UNIVERSITÉ LIBRE DE BRUXELLES

Master's Thesis

Visualization of MobilityDB Data in QGIS

Ali Imam MANZER

Under the supervision of
M. Esteban ZIMANYI

Academic year
2023 - 2024

I hereby declare on my honor that the work presented in this thesis is the result of my own personal and individual effort. All sources of information and literature used have been properly cited and referenced.

Brussels, August 2024

Ali Imam MANZER

Abstract

This master thesis presents comprehensive research on visualizing spatiotemporal data from MobilityDB within QGIS, a leading open-source Geographic Information System (GIS) tool. The study explores the current state of visualization methods for MobilityDB data in QGIS, as well as other prominent tools such as OpenLayers, Deck.gl, and Leaflet. Motivated by the recent development of MEOS, an open-source C library designed for mobility data management, this work leverages the capabilities of MEOS, which decouples spatiotemporal data handling from MobilityDB. This decoupling enables direct interaction with spatiotemporal objects and methods through various programming environments. The thesis utilizes PyMEOS and PyQGIS, which are Python bindings for MEOS and QGIS libraries, respectively. Additionally, it revisits previous efforts, notably the MOVE plugin by Maxime Schoemans, which allows QGIS users to visualize and animate MobilityDB's moving objects using the Temporal Controller.

We propose several solutions for integrating MobilityDB data into QGIS, with a particularly effective design based on dividing the timeline into subsets called Time Deltas. This approach allows for continuous and unrestricted animation of spatiotemporal data, simultaneously minimizing memory usage. The thesis illustrates how these solutions enhance visualization and animation capabilities in QGIS, significantly extending the functionality of the MOVE plugin. We present a thorough explanation of how MobilityDB and PyMEOS's `value_at_timestamp` and `tsample` operations are utilized to compute trajectory positions, and showcase how the extensive features of PyQGIS are applied to create efficient and practical plugins for QGIS using Python.

Acknowledgments

I would like to start by expressing my gratitude to my family and friends for their unwavering support and encouragement throughout my academic journey. Their belief in me has been a constant source of motivation and strength. I would also like to extend my sincere thanks to my thesis supervisor, Esteban Zimanyi, and my thesis assistant, Maxime Schoemans. Their expertise, invaluable advice, and continuous encouragement guided me throughout this process and were instrumental in the completion of this thesis. Their mentorship has been invaluable, and I am deeply grateful for their contributions.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In today's world, mobility data has become an essential component of our daily lives. From smartphones to GPS-enabled devices, the proliferation of location-based services has led to an exponential increase in the volume of mobility data available. This data encompasses a wide array of information, including the movement of individuals, vehicles, goods, and more, across geographical spaces over time. Such rich datasets offer invaluable insights into mobility patterns, transportation networks, urban planning, and environmental impacts, among others. To handle such large volumes of data, traditional relational database systems can prove to be inefficient and cumbersome. Recently, the development of spatiotemporal database systems has emerged as a promising solution to this problem. These systems are designed to store and manage spatiotemporal data, providing efficient querying and analysis capabilities.

MobilityDB [27], specifically designed to manage mobility data, is an open-source extension of PostGIS, which is a powerful spatial extension to PostgreSQL. It enhances PostgreSQL by introducing support for spatiotemporal data types and functions, thereby facilitating the storage and manipulation of mobility data. The introduction of these spatiotemporal data types brings exciting challenges in the field of visualization. MobilityDB is part of the growing trend of open-source software solutions. Therefore, it is essential to explore how its data types can be visualized using QGIS, one of the most widely used open-source visualization tools in the industry.

1.1 Motivation and Challenges

QGIS is one of the most popular open-source Geographical Information System (GIS) tools, offering a wide range of features for visualizing spatial data. In the recent version 3.14 [1], QGIS introduced a temporal controller, allowing users to interact with temporal data more intuitively. This feature enables users to visualize spatial data from traditional spatial databases by also considering the associated temporal properties, making it easier to understand the evolution of the data.

However, QGIS does not natively support the spatiotemporal data types from MobilityDB, preventing their direct visualization. Previous efforts to address this limitation include the work of Ludéric Van Calck [6] and the MOVE plugin by Maxime Schoemans [20], which we detail in the following chapter. These solutions, however, have their own issues and fundamentally do not provide support for visualizing and animating very large spatiotemporal datasets. With the recent

[1] *https://blog.qgis.org/2020/06/25/qgis-3-14-pi-is-released/*

development of PyMEOS [26], we are able to manage and interact with MobilityDB's spatiotemporal objects and functions directly within Python. Our goal with this research project is to build on the existing research in the field to create a robust and efficient solution for visualizing MobilityDB objects within QGIS.

1.2 Objectives

The primary objective of this thesis is to propose a self-contained downloadable QGIS plugin that enables the visualization and animation of MobilityDB's spatiotemporal data by utilizing the built-in Temporal Controller. Achieving this objective entails addressing the challenges listed below. Each of these challenges requires careful consideration and innovative solutions within the limitations of our tools.

