataset.

An interesting option arises when we analyse the results obtained for Approximate Adaptive Bins implementation. To recall, this partitioner leverages a MapReduce approach, where the each default Spark partition of the

dataframe is considered a sample of the data. The Adaptive Bins is calculated for each sample and then the results are reduced into a single estimation of the tiles for the grid.

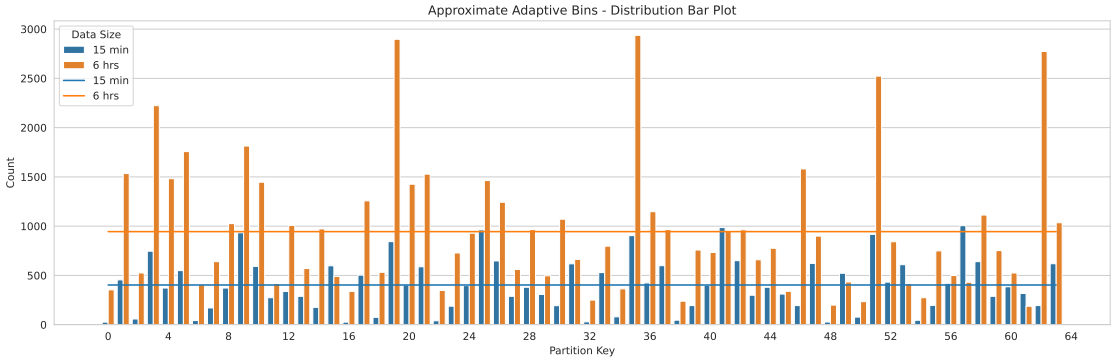

Figure 3.24: OpenSky States load balance using Approximate Adaptive Bins Grid.

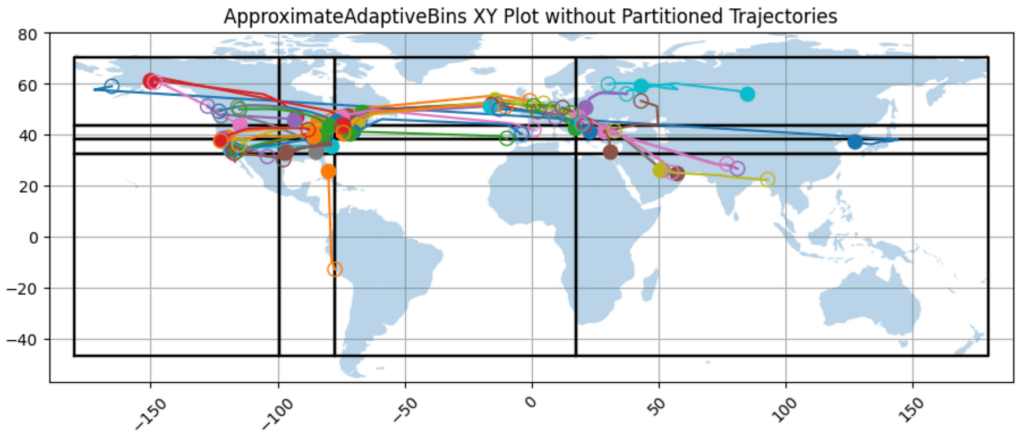

Figure 3.25: OpenSky States 6 hours data sample with Approximate Adaptive Bins Grid.

This partitioner makes a trade-off between obtaining a more exact answer, and time. It can be appreciated in Table 3.7 that the creation times of 11.34 and 73.04 seconds are very reasonable, and the obtained distributions in Figure 3.24 are very similar to the expected result. Figure 3.25 shows the same trajectory sample with the obtained grid. Another advantage of using this partitioner comes in that this approach also makes use of the whole dataset, it is not only analysing a sample of the data, and that it parallelizes effectively the operations, scaling horizontally across each default partition for the Dataframe in question.

Finally, a visual comparison between the resulting grids obtained with the In-Memory Adaptive Bins and the Approximate Adapative Bins can be studied in Figure 3.26[6]. Here it can be observed how both grids are almost

[6]The trajectory data sample in this plot, as it is appreciated, is different from the ones used in other figures in this thesis, but belong to the same experiment. Nevertheless, the grids contrasted in Figure 3.26 are the real ones obtained from the experiment.

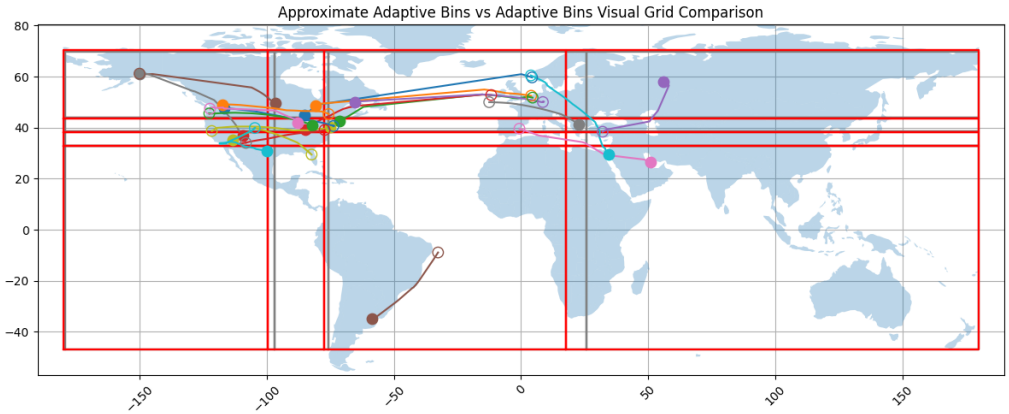

Figure 3.26: OpenSky States 6 hours Approximate Adaptive Bins Grid (red) vs In-Memory Adaptive Bins Grid (gray).

equivalent. Let's recall that the in-memory option uses only a sample of the data, and the recommendation is to build the grid with as much information as possible.

In a few words, the Approximate Adaptive Bins approach is definitely a good choice for users who want to obtain an estimated grid, reliably, in a reasonable time frame.

3.4.8 Spark-based Bisective K-Means Partitioner

Lastly, a different partitioning approach was studied with the Bisective K-Means partitioner. This approach has the characteristic that it is cluster-based, meaning that we don't assume an underlying grid which is afterwards split in some way. This can be appreciated in the map of Figure 3.28.

Besides, the generated tiles are not guaranteed to be separate from each other, meaning that overlaps can happen between them. Nevertheless, the results of this approach, shown in Table 3.8 and Figure 3.27, show how load balancing is not achieved in this scenario. Naturally, the clusters might contain more or less data points, and the algorithm does not consider the cluster size.

Experiment	Creation Time (s)	Mean Points per Partition	Standard Deviation
15 minutes	87.44	1708.2500	1921.1670
6 hours	130.10	4259.3125	3140.7799

Table 3.8: Results of Bisective K-Means Partitioning Experiments with OpenSky States Dataset.

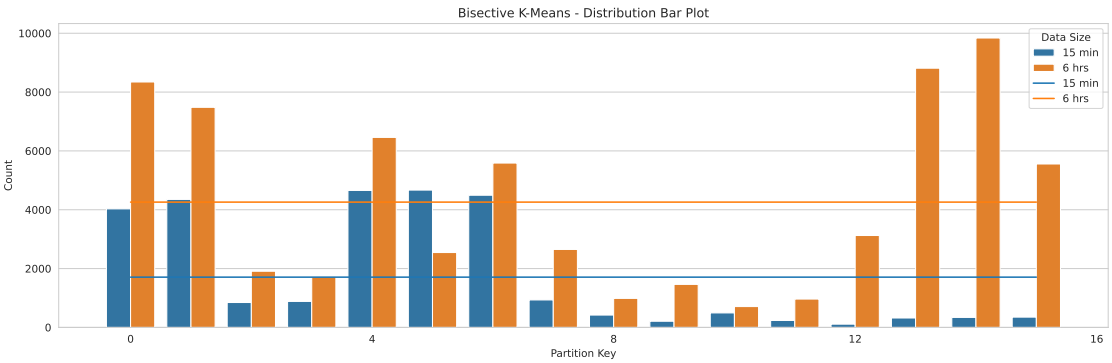

Figure 3.27: OpenSky States load balance using Bisective K-Means Grid.

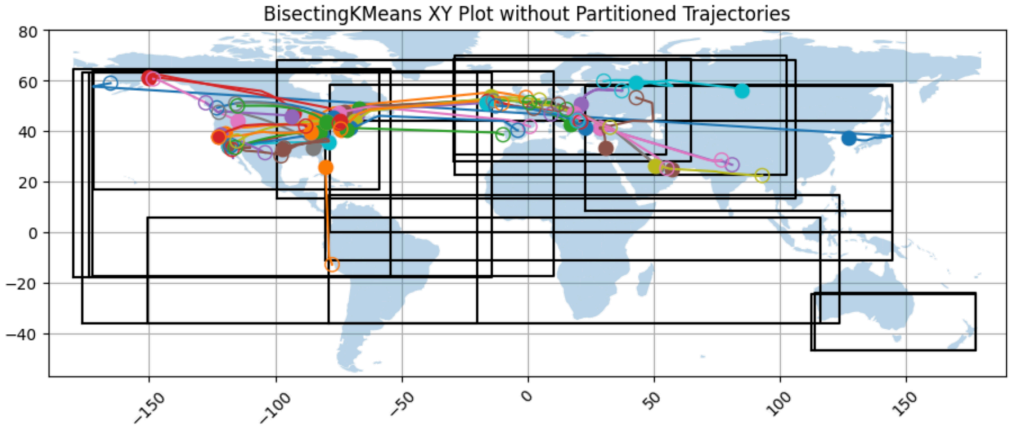

Figure 3.28: OpenSky States 6 hours data sample with Bisective K-Means Grid.