- **Data and Memory Management**: Handling and processing large volumes of spatiotemporal data is a core challenge. The system must efficiently manage data retrieval and storage, ensuring quick access and minimal latency while minimizing memory usage during the animation.
- **Fluid Animation**: Ensuring seamless and smooth animation of moving objects demands highly optimized code to maintain a consistently high and stable frame rate. The challenge lies in sustaining smooth and responsive interactions, especially when dealing with large datasets.
- **Scalability and Adaptability**: The plugin must be scalable to handle varying data sizes and adaptable to different hardware configurations. This involves designing a flexible architecture that can scale both in terms of hardware and dataset, ensuring performance on all combinations of data and hardware.
- **User Experience and Interface Design**: Providing an intuitive and user-friendly interface is vital for effective visualization. This includes interactive controls for temporal navigation without delays and load times, and ensuring overall usability of the plugin while maintaining all functionalities of the Temporal Controller itself.
- **Benchmarking and Evaluation**: Rigorous testing and benchmarking are necessary to evaluate the performance and reliability of the solution. This includes assessing metrics such as data processing speed, average frame rate, stability of the frame rate, memory consumption, and system responsiveness.

Resolving QGIS-PyMEOS Incompatibility

Initially, we encountered a significant issue when importing PyMEOS into QGIS's Python environment, which led to instant crashes. The issue was related to the compilation of PyMEOS, which created a naming conflict between the libraries used by both QGIS and PyMEOS. A temporary fix we used was the Subprocess

library for Python, which allowed us to run any scripts containing PyMEOS in an external Python environment to avoid QGIS crashes. Ultimately, the issue was resolved with the remarkable work from Maxime Schoemans, Prof. Esteban Zimanyi and Victor Divi (creator of PyMEOS).

1.3 Organization

Our goal is to make this report as self-contained as possible, ensuring that a reader with a basic understanding of Computer Science will be able to read and comprehend its content without much difficulty. Accordingly, we have structured and organized the report as follows:

1. **Chapter 1: Introduction** - Provides an overview of the thesis, including the background, the motivation behind the project, and the objectives.
2. **Chapter 2: State of the Art** - Offers an in-depth review of MobilityDB and explores the field of MobilityDB data visualization within the most popular visualization tools: OpenLayers, Leaflet, deck.gl, and most importantly, QGIS.
3. **Chapter 3: Technologies** - Details the tools used to develop our solutions, including an introduction to GIS, PyMEOS, PyQGIS, and vector tiles servers.
4. **Chapter 4: Setup Overview** - Describes the different hardware configurations used to test and develop the solutions, as well as the datasets utilized.
5. **Chapter 5: Vector Layer** - Details two implementation ideas: the Interactive Mode, which provides high interactivity with the temporal controller but at the cost of high memory usage, and the TimeDeltas Mode, which offers animations of unlimited duration for extremely large datasets but introduces a load time delay whenever the Temporal Controller configuration is changed.
6. **Chapter 6: Vector Tile Layer** - Describes the vector tiles server solution we developed using QGIS's vector tile layer and PG tileserv.
7. **Chapter 7: Resampling Trajectories** - Explains the Resampling Mode implementation, a solution that is inspired from the TimeDeltas Mode presented in Chapter 5, incorporating MobilityDB's `tsample` operation to compute positions.
8. **Chapter 8: Conclusions** - Summarizes the final plugin implementation and provides insights into future improvements and areas of research.

1.4 Thesis Outcomes

This master's thesis has successfully extended the MOVE plugin [20] [2] to enhance the QGIS platform, enabling users to visualize and animate spatiotemporal data from MobilityDB using the Temporal Controller functionality. The final version of MOVE includes the implementation of the Interactive Mode(detailed in Section 5.1), which delivers smooth and efficient animations while allowing users to interact seamlessly with the Temporal Controller. To ensure the transparency and reproducibility of this research, all developed code and related resources have been made publicly available in a dedicated GitHub repository [3]. This repository serves as a valuable resource for those interested in exploring the technical aspects of the implementation, as well as for researchers looking to verify or build upon the findings of this thesis.

[2] *https://github.com/mschoema/move*
[3] *https://github.com/amanzer/mobilitydb-qgis-visualization*

Chapter 2

State of the Art

Traditional Relational Database Management Systems (RDBMS) provide a comprehensive suite of security features and guarantees regarding data structures. PostgreSQL is a notable open-source RDBMS that benefits from numerous extensions addressing specific complex issues. One such extension is PostGIS, which allows users to handle spatial data effectively. In this thesis report, we will focus on MobilityDB, an extension for PostgreSQL that builds on top of PostGIS and provides spatiotemporal types to handle the abundant spatial data of today's data-rich world.

2.1 MobilityDB

MobilityDB [27] is a moving objects database [15] that extends PostgreSQL and PostGIS (see Figure 2.1). It enriches the PostgreSQL platform with abstract data types (ADTs) designed specifically for representing moving object data. These types integrate seamlessly into the platform, leveraging existing data management features and future improvements. For instance, the tgeompoint type in MobilityDB builds on the PostGIS geometry type, restricted to 2D/3D points, and utilizes PostGIS's coordinate system transformation capabilities.

The current implementation of MobilityDB includes six types: temporal geometry point (tgeompoint), temporal geography point (tgeogpoint), temporal integer (tint), temporal float (tfloat), temporal boolean (tbool), and temporal text (ttext). This extensible type system is complemented by a rich set of functions for SQL queries. These functions utilize the existing operations, indexing, and optimization frameworks of PostgreSQL and PostGIS, meaning any enhancements to these underlying systems benefit MobilityDB directly.

MobilityDB adheres to the ongoing Open Geospatial Consortium (OGC) standards on Moving Features [2], ensuring compatibility and adherence to industry standards. As open-source software, it is accessible for both research and practical industry applications. The project began in 2016 with the vision of making moving object database research accessible to users within a mainstream DBMS framework like PostgreSQL and PostGIS.

[1] Source: lecture notes from INFO-H417 - Database systems architecture by Prof. Mahmoud Sakr, 2021-2022

[2] *https://www.ogc.org/standard/movingfeatures/*

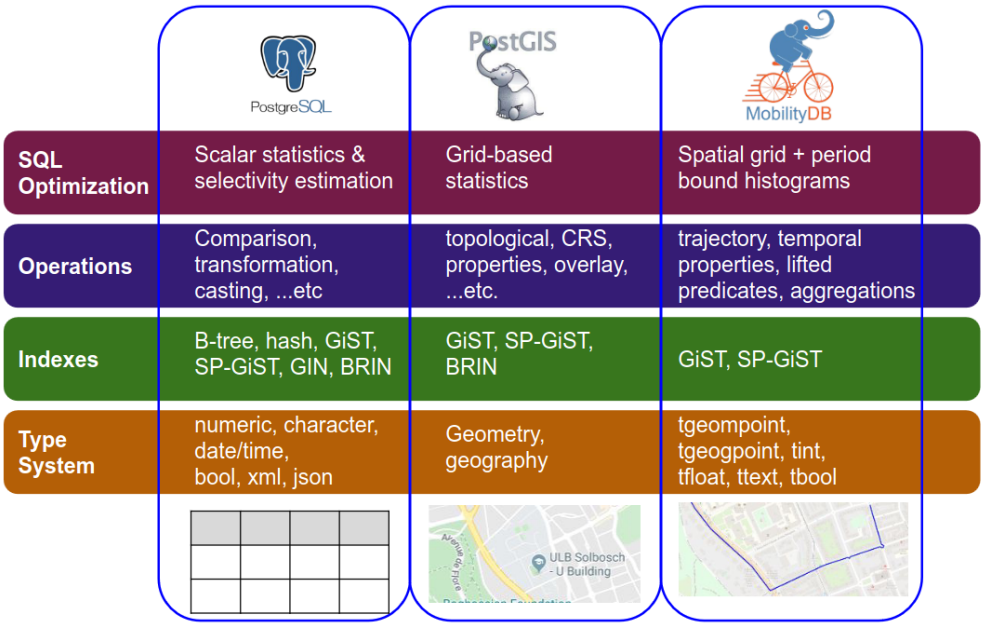


Figure 2.1: PostgreSQL, PostGIS and MobilityDB [1]

2.1.1 Features and Performance Enhancements

MobilityDB leverages an abstract data type approach to implement time-varying data types using a discrete data model known as sequence representation. This model includes a rich set of operators available to users as SQL functions and benefits from the powerful indexing and optimization framework of PostgreSQL and PostGIS. This integration results in an efficient and expressive moving object database.

In addition to its core features, MobilityDB includes specialized temporal types and bounding boxes to improve performance, especially for indexing operations. Bounding boxes optimize queries and reduce computational overhead. For example, the `tbox` type is used for temporal integer and float types, where the value extent is defined in one dimension and the temporal extent in another. Similarly, the `stbox` type is used for spatiotemporal types, combining spatial and temporal extents.

2.1.2 Moving Objects: Temporal Types

MobilityDB enhances the traditional point-based representation, where each GPS record of a trajectory is stored separately with its corresponding timestamp, by adopting a sequence-based approach. In this method, the trajectory of a moving object is represented as a sequence of spatial location and timestamp pairs, encapsulated within a single spatiotemporal object. An illustration of this can be seen in Figure 2.2. This sequence-based representation allows for the efficient storage of an entire trajectory within a single database row. Moreover, it enables the use of advanced spatiotemporal analysis tools, facilitating the querying and examination

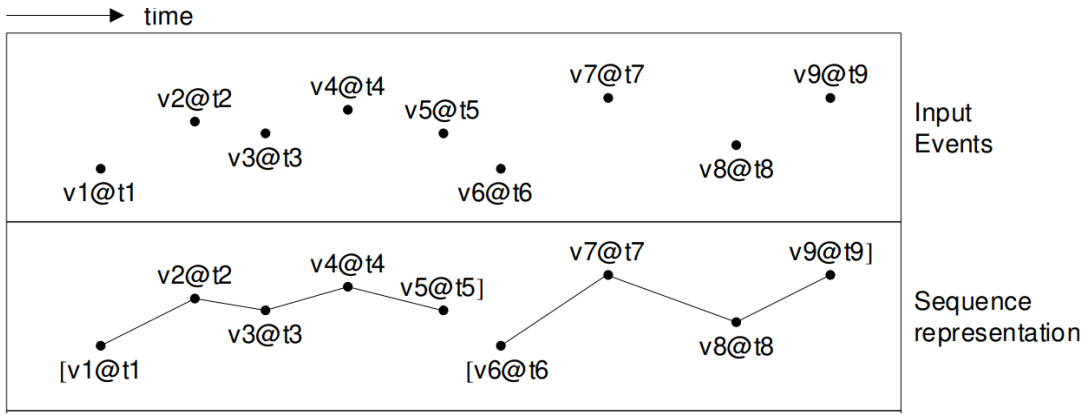


Figure 2.2: Point based vs sequence representation [29]

of object movement patterns over time.

2.1.3 Practical Examples

By allowing the representation of entire trajectories in a single row, MobilityDB facilitates efficient storage and querying of spatiotemporal data, enabling sophisticated analyses such as trajectory tracking, movement pattern identification, and proximity queries. MobilityDB's temporal types offer a powerful means to model and analyze the dynamic behavior of moving objects in various applications, enabling interesting research studies like the analysis of public transport in the city of Buenos Aires [12], or the analysis of aviation trajectories [5].