3.5 MobilitySpark Benchmarking with BerlinMOD

The properties of the different partitioners in MobilitySpark, in particular their ability to achieve an optimal load balancing, were studied in the *OpenSky States* experiments. Now, the focus turns to study the effectiveness of partitioning mobility data and its performance when dealing with common mobility queries, to extract insightful information from the spatiotemporal data at hand.

The BerlinMOD Benchmark [43] allows for the creation and performance measurement of moving object DBMS. Under MobilityDB and MEOS, an adaptation of this benchmark is available[7]. The goal is to emulate data from a particular spatiotemporal restricted domain and measure common enquiries that require some sort of mobility processing. For instance, BerlinMOD emulates trajectories of cars over a city within a time frame, i.e. a day. The data used for this thesis, for example, covers the city of Brussels.

[7] `https://github.com/MobilityDB/MobilityDB-BerlinMOD`.

The benchmarking experiments in this thesis cover three different scale factors: 0.1, 0.2 and 0.5. On each scale factor, four or three experiments were executed to test the query performance. Experiment 1 was designed to run the queries *AS-IS*, meaning no partitioning but the default one by Spark when loading the dataframes is enforced. Experiment 2 partitions the tables according to the trajectories, using the `RegularGridPartitioner`. Experiment 3 partitions the trajectories using the `AdaptiveBinsPartitionerSpark`, while experiment 4 partitions the data utilising the `ApproximateAdaptiveBinsPartitioner`. Hence, another goal is to evaluate the performance between the two Adaptive Bins implementations.

The experimental setup for this exercise was also performed using the same cluster as for the *OpenSky States* experiments: Databricks with AWS, 1 driver node, 4 worker nodes, each of them with 32GB memory and 4 cores. All nodes in the system contained Spark 3.5.0, and PyMEOS. The experiments were built using a Jupyter Notebook in Databricks, and the datasets were uploaded to AWS S3 to be accessible by the cluster.

3.5.1 Data Processing

Each BerlinMOD scale factor contains the datasets as described in Tables 3.9 and 3.10, which are then processed and instantiated into Spark as tables.

The *instants*, *licences*, *municipalities*, *periods*, *points*, and *regions* tables are common among each of the three scale factors, and contain the same number of rows and data. They contain information that is relevant to the context of the queries in the benchmark. For instance, the *regions* table defines different `Geometry` objects that are queried in the spatiotemporal context of the *trips* table.

Dataset	Size	Number of Rows
instants.csv	3.2 KB	100
licences.csv	1.6 KB	100
municipalities.csv	278.9 KB	19
periods.csv	1.4 KB	100
points.csv	8.8 KB	100
regions.csv	147.7 KB	100

Table 3.9: BerlinMOD: Datasets with no variation.

The remaining datasets, *tripsinstants* and *vehicles*, vary between scale factors. The first one defines all the `TPointInst` objects that represent a trip related to a particular vehicle. The *vehicles* table, on the other hand, defines the data of each particular vehicle in the database. The *tripsinstants* table, hence, can be interpreted as a *fact* table, while the other tables are *dimensions*.

Dataset	Size	Number of Rows	Scale Factor
tripsinstants.csv	2.6 GB	26,931,804	0.1
tripsinstants.csv	4.5 GB	47,180,511	0.2
tripsinstants.csv	10.6 GB	109,119,928	0.5
vehicles.csv	19.0 KB	632	0.1
vehicles.csv	26.9 KB	894	0.2
vehicles.csv	43.3 KB	1,414	0.5

Table 3.10: BerlinMOD: Trips and Vehicles datasets.

The first step to perform the experiments, loading the data, is performed following the pseudocode in Algorithm 3.12. It can be noted that the trips table should be created first, as this table is the one considered to partition the rest of the tables. This is a key step, as the created grid G will serve as an index to co-locate all the partitions of all the tables.

We can illustrate this with an example between the *trips* and the *periods* table. The trip rows here, each contain a `TGeomPointSeqSet` per tile in the partitioned data. Let's assume the grid has only two tiles: A and B, the trips table would end up having, per each original *trip*, two rows: $trip_A$ and $trip_B$. The periods table would also have, per period, two rows: $periods_A$ and $periods_B$. Here, we assume that the partitioned trips and the partitioned periods both intercept A and B. If they don't intercept, the relationship is not saved in the partitioned table. This relationship is saved as an index column called *tileid*. In Experiment 1, as no partitioned is enforced, and to maintain the queries equal between experiments, the *tileid* for all spatiotemporal values in all tables is set to -1.

Algorithm 3.12 BerlinMOD Load Table Pseudocode

Require: A table T.
Ensure: A partitioned and processed table.
1: Load T as raw data.
2: Make any necessary preprocessing transformation in T. $T \leftarrow \text{transform}(T)$.
3: **if** T is the *trips* table
4: Calculate the spatiotemporal bounds B of the *trips* table. $B \leftarrow \text{bounds}(T)$.
5: Create a grid G to partition the *trips* table. $G \leftarrow \text{create\_partitioner}(T, B)$.
6: Partition the *trips* table using the created grid. $T \leftarrow \text{partition\_table}(T, G)$.
7: Assign an index *tileid* to each row in T associated to each tile $t \in G$.
8: Save T, B and G.
9: **else**
10: Partition the T table using the created grid. $T \leftarrow \text{partition\_table}(T, G)$.
11: Assign an index *tileid* to each row in T associated to each tile $t \in G$.
12: Save T.
13: **end if**

To ensure the co-location of the data, in PySpark we utilize bucketing. In Spark SQL, bucketing a table is achieved by doing:

```
CREATE TABLE periods
USING parquet
CLUSTERED BY (tileid) INTO 8 BUCKETS
AS SELECT * FROM periodsRaw
```

Here, the periods table is created and bucketed into 8 sections, using the *tileid* as constraint. This is performed for all tables, ensuring that all data belonging to the same *tileid* will be located in the same bucket. This way, the *tileid* is used as an artificial index to query the tables. This concept is better apreciated in Figure 3.31, where the grid allows to restrict the search space of trips, periods, regions, etc.

Finally, the resulting tables are saved into the Databricks environment and cached, to ensure that when running the queries, we don't execute the previous steps, which can be longer for different tables. A sample of trajectories and their resulting grid from experiments 2, 3 and 4 is depicted in Figure 3.29. Note, again, how the different grids behave. The ones from experiments 3 and 4 concentrate the tiles where more trip instants are located, which happen to be the city center, downtown Brussels.

Now that we have processed the tables, the system is ready to benchmark spatiotemporal queries that involve the interaction between the trips table and the instants, periods, regions, etc.

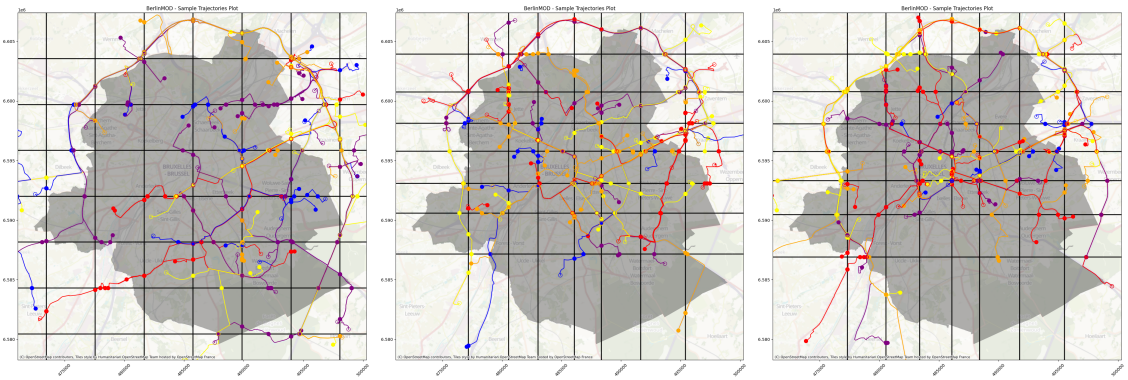

Figure 3.29: BerlinMOD 0.5 data sample and Grid, experiments 2, 3, and 4.

3.5.2 Query Benchmarking

As it has been previously mentioned, this thesis uses *BerlinMOD* within 4 sets of experiments: Experiment 1 executes the queries AS-IS, experiment 2 partitions the data using the `RegularGridPartitioner`, experiments 3 and 4, on the other hand, use the *Spark-based* and the *Approximate* implementations of the `AdaptiveBinsPartitioner` to partition the tables.

On each experiment, 7 queries in total are tested, which correspond to Queries 1, 2, 3, 4, 6, 11, and 18 of the original BerlinMOD benchmark [43]. As seen in Table 3.11, Queries 1 and 2 are *standard* queries, in the sense that they are only used to test the correct implementation of the database and perform no spatiotemporal calculation. Queries 3, 4, 6, and 11 test queries that involve some sort of spatial, temporal or spatiotemporal range between tables. Query 18, finally, is a nearest-neighbors query. The queries used, adapted to the context of MobilitySpark, can be found in the project repository.

Query Number	Type
1	Standard
2	Standard
3	Range
4	Range
6	Range
11	Range
18	Nearest-Neighbors

Table 3.11: BerlinMOD Query Types.

As mentioned, Queries 1 and 2 are framed as *standard*. Their goal is to assess that the database creation process was executed correctly and that now the user can proceed with the heavier queries in the benchmark.

Query 1, answers the question: *What are the models of the vehicles with licence plate numbers from QueryLicences?*. Here, a simple inner join on licence is executed. As this variable is not of mobility type, the execution should be seamless.