Detecting Close Encounters

Detecting close encounters between moving objects can be effectively managed using MobilityDB. For example, Anita Grazer's blog [13] illustrates how to calculate the closest points of approach (CPAs) between trajectories. MobilityDB provides functions like shortestLine to compute the line connecting the nearest points between two tgeompoint_seq trajectories. Additionally, the twavg function helps compute time-weighted average movement speeds to filter out stationary or very slowly moving objects.

For example, to compute CPAs between ships (with MMSI being the distinct ID), the following query can be used:

```
SELECT S1.MMSI mmsi1, S2.MMSI mmsi2,
      shortestLine(S1.trip, S2.trip) Approach,
      ST_Length(shortestLine(S1.trip, S2.trip)) distance
FROM Ships S1, Ships S2
WHERE S1.MMSI > S2.MMSI AND
     twavg(S1.SOG) > 1 AND twavg(S2.SOG) > 1 AND
     dwithin(S1.trip, S2.trip, 0.003);
```

This query finds the closest points of approach between ships with average speeds greater than 1 and within a certain distance threshold. The resulting query

layer can be visualized in QGIS to analyze the encounters.

Measuring a Billboard's Visibility to a Moving Vehicle

Mahmoud Sakr provides an example [3] to measure the visibility of a billboard to a moving vehicle. This involves analyzing the spatiotemporal interaction between the vehicle's trajectory and the billboard's location. By leveraging MobilityDB's capabilities, the visibility duration of billboards from moving vehicles can be quantified.

For instance, consider a scenario where we have the trajectory of a bus passing by a billboard. Using MobilityDB, we can implement a query to find the time periods during which the bus is within a certain distance (e.g., 30 meters) from the billboard. In this example, the tdwithin function determines if the bus is within 30 meters of the billboard, and the atperiodset function restricts the trip to these time periods. The astext function then converts the coordinates to a textual format, showing the specific times and locations where the billboard is visible to the bus passengers.

2.2 Visualization of Moving Objects

Visualization is a powerful tool for exploring and understanding spatiotemporal data. It allows users to see patterns, trends, and anomalies that are not apparent in raw data. Effective visualization can facilitate decision-making, communication, and collaboration. Since MobilityDB extends the existing PostgreSQL and PostGIS tools, the existing visualization solutions do not natively support spatiotemporal types. Numerous research efforts have aimed to provide implementations that enable the visualization of MobilityDB data within popular tools. In this section, we describe these efforts.

2.2.1 OpenLayers

OpenLayers is an open-source JavaScript library for displaying map data in web browsers. Soufian El Bakkali Tamara's work [10] integrates MobilityDB with OpenLayers to visualize spatiotemporal data (see Figure 2.3). This integration facilitates dynamic visualizations of MobilityDB's spatiotemporal objects on interactive maps, enabling users to see animation of trajectories.

The implementation requires setting up the project environment using npm. After cloning the repository, users need to install the necessary dependencies and build the project. The visualizations can be customized by modifying the dataset variables in the JavaScript files. This setup allows the rendering of complex spatiotemporal data, such as AIS datasets and urban mobility data, providing a comprehensive view of moving objects over time.

[3] *https://techcommunity.microsoft.com/t5/azure-database-for-postgresql/analyzing-gps-trajectories-at-scale-with-postgres-mobilitydb-amp/ba-p/1859278*

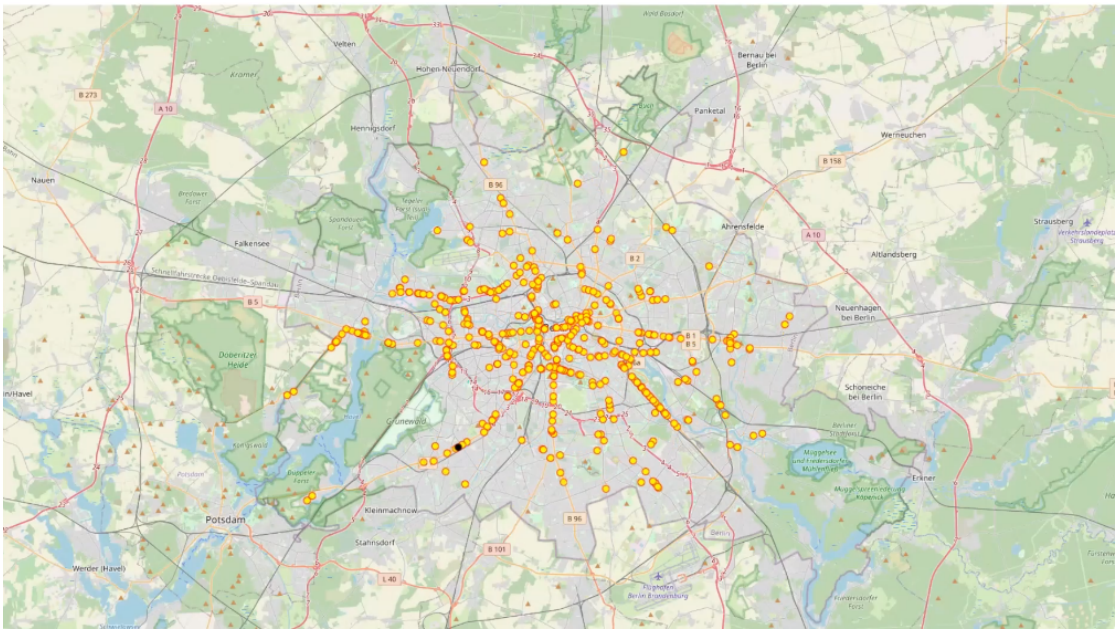


Figure 2.3: OpenLayers solution [10]

2.2.2 Leaflet

Leaflet is another popular open-source JavaScript library for interactive maps. Florian Baudry's work [4] provides a robust integration of MobilityDB with Leaflet, enabling efficient and dynamic visualization of moving objects. The MobilityDB-Leaflet tool is designed with a focus on usability and performance. It offers features such as temporal control, object filtering, and dynamic visualization. Users can control the time frame for the visualized data, play, pause, and navigate through time, and filter objects to manage the visual load. This tool is structured with a React frontend and a FastAPI backend, making it highly extendable and easy to set up. The backend handles data queries and processing, ensuring that the visualizations remain responsive even with large datasets. The best use case for the Leaflet solution is when developing an application that requires interactive maps with temporal controls and for projects where ease of setup and use is prioritized.

2.2.3 Deck.gl

Deck.gl is a WebGL-powered framework designed for the visual exploratory data analysis of large datasets. Fabrício Ferreira da Silva's work [22] on integrating MobilityDB with Deck.gl leverages the framework's capabilities to efficiently handle and visualize large volumes of spatiotemporal data.

The proposed solution incorporates a visualization framework using a spatiotemporal vector tiling strategy to minimize data transfer and enhance performance. By processing data on the database side using SQL queries, the system can filter, transform, and aggregate data effectively before visualization. Two architectural solutions are proposed: one utilizing a tile server for handling extensive

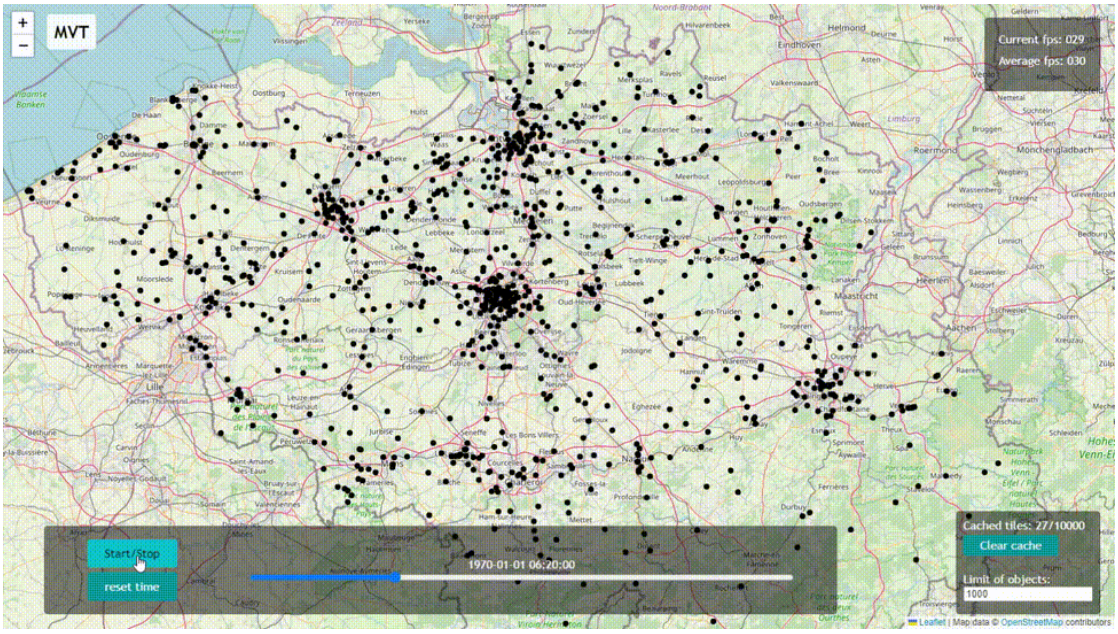


Figure 2.4: Leaflet solution [4]

data volumes, and another using an in-memory tile index for fast rendering. This setup allows for the efficient visualization of trajectories with millions of points, significantly outperforming existing solutions in terms of scalability and interaction speed.

The project setup involves configuring a PostgreSQL database with the PostGIS and MobilityDB extensions, setting up a`pg_tileserv` server for serving map tiles, and running the Deck.gl frontend. The best use case of Deck.gl is for large-scale visualization and analysis, and for scenarios requiring high performance and advanced visualizations thanks to the highly optimized WebGL framework.

2.3 QGIS

QGIS is one of the most popular open-source GIS tools, providing a host of features that enable spatial analysis and visualization. However, like the tools mentioned earlier, it does not natively support MobilityDB's data types. This research provides a solution that allows users to view their MobilityDB data inside QGIS by downloading a plugin from the QGIS market. However, this is not the first research that attempted to do this. Let's look at previous research projects that delivered solutions to this problem and which also directly or indirectly had a significant role in our final solution.