The same happens with Query 2. *How many vehicles exist that are 'passenger' cars?* This question is addressed immediately through a `COUNT` on *vehicles*, filtering by type *passenger*.

Query 3 answers to: *Where have the vehicles with licences from QueryLicences1 been at each of the instants from QueryInstants1?* First, a `CTE` is performed to filter the trips that match the licences of interest. Then, a join with the *instants* is triggered using the previously mentioned constraint: $trips.tileid = instants.tileid$. The MobilitySpark function `tpoint_at` is used to determine if two of the joined trips and instants are actually intersecting. If the value is null, the row is filtered out.

Query 4 is used to: *Which licence plate numbers belong to vehicles that have passed the points from QueryPoints?* Note the similarity with Query 3. However, the range value here is changed for a spatial value, namely, the point. Here, the MobilitySpark function is changed for `ever_intersects`. This is done in order to also assess the impact and performance of different MobilitySpark functions.

Query 6, *What are the pairs of licence plate numbers of "trucks", that have ever been as close as 10m or less to each other?* Here the useful `nearest_approach_distance` is used to determine the values to consider.

Query 11 resolves: *Which vehicles passed a point from QueryPoints1 at one of the instants from QueryInstants1?*. This query, essentially, mixes Query 3, of temporal nature, and 4, of spatial nature, into a spatiotemporal query. Again, the `tpoint_at` function is used in both instances, to join the trips, instants, and points.

Finally, Query 18 answers a nearest-neighbors question: *For each vehicle with a licence plate number from QueryLicences1 and each instant from QueryInstants1: Which are the 10 vehicles that have been closest to that vehicle at the given instant?* Here, a self join in the trips table is needed. First, the value of the trips at each instant has to be extracted, if it exists. After, the `nearest_approach_distance` is again used between the trip points at instant i. Finally, the results are ranked with the help of window functions and the top results are returned.

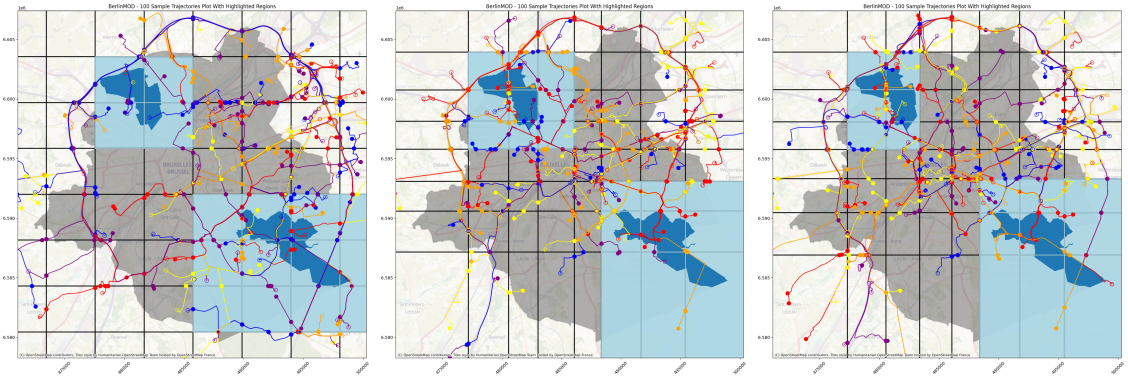

Figure 3.30: BerlinMOD 0.5 with Jette and Auderghem tiles highlighted, experiments 2, 3, and 4.

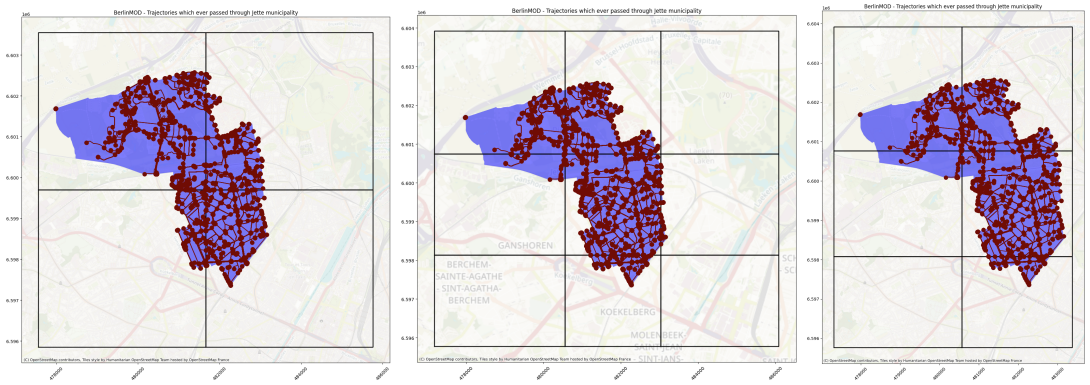

Figure 3.31: BerlinMOD 0.5 Trips which ever passed through Jette, experiments 2, 3, and 4.

As two extra products of the benchmarking, two sets of maps were created: Figures 3.30 and 3.31. The first figure includes a subset of the trips, with the grid, and the highlighted tiles for two regions in Brussels: Jette and Auderguem.

The second figure, shows the number of trips which ever passed through Jette. Note how, depending on the partition strategy, more or less tiles are selected. This has a direct impact on the number of computations that are triggered when doing, first the join by *tileid*, and second when calling the UDF functions to perform the spatiotemporal operations.

Figure 3.32 shows the comparison of the query execution times. To avoid missing the difference between query times, the values were transformed using the `log` function.

As it can be appreciated, Experiment 1 fails at executing queries 4, 6, 11, and 18. Of all the spatiotemporal queries, the only one which succeeds in executing, is Query 3. The rest of the queries, as seen, are executed for Experiments 2, 3, and 4.

The rest of the queries show a similar behavior between each other over the experiments, with Query 11 being the one which takes the most time overall. Small gains are achieved when using the `AdaptiveBins` from Experiments 3 and 4, particularly in Query 3, 4, and 6.

Next section, Section 3.5.3 will analyse overall the query performance from the perspective of PySpark.

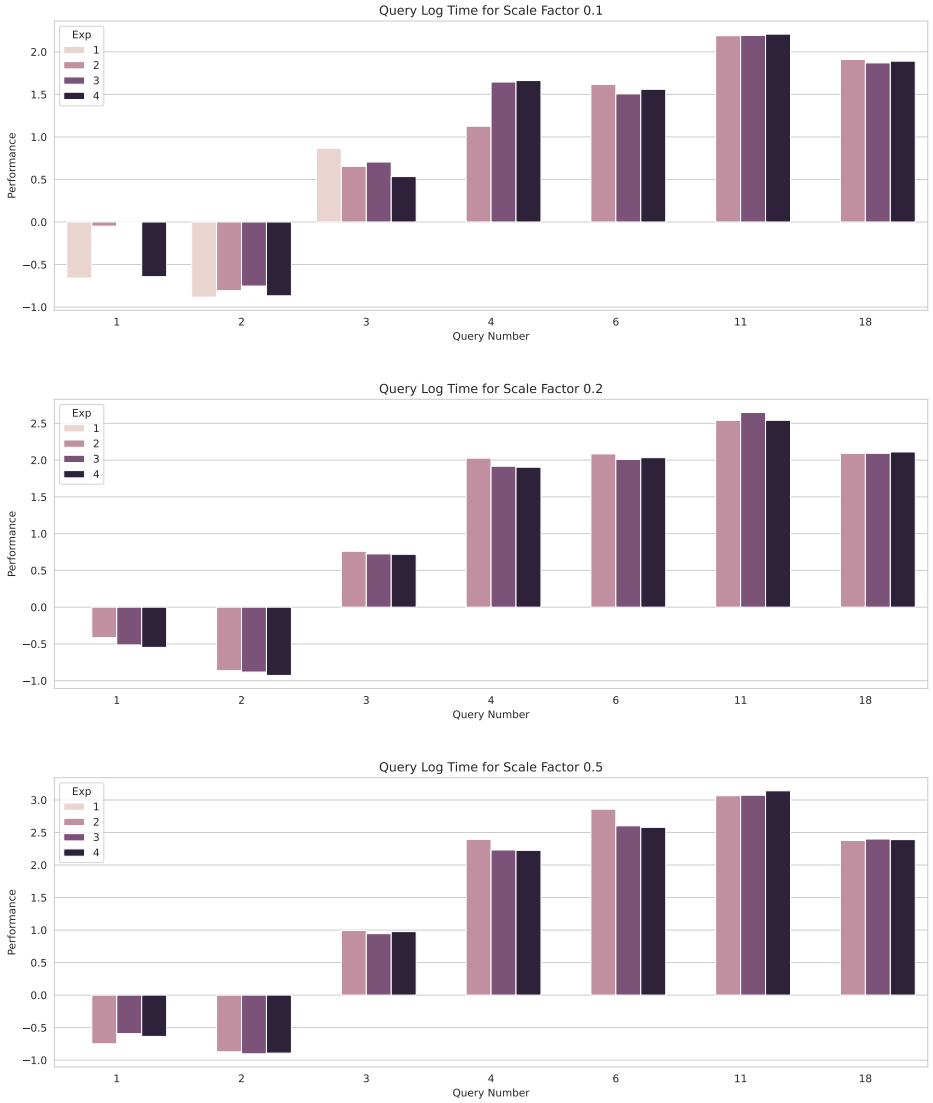

Figure 3.32: BerlinMOD Query Log Execution Times per Scale Factor.

3.5.3 On the Query Performance using MobilitySpark

So far, the effectiveness of the queries in MobilitySpark has been studied using a numerical comparison approach between experiments, scale factors, and queries. However, it is crucial to take a step back and analyze the overall query performance of a system like MobilitySpark, which utilizes PySpark and PyMEOS as its engines. This comprehensive understanding will help in identifying the strengths and potential bottlenecks of the system, leading to more informed decisions for future improvements.