QGIS-MobilityDB Project

Ludérick Van Calck conducted a research project [6] focused on animating MobilityDB trajectories in QGIS using the temporal controller. The goal was to explore ways to visualize moving objects automatically in QGIS by linking MobilityDB

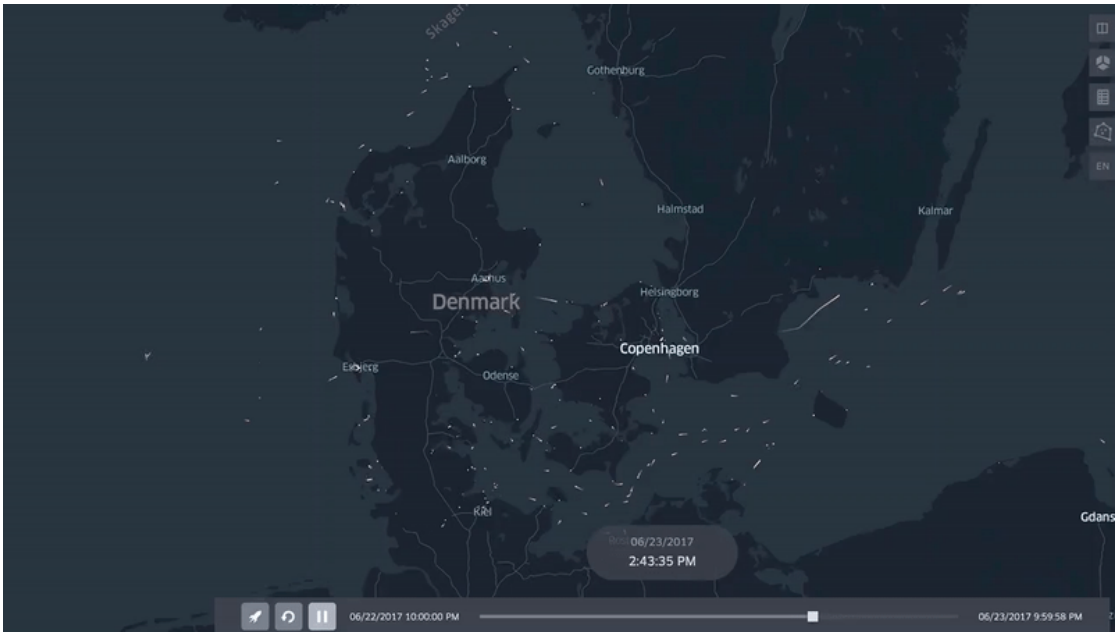


Figure 2.5: Deck.gl solution [22]

data with QGIS layers through the PyQGIS Python library and interpolating the positions via the MobilityDB Python driver (now replaced by PyMEOS) or directly through the database. This integration required transforming MobilityDB's spatiotemporal trajectories into geometries that QGIS could recognize and display.

Despite some challenges, the project demonstrated the feasibility of integrating MobilityDB with QGIS for visualizing moving object trajectories and included many useful elements:

- **on_new_frame function**: This method is connected to the Temporal Controller's update signal, enabling custom computation at each state change of the Temporal Controller.
- **On-the-Fly Interpolation**: This method involved interpolating trajectory points at each new frame, providing real-time updates but with performance limitations. The interpolation was done every time the on_new_frame function was called.
- **Buffering Frames**: This method involved interpolating trajectory points for a fixed number of frames (N) and buffering them. The interpolation and feature addition were done once every N frames.
- **Database-Side Interpolation**: This method involved querying the interpolation of trajectories directly from the database using the postgisexecuteandloadsql algorithm provided by QGIS.

Vector Layer with SQL Query

Anita Graser [14] proposed a simple and effective idea in which she uses a query layer connected to MobilityDB, we attach a SQL query that fetches the positions of the spatiotemporal objects for a given timestamp. After this, using the same principle as the on_new_frame from Ludérick's work, we attach the frame change signal of the Temporal Controller using PyQGIS, to a custom function which updates the SQL query's timestamp value. This solution is very elegant because it doesn't require any additional installation from the user and works on any QGIS installation. The unfortunate drawback of this solution is that as the dataset size scales up, performance significantly degrades.

MOVE Plugin

One of the most significant contributions to this field is the MOVE plugin developed by Maxime Schoemans for visualizing spatial and spatiotemporal columns from MobilityDB inside QGIS [20]. Figure 2.6 shows the Interface of MOVE, it provides users with a dock where they can write their SQL queries. If the query output is a PostGIS spatial data column, it simply creates a normal vector layer with the associated database URI immediately. However, if the output is a MobilityDB-specific spatiotemporal column, MOVE first creates a materialized view containing regular PostGIS geometries for different intervals of time, enabling users to visualize the spatiotemporal objects via the Temporal Controller. Users can then adjust the frame rate of the temporal controller and play the animation to see the moving objects.

MOVE has its pros and cons. The advantages include providing a simple and effective way for users to visualize MobilityDB trajectories inside QGIS by downloading a plugin. However, the main disadvantage is that it has to generate the materialized view containing all the points for different times intervals initially, leading to high initial load time and memory costs as we scale the dataset size.

Choosing the Right Tool for the Job

We have described multiple tools that can enable MobilityDB users to visualize their spatiotemporal data, and one might ask which tool is the best. It is important to emphasize that there is no one-size-fits-all solution. We encourage MobilityDB users to try different tools to determine which tool best suits their needs. Here are our recommendations on which tool to use depending on various scenarios.

1. **Leaflet**: Ideal for applications requiring ease of use and interactive features, with a user-friendly interface and sufficient capabilities for moderate-sized datasets. Suitable for web integration with simpler requirements.

2. **OpenLayers**: Recommended for web integration due to its flexibility and extensive customization options for complex web-based applications.

3. **Deck.gl**: Best suited for large-scale data visualization and high-performance applications, offering advanced visualization capabilities and scalability.