During the preprocessing steps, the tables were bucketed according to the generated spatiotemporal grid. The goal of this step is to enhance the concept of co-location between values. Co-location is particularly beneficial for spatiotemporal data, as it ensures that values sharing similar spatial and temporal characteristics are stored together. This proximity minimizes the data retrieval time and optimizes the performance of spatial queries.

By implementing co-location, PySpark can efficiently access the same bucket when querying values belonging to the same tile. This reduces the number of shuffles performed during query execution, leading to significant performance gains. Shuffles in distributed data processing systems like PySpark involve redistributing data across different nodes, which can be an expensive operation in terms of both time and resources. Minimizing shuffles by leveraging co-location ensures that data movement is kept to a minimum, thereby accelerating query processing and improving the overall efficiency of the system.

```
SELECT
    t.tileid, t.movingobjectid, t.movingobject,
    p.period, t.vehid
FROM trips t INNER JOIN periods p ON (
    at_period(t.movingobject, p.period) IS NOT NULL
)
```

Figure 3.33: Join on spatiotemporal condition only.

```
SELECT
    t.tileid, t.movingobjectid, t.movingobject,
    p.period, t.vehid
FROM trips t INNER JOIN periods p ON (
    t.tileid = p.tileid AND
    at_period(t.movingobject, p.period) IS NOT NULL
)
```

Figure 3.34: Join on *tileid* and spatiotemporal condition.

As discussed in earlier sections, one of the significant drawbacks of selecting PySpark's API as the foundation for MobilitySpark is the limited configurability it entails. For example, while using Spark, one could create a custom RDD and custom Catalyst rules for spatiotemporal functions in MEOS. However, this level of customization is currently unattainable in MobilitySpark due to the reliance on PySpark, impacting the system in various ways, particularly in terms of query performance.

One notable issue is PySpark's treatment of UDF functions as *black boxes*. This means that the UDF function, being a Python executable, is passed unchanged to the Java Virtual Machine, with no optimization performed by Catalyst. This represents a significant disadvantage, needing a thoughtful design of Python UDF functions. Additionally, these functions are executed in a row-wise manner, and despite being parallelized across each partition of the DataFrame, execution times can escalate rapidly.

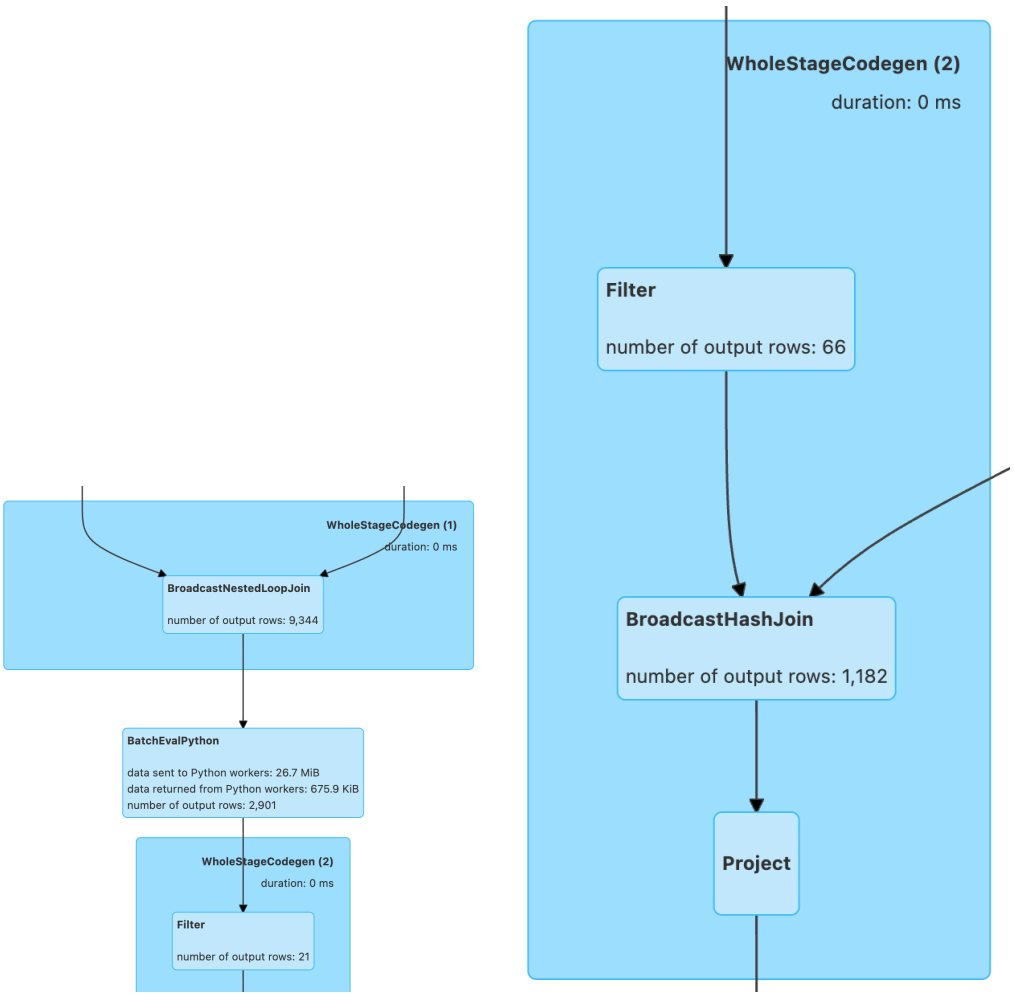

Figure 3.35: MobilitySpark query plans using BroadcastNestedLoopJoin and BroadcastHashJoin.

An alternative that mitigates the lack of Catalyst optimizations is the use of Pandas UDFs instead of regular UDFs. Pandas UDFs handle each partition as a Pandas Series or DataFrame, allowing internal parallelization of operations. However, this tool is still under development and currently requires the use of Spark's Arrow Serializer. This means that for MobilitySpark UDTs, custom Arrow data types would need to be created, which is beyond the scope of this thesis. While creating a Pandas UDF in MobilitySpark is feasible and was tested in this project, it proved inefficient. The data needs to be converted to a Pandas `DataFrame` or `Series` of `String` and then cast back to the original PyMEOS data type within the function. Consequently, it was decided to use regular UDFs as wrappers for PyMEOS functions in MobilitySpark.

The primary disadvantage, however, is that UDF function calls can trigger a cross-join if they are the sole join condition in a Spark SQL query, for instance, as in Figure 3.33.

This scenario results in a warning from Spark: `WARN ExtractPythonUDFFromJoinCondition`. This warning is issued when Spark detects that the join condition contains only a Python UDF. By default, Spark pushes

this condition out of the join, thereby enforcing a cross-join on the two relations, which can be a significant performance bottleneck.

This issue can be mitigated by adding an additional condition for the join. In MobilitySpark, utilizing the implemented tile partitioning as the join condition is advantageous, as it pushes the spatiotemporal query out of the join condition and converts it into a filter.

The warning disappears when the condition on *tileid* is added. Moreover, the change in the query plan can be observed in Figure 3.35, where simply adding the *tileid* condition elevates the join instruction from a `BroadcastNestedLoopJoin` to a `BroadcastHashJoin`. The co-location of the relations by *tileid* is essential at this stage, as the query plan benefits significantly from the reduction in shuffles. This pair of queries were tested locally on a subset of the *BerlinMOD* Scale Factor 0.1, utilizing the `AdaptiveBinsPartitionerSpark`, resulting in query times of 7 seconds for the cross-join query versus 2 seconds for the hash-join, co-located query.

This example highlights the importance of dedicated query design and effective data management in scenarios such as these. In MobilitySpark, this optimization is implemented in the *BerlinMOD* experiments and is recommended for users to adopt. As mentioned, there are numerous enhancements that can further optimize query performance. Some of these improvements can be achieved using the PySpark API as demonstrated, such as implementing the PyMEOS data types as Arrow data types and extending the current UDFs as Pandas UDFs.

3.6 Overview of MobilitySpark Results

In Chapter 3, we explored the architecture and functionality of the MobilitySpark system, along with two experimental sets demonstrating the importance of a sophisticated partitioning strategy for efficiently managing spatiotemporal mobility data across various use cases. The *OpenSky* experiments assessed the partitioners in terms of load balancing and instantiation time, while the *BerlinMOD* experiments concentrated on evaluating query performance within the MobilitySpark framework. Though we briefly addressed some advantages and limitations, a more comprehensive discussion of our conclusions and future directions will be presented in Chapter 4.

This concluding section of the chapter aims to summarize the MobilitySpark system and the results obtained from the two experimental sets: *OpenSky* and *BerlinMOD*.

As investigated, MobilitySpark integrates PyMEOS with PySpark, facilitating the execution of mobility-related use cases in a distributed environment using PyMEOS data types and functions. The system comprises four distinct modules that interact with each other: one for user-defined types, one for user-defined functions, one for user-defined table functions, and one for the partitioning techniques assessed in this study.

The partitioning strategies were rigorously tested through a series of experiments to determine their effectiveness in mobility data scenarios. MobilitySpark features four primary partitioning methods, each with multiple implementations: Regular Grid, KD-Tree, Adaptive Bins, and Bisective K-Means. The KD-Tree partitioner includes both in-memory and Spark-based versions. The Adaptive Bins approach, in addition to having in-memory and Spark implementations, introduces a MapReduce methodology through the ApproximateAdaptiveBins variant.

The *OpenSky* experiment specifically evaluated the load balancing and creation times of the partitioners. The results are detailed in Tables 3.12 and 3.13. To illustrate load balance, we utilized entropy as a metric; higher entropy values indicate better load distribution.

It is noteworthy that the KD-Tree implementations provided the best load balance, albeit at the expense of more computationally intensive operations, particularly with the Spark KD-Tree variant. A notable trade-off was observed with the Adaptive Bins implementations, especially with `AdaptiveBinsSpark` and `ApproximateAdaptiveBins`, as both utilize the entire dataset rather than a sample, enhancing the accuracy of load distribution.

Experiment	Time 15 min (s)	Time 6 hours (s)
RegularGrid	**0.48**	**0.53**
In-Memory KDTree	4.56	82.57
KDTree Spark	1841.67	3142.92
In-Memory AdaptiveBins	2.80	46.64
Adaptive Bins Spark	18.75	138.03
Approximate Adaptive Bins	11.34	73.04
Bisective K-Means	87.44	130.10

Table 3.12: Time measurements for OpenSky experiments.

Experiment	Entropy 15 min	Entropy 6 hours
RegularGrid	3.412889	3.565827
In-Memory KDTree	**4.154129**	4.144415
KDTree Spark	4.150371	**4.146060**
In-Memory AdaptiveBins	4.059430	3.921177
Adaptive Bins Spark	4.055483	3.924045
Approximate Adaptive Bins	3.913422	3.948653
Bisective K-Means	2.207589	2.511231

Table 3.13: Entropy measurements for OpenSky experiments.

The results of the *BerlinMOD* experiment are summarized in Table 3.14. It is evident that a non-partitioned data strategy is inefficient, as demonstrated by the inability of most queries to execute in Experiment 1. However, the subsequent experiments show varying results concerning query execution times. While the advantage of using a partitioning strategy is clear, the performance benefits vary across different queries, with some queries showing better results with specific strategies such as Regular Grid, and others with Adaptive Bins.

For instance, Queries 3 and 4 are executed more efficiently when the data is partitioned using the Adaptive Bins strategy, whereas Query 18 performs slightly better with the Regular Grid approach. This discrepancy can be attributed to the inherent nature of the Regular Grid method, which partitions data based solely on the dataset's bounds. Consequently, the tile boundaries might align in such a way that optimally supports certain queries. As depicted in Figure 3.31, the Regular Grid partitioning notably benefits queries targeting the Jette area in Brussels, where the grid aligns well with the area's spatiotemporal characteristics.

In the forthcoming final chapter, we will provide a comprehensive reflection on the MobilitySpark system as a whole. This will include discussions on potential future work and enhancements that could be made to the framework. The chapter will conclude with a brief recapitulation of the key topics and findings presented throughout the thesis.

Query number	1	2	3	4	6	11	18
Scale Factor	**Experiment 1: AS-IS**						
0.1	0.2192	0.13097	7.34467	-	-	-	-
0.2	-	-	-	-	-	-	-
0.5	-	-	-	-	-	-	-
Scale Factor	**Experiment 2: Regular Partitioner**						
0.1	0.893	0.156	4.510	13.368	41.444	154.613	81.317
0.2	0.385	0.137	5.761	106.445	121.515	347.434	123.299
0.5	0.178	0.134	9.863	248.041	722.068	1163.426	238.798
Scale Factor	**Experiment 3: Spark-based Adaptive Bins Partitioner**						
0.1	0.988	0.176	5.072	44.198	32.074	156.269	74.136
0.2	0.306	0.131	5.302	82.532	102.427	444.813	123.816
0.5	0.256	0.125	8.811	170.503	400.776	1184.570	251.095
Scale Factor	**Experiment 4: Approximate Adaptive Bins Partitioner**						
0.1	0.227	0.135	3.421	46.023	36.294	161.597	77.768
0.2	0.283	0.117	5.237	80.110	108.061	348.092	129.123
0.5	0.231	0.128	9.473	168.33	379.753	1384.160	246.083

Table 3.14: BerlinMOD Query Execution Times (in seconds).

3.7 Final Remarks

This section introduces a novel Spark binding for MEOS via PySpark and PyMEOS, named *MobilitySpark*. A series of experiments were conducted to assess the efficiency of various partitioning strategies within a distributed environment like Spark.

Initially, *MobilitySpark* was presented as a binding to process mobility data using the PyMEOS library in PySpark. The library comprises four core modules: User-defined Types (UDT), User-defined Functions (UDF), User-defined Table Functions (UDTF), and Partitioning strategies. Additionally, an auxiliary Utilities module is included.

Subsequently, the Partitioning module was examined in detail, outlining the functionality of the proposed partitioning strategies: *Regular Grid*, *KD-Tree*, *Adaptive Bins*, and *Bisective K-Means*. Both in-memory and Spark-based implementations were proposed for the *KD-Tree*. An approximation method for *Adaptive Bins* was also proposed, utilizing MapReduce to estimate partition values, treating each default partition as a sample.

Following this, these partitioning schemes were tested through two series of experiments: *OpenSky* and *BerlinMOD*. The *OpenSky* experiment focused on evaluating the quality of load balancing and the creation time for each partitioner. In contrast, the *BerlinMOD* experiment used synthetic data from Brussels to evaluate mobility queries in a distributed setting. Lastly, a reflection on the benefits of improved query times and potential enhancements for future versions of MobilitySpark for mobility queries was provided.

This chapter ends with an executive summary of the experimental results obtained thorugh the *OpenSky* and *BerlinMOD* experiments. In the next chapter, Chapter 4, we will systematically review the insights gained from this research. This will include final thoughts and reflections, results discussed in the current chapter.

4

Conclusions and Future Work

One learns from books and example only that certain things can be done. Actual learning requires that you do those things.
— Frank Herbert

This thesis has introduced and proposed a mobility data management system designed for a distributed environment. The implementation leverages the capabilities of the Mobility Engine Open Source (MEOS) library and Apache Spark. By integrating these two systems, the project extends MEOS functionality into a widely-used framework like Spark, thereby providing the necessary infrastructure for a distributed setting facilitated by Spark's architecture.

In this concluding chapter, we will revisit and thoroughly review the key findings and insights gained through the development of *MobilitySpark*. Additionally, we will reflect on the empirical comparisons of various partitioning strategies evaluated during the *OpenSky* and *BerlinMOD* experiments. These comparisons have highlighted the strengths and limitations of different approaches in handling spatiotemporal mobility data, offering valuable insights into the efficiency and scalability of partitioning strategies in distributed systems.

Finally, we will discuss potential avenues for future work. This section will address the limitations of the current MobilitySpark implementation, identifying areas where improvements can be made. We will also explore potential research directions that have emerged from the findings presented in this thesis, offering a roadmap for further innovation and development in the field of distributed mobility data management. This exploration will include considerations for enhancing the scalability, efficiency, and usability of the system, ensuring that MobilitySpark can adapt to evolving technological and data landscape requirements.

4.1 Impact of MobilitySpark

Before delving into the specific research questions, it is crucial to evaluate the impact of implementing a binding like MobilitySpark. The influence of MobilitySpark can be assessed in two main areas: its impact on the MEOS environment and its contributions to Mobility Data Management research. It is important to make this distinction, as it showcases the benefits obtained from an implementation and a research perspective.

The MEOS family[1] significantly benefits from the inclusion of a binding dedicated to a framework that targets mobility data. MobilitySpark leverages PyMEOS to create an initial version of a distributed MobilitySpark binding. This development highlights the richness of the MEOS library family and highlights the extensive opportunities for further extensions into multiple programming languages, frameworks, paradigms, and databases. Additionally, MobilitySpark has the potential to serve as an alternative to Distributed MobilityDB in the future or even as an integrated connector to it, providing a versatile tool for managing distributed mobility data.

In recent years, research on mobility data has become increasingly significant. Mobility data is crucial for advancing modern logistics, smart city infrastructure, intelligent devices, and more. From a data science perspective, this thesis makes a valuable contribution by providing practical insights into the application of partitioning strategies for mobility data. By incorporating MEOS, MobilitySpark offers a standardized set of functions and data types that can serve as a foundational resource for future research and projects within the community. This standardization facilitates more consistent and comparable research outcomes and encourages further innovation in the field of mobility data management.

4.2 Conclusion to Research Objectives

In this section, we address the specific outcomes achieved in response to the research objectives outlined in Chapter 1, which introduced the significance of mobility data and its importance. Each of the four research objectives was addressed either directly or indirectly throughout the project, and as will be demonstrated, these objectives have been successfully met.

4.2.1 Implementing a MEOS Binding Using Spark

The primary objective of this research was to integrate MEOS with Spark effectively. As detailed in Chapters 2 and 3, the challenge in creating this binding stemmed from the fact that MEOS is written in C, whereas Spark is based on Scala/Java. To address this challenge, two potential solutions were considered: leveraging JMEOS, a Java binding of MEOS, or utilizing PyMEOS, a more actively maintained Python binding of MEOS, in conjunction with PySpark. Ultimately, the latter approach was chosen, resulting in the successful development of the project's first iteration, known as *MobilitySpark*.

As discussed in Chapter 3, MobilitySpark encompasses various modules that enable users to effectively encapsulate PyMEOS functions and data types. Additionally, it utilizes UDTF generators and a partitioning module

[1] See Chapter 2.

to create partitioned tables with spatiotemporal context. Therefore, the objective of developing an initial working version of the MEOS-Spark binding has been achieved.

4.2.2 Identifying Trajectory Data Partitioning Schemes

The research presented in Chapter 2 emphasizes that distributed DBMS systems specialized in mobility data should incorporate a *spatiotemporal context* to optimize query execution, ETL processes, and overall data management. Achieving this spatiotemporal context is partially facilitated by effective partitioning strategies.