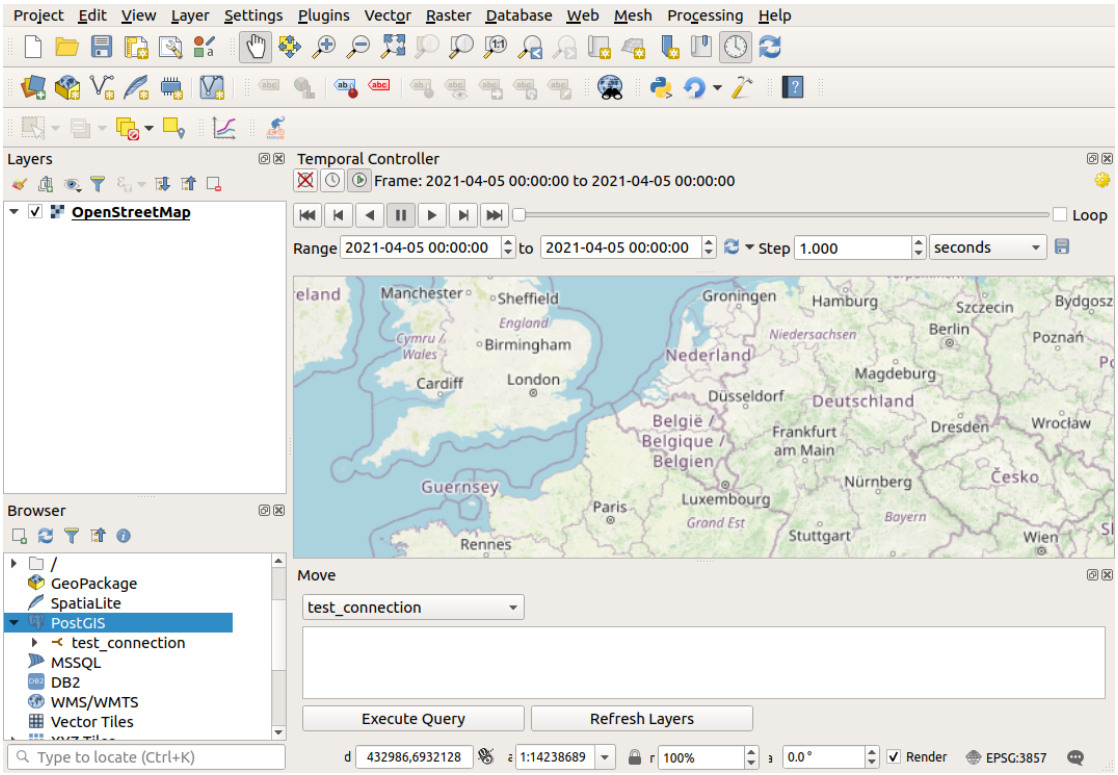


Figure 2.6: Interface of MOVE plugin

4. **QGIS**: Perfect for comprehensive spatial analysis, providing an extensive suite of GIS tools ideal for desktop-based projects.

As we will show in later chapters, QGIS is not the best tool if you are looking for a high and stable animation frame rate with support for extremely large datasets. For such cases, Deck.gl is more suitable due to its superior performance and scalability. Additionally, QGIS is not easily incorporated into web applications. However, if you are looking for a way to visualize spatiotemporal data inside a multiplatform GIS without requiring any programming knowledge or additional installations, then the solution we propose in this research paper is ideal. Our approach leverages the strengths of QGIS to provide an accessible and powerful tool for visualizing MobilityDB data.

2.4 Summary of the State of the Art

This chapter presents a state-of-the-art review covering MobilityDB, its spatiotemporal objects, and the current state of research into the integration of MobilityDB with the most popular visualization tools. Current solutions exist for OpenLayers and Leaflet for web-based applications, providing interactive maps and geospatial data visualization. Additionally, Deck.gl offers high-performance large-scale data visualization for extremely large datasets. A significant focus of this chapter is the existing research on the visualization of MobilityDB data within QGIS. Currently

the most advanced solution is the MOVE plugin, however in its current state it has some vital flaws, specifically a high memory cost as well as poor animation performances when scaling the dataset size. Our aim with this research project is to extend MOVE and provide QGIS users with the ability to view MobilityDB data and animate them through a simple downloadable plugin.

In the upcoming chapter, we cover in detail the core technologies used in the solutions presented in this Thesis, we also provide a section detailing QGIS. GIS systems are invaluable for spatial data analysis due to their robust capabilities in managing, analyzing, and visualizing spatial information. By integrating MobilityDB with QGIS, we enable MobilityDB users to perform advanced spatiotemporal data analysis and visualization inside a strong GIS tool.

Chapter 3

Technologies

This chapter details the technical tools we used in order to achieve the solutions presented in the research paper. It starts with an introduction to Geographical Information System (GIS) tools and QGIS, describing how these systems model the spatial information and provide a plethora of features to represent reality inside a computer. We present an advanced mobility data management tool called MEOS. Since QGIS plugin development is done using the Python programming language, we also describe PyMEOS and PyQGIS, the Python-specific libraries for MEOS and QGIS development, respectively. While this report does not include a tutorial or annex for learning Python, a basic understanding of programming is sufficient to follow the content presented. Any necessary Python-specific concepts or library behaviors will be explained as needed.

3.1 Geographical Information System

This section provides an introduction to Geographical Information Systems (GIS) tools, such as QGIS. The following content is inspired by these lecture notes:

1. Course of Geospatial and Web Technologies (2022-2023) by Prof. Mahmoud Sakr.
2. Course of Advanced Databases (2022-2023) by Prof. Esteban Zimanyi.

Additionally, we also use the well-written documentation of QGIS [1], which relies on these resources:

1. Introduction to Geographic Information Systems [7]
2. Fundamentals of Geographic Information Systems [8]
3. Geographic Information Systems Demystified [11]

All the images in this chapter come from either the QGIS documentation or the lecture notes.