Although the literature on partitioning for spatial and spatiotemporal data presents a broad range of strategies, this project identified and selected four distinct approaches guided by principles of simplicity and efficiency, combined with adequate load balancing: Regular Grid, KD-Tree, Adaptive Bins, and Bisective K-Means. As noted in Chapter 2, while more complex strategies exist—some even involving machine learning models to refine partitioning—such strategies are generally discouraged unless specifically required by the use case.

In conclusion, the work presented here has successfully addressed the proposed research objectives, demonstrating the feasibility and benefits of the developed systems and strategies. This foundational work sets the stage for further exploration and refinement in the area of distributed mobility data management.

4.2.3 Adaptation of a Spark Environment to Distribute Trajectory Data

Building on the previous objective, the next logical step involved implementing the identified partitioning strategies within the context of MEOS and Spark. In MobilitySpark, a comprehensive Partitioning module was meticulously integrated, offering seven distinct implementations corresponding to the four strategies examined throughout the thesis.

MobilitySpark's partitioners, as outlined in Table 4.1, were deployed in various experiments to assess their impact. The *OpenSky* experiments provided a critical testing ground, where all seven implementations were successfully applied and tested. This success highlights the integration of these partitioning strategies into a Spark environment, leveraging MEOS as the foundational mobility data processing engine. The successful implementation of these partitioners demonstrates the feasibility and effectiveness of distributing trajectory data using Spark, allowing for scalable and efficient data management in a distributed system.

The values presented in Table 4.1 offer a detailed comparison of the partitioning strategies utilized in MobilitySpark. The table categorizes the partitioners based on their deployment in In-Memory, Spark, and MapReduce environments, as well as whether they use samples. For instance, the Regular Grid partitioner, while effective, does not utilize Spark or MapReduce technologies and does not require samples. On the other hand, the KD-Tree partitioner leverages both In-Memory and Spark environments and uses samples to enhance its functionality. The Adaptive Bins partitioner is versatile, employing In-Memory, Spark, and MapReduce methods, although it relies on samples exclusively in its In-Memory implementation. Lastly, the Bisective K-Means partitioner, primarily based in Spark, also uses samples to optimize data distribution. These distinctions highlight the flexibility and varied applications of each partitioning strategy, demonstrating how they cater to different aspects of mobility data management within a distributed system.

Partitioner	In-Memory	Spark	MapReduce	Uses Sample
Regular Grid	Yes	No	No	No
KD-Tree	Yes	Yes	No	Yes
Adaptive Bins	Yes	Yes	Yes	Only In-Memory
Bisective K-Means	No	Yes	No	Yes

Table 4.1: Comparison of partitioning strategies in MobilitySpark.

4.2.4 Demonstrate How a Partitioning Strategy Impacts Data Science Tasks

A crucial objective of this thesis was to demonstrate the significant impact of an effective partitioning strategy on data science tasks. This goal was addressed through benchmarking in the *BerlinMOD* experiments. In mobility data science, typical tasks include executing spatiotemporal queries that need a series of data transformations. The *BerlinMOD* experiments were designed to rigorously test the mobility data management system's capability to handle such queries and to evaluate its overall performance.

Through this research, we successfully illustrated that employing different partitioning strategies can enhance the efficiency of spatiotemporal query execution. Moreover, these strategies help achieve load balancing that incorporates spatiotemporal considerations, thereby optimizing system performance. The various partitioners used in the experiments demonstrated their unique strengths and how they can be strategically selected based on the specific requirements of the data science tasks at hand.

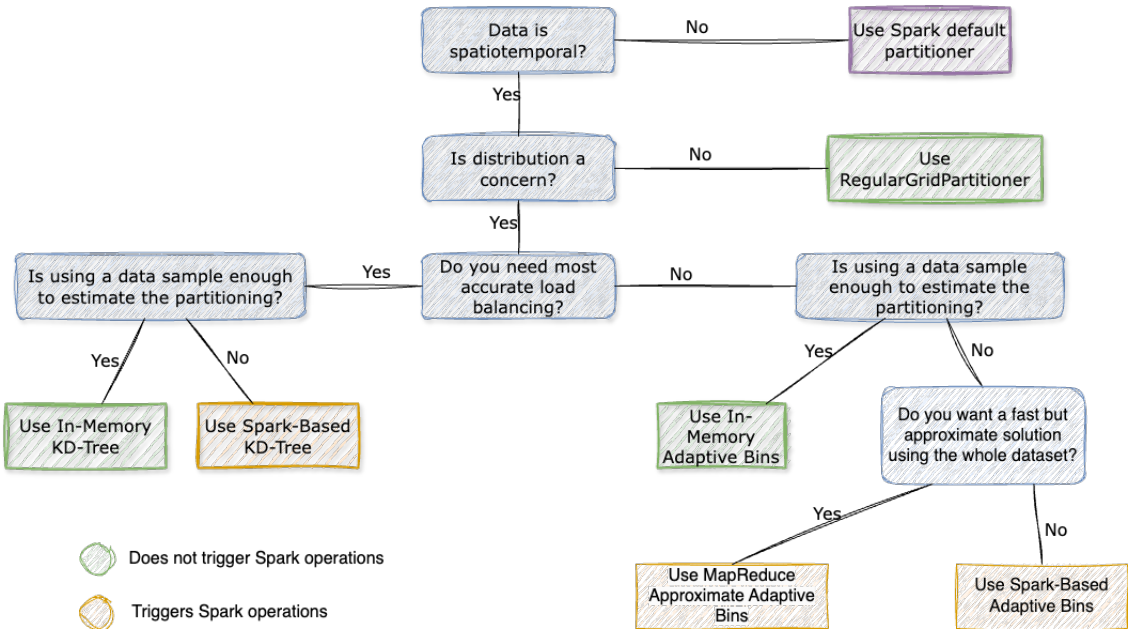

Figure 4.1: Decision Diagram to choose a MobilitySpark Partitioner.

In conclusion, to summarize these research goals, Figure 4.1 provides a practical guide on selecting the appropriate partitioner based on different use case scenarios. This figure serves as a recommendation tool, helping users decide the most suitable partitioning strategy for their specific needs, thereby maximizing the effectiveness of their mobility data management and analysis efforts.

4.3 Challenges Encountered During the Project

Before outlining potential areas for improvement and future work that could build upon MobilitySpark, it is important to reflect on the challenges faced during the implementation of this project and how these obstacles were addressed to ultimately produce a functional first version of MobilitySpark.

The most significant challenge encountered was selecting the appropriate toolset between PyMEOS/PySpark and JMEOS/Spark. As previously mentioned, PyMEOS/PySpark was ultimately chosen. However, ideally, the implementation would leverage Spark's native code, which is primarily accessible through Java or Scala. Utilizing Scala alongside JMEOS to trigger MEOS functions could have allowed for numerous optimizations. Additionally, direct modifications to Spark's Catalyst optimizer could facilitate the implementation of specific spatiotemporal rules for various operations, enhancing the system's efficiency.

Another challenge arose from the nature of Spark itself. Although Spark and PySpark are powerful frameworks, the documentation and error tracing are often insufficient, particularly when issues arise. For instance, memory errors in Spark are not accompanied by detailed error messages, leading to the application simply shutting down. This lack of specific error information requires developers to conduct extensive research to identify the root cause of the problem.

Running MEOS locally also presented a unique challenge. At the project's inception, MEOS lacked compatibility with Mac computers equipped with M-Series chips. As the local development was conducted on such a Mac, a solution was found by containerizing the development environment, thereby resolving the compatibility issues.

Moreover, Spark's codebase and documentation have significant room for improvement. Many functions required in-depth investigation, often requiring a review of the source code due to inadequate public documentation. For example, the `partitionBy` method in PySpark's RDD API is poorly documented[2]. This required thorough examination of the *rdd.py* file to understand the function's internal workings. While reading and interpreting code is a standard part of the research process, the lack of comprehensive documentation adds an extra layer of complexity to the research and development process.

In summary, the project encountered multiple challenges, from toolset selection and framework limitations to documentation gaps and compatibility issues. Addressing these challenges has not only highlighted the areas in need of improvement but also provided valuable insights into the potential for further development and refinement of the MobilitySpark system.

[2] `https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.RDD.partitionBy.html`

4.4 Future Work

Having discussed the challenges encountered during the development of MobilitySpark, we now turn our attention to potential future enhancements for MobilitySpark, MEOS, and the broader field of mobility data research. These suggestions are based on the insights gained from the project's results and the obstacles faced along the way.

1. *Standardize MobilitySpark.* A critical next step in strengthening the framework is to fully standardize MobilitySpark with MEOS. Currently, MobilitySpark has only partial compatibility with MEOS, with several user-defined types (UDTs) and user-defined functions (UDFs) yet to be implemented. For example, while this project has focused extensively on partitioning trajectories, leading to the comprehensive mapping of functions related to `TPoint` and `STBox`, other data types like `TInt` have not been similarly addressed. A future iteration of MobilitySpark should aim to include a complete suite of MEOS data types and functions, ensuring a more robust and versatile tool for users.

2. *Implementation of MobilitySpark using Java/Scala.* Another promising direction for future development is the creation of a MobilitySpark version that uses JMEOS/Spark as its core engine. This approach would be a natural progression for the project, leveraging Spark's native capabilities to maximize MEOS's potential. The primary advantage of this transition would be the elimination of unnecessary data transfers between the Java Virtual Machine (JVM) and the Python environment in Spark. Currently, these transfers create a bottleneck due to the serialization and deserialization processes required for Java to interact with MEOS data types and functions. A Java/Scala-based implementation would streamline these processes, resulting in more efficient data handling and potentially significant performance gains.

3. *Research on partitioners using the Java/Scala version of MobilitySpark.* Developing partitioners directly in Spark using Java or Scala could significantly enhance performance, particularly for trajectory partitioning. Future research could focus on implementing the partitioners explored in this thesis within Spark's native framework. To achieve this, these partitioners would need to extend Spark's `Partitioner` class, allowing them to be utilized in conjunction with the `partitionBy` function, which is currently limited in PySpark. This capability would enable more sophisticated partitioning strategies and improve the efficiency of data processing in distributed environments.

4. *Optimize MobilitySpark DataFrame operations with Catalyst.* An essential goal for future work is to integrate spatiotemporal awareness into Spark's Catalyst optimizer. By developing specific rules and optimizations for handling MEOS data types, Catalyst could more effectively manage join operations and resource allocation. This enhancement would not only streamline query execution but also optimize the overall performance of data processing tasks involving complex spatiotemporal data. For example, integrating custom Catalyst rules could improve the efficiency of operations such as spatiotemporal joins and aggregations, leading to faster query times and reduced computational overhead.

4.5 Final Remarks

Throughout this thesis, we have explored the growing significance of mobility data, which has become increasingly relevant in recent years. The analysis and management of spatiotemporal data demand substantial storage and processing resources, making the use of distributed systems a natural necessity, much like other Big Data applications.

The first section of the thesis, Chapter 2, was dedicated to review the state of the art and important concepts involving mobility data, distributed frameworks, and partitioning strategies. This introductory chapter allows for a initial understanding of the proposed solution in Chapter 3.

The success of mobility data systems, both current and future, stems on their ability to provide a comprehensive and versatile environment. This includes offering users a wide range of options across multiple languages and platforms. MEOS and MobilityDB are at the forefront of this task. The addition of a new binding, *MobilitySpark*, to the MEOS family is a significant milestone. MobilitySpark is designed to extend the capabilities of the MEOS framework into a distributed environment, specifically leveraging the powerful analytic capabilities of Spark.

Consider the value to a user who can seamlessly process any dataset using a unified suite of libraries, confident in the accuracy and efficiency of the data processing. This vision is being realized with MEOS and, now, with the integration of MobilitySpark. This system not only broadens the scope of MEOS but also enhances its application by providing distributed processing capabilities.

The partitioning strategies evaluated in this project address a critical challenge in mobility data research: determining the most effective methods for splitting spatiotemporal data and distributing it across a cluster. This is a crucial aspect of mobility data management. Our study of these partitioners, with MEOS as the foundational technology, highlights the potential for optimized data handling in distributed systems.

In conclusion, the future of mobility data is inextricably linked to the evolution of distributed databases. As Big Data continues to grow, particularly in the domain of mobility data, effective management strategies become ever more critical. Research in mobility data management and distributed systems will mutually reinforce each other, driving advancements in both fields. *MobilitySpark* stands as a pioneering effort in integrating a robust library with a powerful distributed framework, setting the stage for future innovations in this dynamic area of study.

This work not only lays a foundation for further exploration but also emphasizes the importance of collaboration between mobility data management and distributed computing. As we continue to develop and refine these technologies, the insights gained from MobilitySpark will undoubtedly contribute to shaping the next generation of mobility data solutions, ensuring that they are both comprehensive and efficient.

A

Annex

A.1 Installing MobilitySpark

MobilitySpark may be installed[1] through Docker, or locally. It is also possible to install it in a cloud environment such as Google Colab or Databricks Notebook by downloading the repository and performing the local installation.

A.1.1 Through Docker

The project can be run locally using Docker. This method is preferred if you have a Mac with an M-Series Chip.

1. Download this repository.
2. `cd` into the *container* folder.
3. Build the image, name it as you prefer, for example *mobilityspark*:

   ```
   docker build -t <LOCAL-NAME-OF-THE-IMAGE> <PATH-TO-Dockerfile>
   ```

   Example:

   ```
   docker build -t mobilityspark ./
   ```

4. Run the container:

   ```
   docker run \
       -v <VOLUME-PATH> \
       -p <JUPYTERLAB-PORT-FORWARD> \
       -p <SPARK-UI-PORT-FORWARD> \
       <CONTAINER-NAME>
   ```

   Example:

   ```
   docker run \
       -v $(pwd)/../../../data:/data \
       -p 8887:8888 \ # For JupyterLab
       -p 4040:4040 -p 4041:4041 -p 4042:4042 \ # For SparkUI
       pyrtitionfinal
   ```

   This example, for instance, runs the project mapping the volume `../../../data` to the `/data` folder inside the container. This is useful if you want to use local files inside your container. Also, you need to specify:

   - Ports for the JupyterLab (recommended `-p 8887:8888`)
   - Ports for the Spark UI (recommended `-p 4040:4040`)

[1] For extra information, go to the README file in the project repository: https://github.com/Action52/MobilityPySpark

You can reserve any number of ports for different Spark sessions, as in the example. This command will run the container and will end up by printing a token to start the JupyterLab. Copy the token.

5. Go to your browser, and go to `localhost:8887`, or the port corresponding to your JupyterLab.
6. Insert the generated token where asked. The JupyterLab will prompt.
7. Now you can run the notebooks of the project by going to `/notebooks` inside the JupyterLab.

Overall, the commands to install through Docker could be as follows:

```
# Clone first the repository and checkout to the latest tag.
git clone https://github.com/Action52/MobilityPySpark
git checkout tags/v0.1.1

cd MobilityPySpark

# Build the image
docker build -t mobilityspark ./

# Run the container
docker run \
    -v $(pwd)/../../../data:/data \
    -p 8887:8888 -p 4040:4040 -p 4041:4041 -p 4042:4042 \
    pyrtitionfinal

# Here you can optionally access the container directly through the console,
#or through the JupyterLab instance (recommended)
docker exec -it mobilityspark /bin/bash

# Run script or notebook with MobilitySpark
```

A.1.2 Locally

To install locally:

1. Download this repository.
2. `cd` into the main folder of the repository.
3. In a clean Python environment, run:

   ```
   pip install .
   ```

   This will install MobilitySpark and the required packages in your environment.
4. If you desire to run the notebooks, now you can do, with your env activated:

```
jupyter notebook
```

5. Now you can run the notebooks of the project by going to `/notebooks` inside your local Jupyter server.

Overall, the commands to install locally could be as follows:

```
# Use a clean environment
conda create --name mobilitypyspark python=3.9
conda activate mobilitypyspark

# Clone the repository and checkout to the latest tag
git clone https://github.com/Action52/MobilityPySpark
cd MobilityPySpark
git checkout tags/v0.1.1

# Install the package
pip install .

# Run script or notebook with MobilitySpark
```

A.2 MobilitySpark Data Types

Table A.1 presents MobilitySpark's available data types as of version *v0.1.1*. These data types constitute a broad enough introduction to the system, and are able to perform the experiments on partitioning data, while also allowing the user to execute data science tasks of simple and medium complexity.

MobilitySpark UDT	**PyMEOS Data Type**
TGeomPointInstUDT	TGeomPointInst
TGeomPointSeqUDT	TGeomPointSeq
TGeomPointSeqSetUDT	TGeomPointSeqSet
TGeogPointInstUDT	TGeogPointInst
TGeogPointSeqUDT	TGeogPointSeq
TGeogPointSeqSetUDT	TGeogPointSeqSet
TFloatInstUDT	TFloatInst
STBoxUDT	STBox
TsTzSpanUDT	TsTzSpan
TBoolInstUDT	TBoolInst

Table A.1: MobilitySpark v0.1.1 data types.

A.3 MobilitySpark User-Defined Functions

Table A.2 shows the UDFs in MobilitySpark *v0.1.1*, indicating if the function is only a wrapper or if it incorporates custom behavior.

Function Name	MobilitySpark Return Type	Is Only Wrapper
create_point_udf	TGeomPointInstUDT()	Yes
create_pointseq_instants_udf	IntegerType()	
get_point_x	FloatType()	Yes
get_point_y	FloatType()	Yes
get_point_z	FloatType()	Yes
get_point_timestamp	TimestampType()	Yes
bounds_as_box	STBoxUDT()	
ever_touches	BooleanType()	Yes
tstzspan	TsTzSpanUDT()	Yes
geometry_from_hexwkb	GeometryUDT()	Yes
trip_from_hexwkb	TGeomPointSeqUDT()	Yes
tpoint_at	GeometryUDT()	Yes
temporally_contains	BooleanType()	Yes
tgeompointseq_from_instant_list	TGeomPointSeqUDT()	
point_to_stbox	STBoxUDT()	Yes
tboolinst_from_base_time	TBoolInstUDT()	Yes
ever_intersects	BooleanType()	Yes
min_distance	FloatType()	Yes
nearest_approach_distance	FloatType()	Yes
tgeompointinst	TGeomPointInstUDT()	Yes
tgeompointseq	TGeomPointSeqUDT()	Yes
at_geom	TGeomPointSeqSetUDT()	Yes
contains_stbox_stbox	BooleanType()	Yes
temporally_overlaps	BooleanType()	Yes
datetime_to_tinstant	TBoolInstUDT()	Yes
timestamps	ArrayType(TimestampType())	Yes
spatial_values	ArrayType(FloatType())	Yes
tgeompointseqset	TGeomPointSeqSetUDT()	Yes

Table A.2: MobilitySpark v0.1.1 UDF functions.

References

[1] M. Bakli, M. Sakr, and E. Zimányi, "Distributed Spatiotemporal Trajectory Query Processing in SQL," in *28th International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*. ACM, 2020, p. 87–98.

[2] E. Zimányi, M. Duarte, and V. Diví, "MEOS: An Open Source Library for Mobility Data Management," in *27th International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 2024, pp. 810–813.

[3] A. Vaisman and E. Zimányi, *Data Warehouse Systems: Design and Implementation*. Springer Nature, 2022.

[4] C. Renso, S. Spaccapietra, and E. Zimányi, *Mobility Data*. Cambridge University Press, 2013.

[5] A. Graser, A. Naghibzadeh-Jalali, J. Lampert, A. Weißenfeld, and K. Janowicz, "Deep Learning from Trajectory Data: a Review of Deep Neural Networks and the Trajectory Data Representations to Train Them," in *Workshop on Big Mobility Data Analytics (BMDA)*, 2023.

[6] E. Zimányi, M. Sakr, and A. Lesuisse, "MobilityDB: A Mobility Database based on PostgreSQL and PostGIS," *ACM Transactions on Database Systems (TODS)*, vol. 45, no. 4, pp. 1–42, 2020.

[7] M. S. Bakli, M. Sakr, and E. Zimányi, "Distributed Moving Object Data Management in MobilityDB," in *8th ACM International Workshop on Analytics for Big Geospatial Data (SIGSPATIAL)*. ACM, 2019, pp. 1:1–1:10.

[8] M. Bakli, M. Sakr, and E. Zimányi, "Distributed Mobility Data Management in MobilityDB," in *21st IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 2020, pp. 238–239.

[9] M. Mokbel, M. Sakr, L. Xiong, A. Züfle, J. Almeida, T. Anderson, W. Aref, G. Andrienko, N. Andrienko, Y. Cao *et al.*, "Mobility Data Science (Dagstuhl Seminar 22021)," in *Dagstuhl reports*, vol. 12, no. 1. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[10] R. H. Güting and M. Schneider, *Moving Objects Databases*. Elsevier, 2005.

[11] International Organization for Standardization, "ISO 19141:2008 Geographic Information — Schema for Moving Features," 2008.

[12] S. Ceri and G. Pelagatti, "Correctness of Query Execution Strategies in Distributed Databases," *ACM Transactions on Database Systems (TODS)*, vol. 8, no. 4, p. 577–607, 1983.

[13] S. Phansalkar and S. Ahirrao, "Survey of Data Partitioning Algorithms for Big Data Stores," in *2016 Fourth International Conference on Parallel, Distributed and Grid Computing (PDGC)*. IEEE, 2016, pp. 163–168.

[14] S. Ponnusamy and P. Gupta, "Scalable Data Partitioning Techniques for Distributed Data Processing in Cloud Environments: A Review," *IEEE Access*, vol. 12, pp. 26 735–26 746, 2024.

[15] M. S. Mahmud, J. Z. Huang, S. Salloum, T. Z. Emara, and K. Sadatdiynov, "A Survey of Data Partitioning and Sampling Methods to Support Big Data Analysis," *Big Data Mining and Analytics*, vol. 3, no. 2, pp. 85–101, 2020.

[16] J. L. Bentley, "Multidimensional Binary Search Trees used for Associative Searching," *Communications of the ACM*, vol. 18, no. 9, p. 509–517, 1975.

[17] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," in *1984 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 1984, p. 47–57.

[18] M. Ding and S. Chen, "Efficient Partitioning and Query Processing of Spatio-Temporal Graphs with Trillion Edges," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 1714–1717.

[19] T. Mallick, P. Balaprakash, E. Rask, and J. Macfarlane, "Graph-Partitioning-Based Diffusion Convolutional Recurrent Neural Network for Large-Scale Traffic Forecasting," *Transportation Research Record: Journal of the Transportation Research Board*, vol. 2674, no. 9, p. 473–488, 2020.

[20] K. Hori, Y. Sasaki, D. Amagata, Y. Murosaki, and M. Onizuka, "Learned Spatial Data Partitioning," in *Sixth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM@SIGMOD)*. ACM, 2023, pp. 3:1–3:8.

[21] Z. Al Aghbari, T. Ismail, and I. Kamel, "SparkNN: A Distributed In-Memory Data Partitioning for KNN Queries on Big Spatial Data," *Data Science Journal*, vol. 19, pp. 35–35, 2020.

[22] B. Wei, J. Zhang, C. Hu, and Z. Wen, "A Clustering Visualization Method for Density Partitioning of Trajectory Big Data Based on Multi-Level Time Encoding," *Applied Sciences*, vol. 13, no. 19, 2023.

[23] T. Vu, A. Belussi, S. Migliorini, and A. Eldway, "Using Deep Learning for Big Spatial Data Partitioning," *ACM Transactions on Spatial Algorithms and Systems*, vol. 7, no. 1, 2020.

[24] E. Haines, "Point in Polygon Strategies." *Graphics Gems*, vol. 4, no. 1, pp. 24–46, 1994.

[25] N. Razavi and M. Johns, "Processing Geospatial Data at Scale With Databricks," https://www.databricks.com/blog/2019/12/05/processing-geospatial-data-at-scale-with-databricks.html, Databricks, 2019, Databricks Solutions Blog.

[26] M. Colic, R. Whiffin, P. Patel, C. Doidge, S. Kingston, and L. Sheard, "Efficient Point-in-Polygon Joins via PySpark and BNG Geospatial Indexing," https://www.databricks.com/blog/2021/10/11/efficient-point-in-polygon-joins-via-pyspark-and-bng-geospatial-indexing.html, Databricks, 2021, Databricks Engineering Blog.

[27] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *2nd USENIX conference on Hot topics in cloud computing (HotCloud)*, 2010, pp. 10–10.

[28] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, vol. 13, no. 6, p. 377–387, 1970.

[29] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, p. 107–113, 2008.

[30] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *19th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2003, pp. 20–43.

[31] 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 *2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 2015, pp. 1383–1394.

[32] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant abstraction for In-Memory Cluster Computing," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012, pp. 15–28.

[33] E. Zimányi, M. Sakr, A. Lesuisse, and M. Bakli, "MobilityDB: A Mainstream Moving Object Database System," in *16th International Symposium on Spatial and Temporal Databases (SSTD)*. ACM, 2019, p. 206–209.

[34] E. Zimányi, M. Sakr, and A. Lesuisse, "MobilityDB: A Mobility Database Based on PostgreSQL and PostGIS," *ACM Transactions on Database Systems*, vol. 45, no. 4, 2020.

[35] L. Alarabi, M. F. Mokbel, and M. Musleh, "ST-Hadoop: a MapReduce Framework for Spatio-temporal Data," *GeoInformatica*, vol. 22, no. 4, pp. 785–813, 2018.

[36] L. Alarabi, "Summit: A Scalable System for Massive Trajectory Data Management," *SIGSPATIAL Special*, vol. 10, no. 3, p. 2–3, 2019.

[37] M. S. Bakli, M. A. Sakr, and T. H. A. Soliman, "HadoopTrajectory: A Hadoop Spatiotemporal Data Processing Extension," *Journal of Geographical Systems*, vol. 21, no. 2, pp. 211–235, 2019.

[38] Z. Zhang, C. Jin, J. Mao, X. Yang, and A. Zhou, "TrajSpark: A Scalable and Efficient In-Memory Management System for Big Trajectory Data," in *Web and Big Data - First International Joint Conference (APWeb-WAIM)*, ser. Lecture Notes in Computer Science, vol. 10366. Springer, 2017, pp. 11–26.

[39] X. Ding, L. Chen, Y. Gao, C. S. Jensen, and H. Bao, "UlTraMan: A Unified Platform for Big Trajectory Data Management and Analytics," *Proceedings of the Very Large Data Bases (VLDB) Endowment*, vol. 11, no. 7, p. 787–799, 2018.

[40] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient In-Memory Spatial Analytics," in *2016 International Conference on Management of Data (SIGMOD)*. ACM, 2016, p. 1071–1085.

[41] Z. Shang, G. Li, and Z. Bao, "DITA: Distributed In-Memory Trajectory Analytics," in *2018 International Conference on Management of Data (SIGMOD)*, 2018, pp. 725–740.

[42] A. Eldawy, V. Hristidis, S. Ghosh, M. Saeedan, A. Sevim, A. Siddique, S. Singla, G. Sivaram, T. Vu, and Y. Zhang, "Beast: Scalable Exploratory Analytics on Spatio-temporal Data," in *30th ACM International Conference on Information Knowledge Management (CIKM)*. ACM, 2021, p. 3796–3807.

[43] C. Düntgen, T. Behr, and R. H. Güting, "Berlinmod: A benchmark for moving object databases," *The Very Large Data Bases (VLDB) Journal*, vol. 18, pp. 1335–1368, 2009.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to God for the strength, wisdom, and perseverance granted to me throughout this journey. Without His guidance and blessings, this thesis and masters degree would not have been possible.

I am profoundly grateful to my family, whose support has been my greatest source of encouragement. To my parents, Alfredo and Maribel, for their endless love and belief in me; to my aunts, Tere and Ale, for their constant encouragement and understanding; and to my dear "Socio" my uncle Eduardo, for being a companion in all my endeavors. Your love and support have been pillars of my strength.

A special thank you goes to my friends, Carlo and Juan Pablo; who have been an integral part of this journey. Your friendship and support have been invaluable, providing me with much-needed encouragement and moments of joy along the way.

I would also like to express my heartfelt appreciation to my supervisors, Esteban and Geppino, for their guidance, insightful feedback, and support throughout this research. Your mentorship has been instrumental in shaping this thesis. Additionally, I extend my gratitude to Professors Mahmoud, Sergi, and Mariangela. You have not only been exemplary educators but also models of kindness and integrity, inspiring me to strive for excellence both academically and personally.

Most importantly, this thesis is dedicated in loving memory to my beloved aunt Tere. Her boundless love, support, and unwavering belief in me have been a guiding light throughout my life. Though she is no longer with us, her memory continues to inspire and motivate me. This work is a testament to her enduring love and the profound impact she had on my life. Rest in peace, dear Tere; this is for you.

Sincerely, Luis