GIS systems are used to handle spatial information on a computer. A GIS application can open digital maps, create new spatial information to add to a map, create customized printed maps, and perform spatial analysis.The power of GIS also lies in its ability to associate non-geographical (attribute) data with

[1] *https://docs.qgis.org/3.34/en/docs/gentle_gis_introduction/index.html*

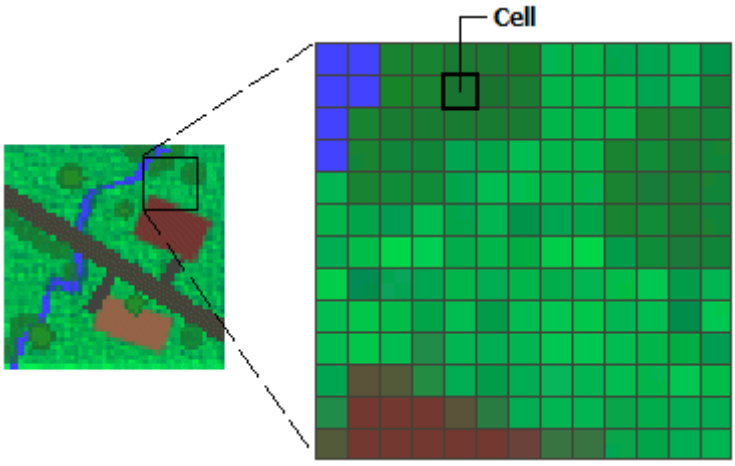


Figure 3.1: Example of raster data [2]

geographical (spatial) data, enabling comprehensive spatial analysis. Most GIS systems work with many different types of data. Vector data is stored as a series of (X, Y) coordinate pairs and is used to represent points, lines, and areas. Raster data is stored as a grid of values where each cell or pixel contains a value representing information such as temperature or the pixel value of a satellite picture. All raster data has a specific resolution (see Figure 3.1).

3.1.1 Vector Data

Vector data provides a way to represent real-world features. For example, Figure 3.2 shows how different features from a landscape are presented in a GIS: rivers (blue) and roads (green) can be represented as lines, trees as points (red), and houses as polygons (white).

A vector feature has its shape represented using geometry, which is made up of one or more interconnected vertices. These geometries can represent different types of real-world features, which are categorized into various vector data types such as points, linestrings, and polygons, each serving a specific purpose in mapping and spatial analysis. Figure 3.3 shows the common architecture of the geometry class as defined by the OGC-2006 [17].

Types of Vector Features

1. **Point Features**: The representation of point features in GIS depends largely on scale, convenience, and feature type. For example, cities may be represented as points on a small-scale map but as polygons on a larger scale. Points are convenient and quick to create compared to polygons. A point feature has X, Y, and optionally Z values, depending on the Coordinate

[2] *https://desktop.arcgis.com/fr/arcmap/latest/manage-data/raster-and-images/what-is-raster-data.htm*

[3] *https://docs.qgis.org/3.34/en/docs/gentle_gis_introduction/vector_data.html#symbology*

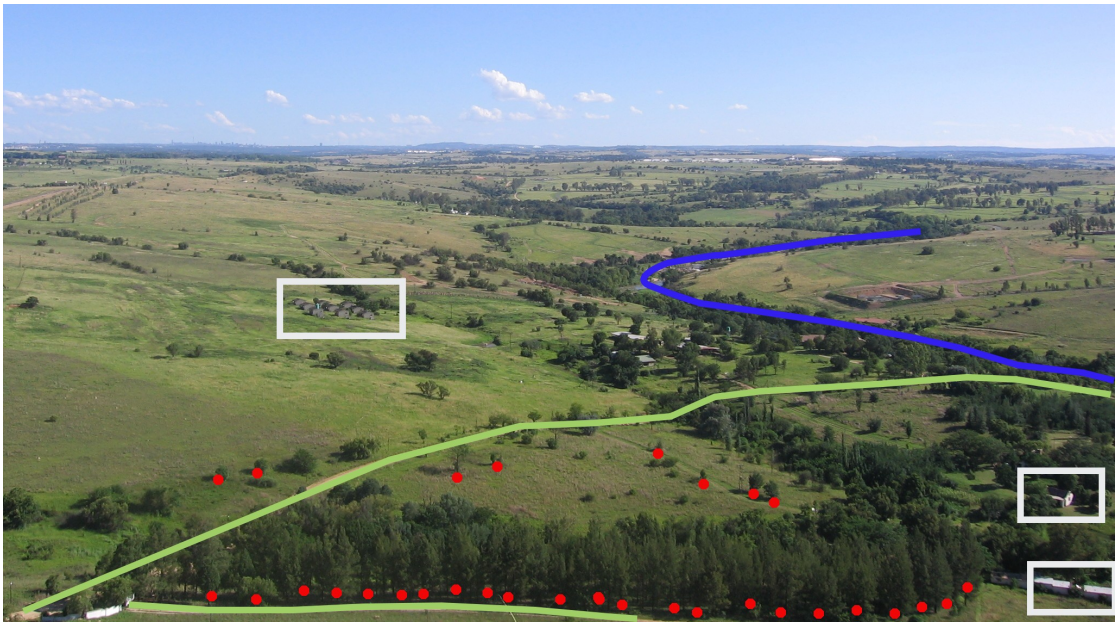


Figure 3.2: Illustration of how real-world features are presented in a GIS [3]

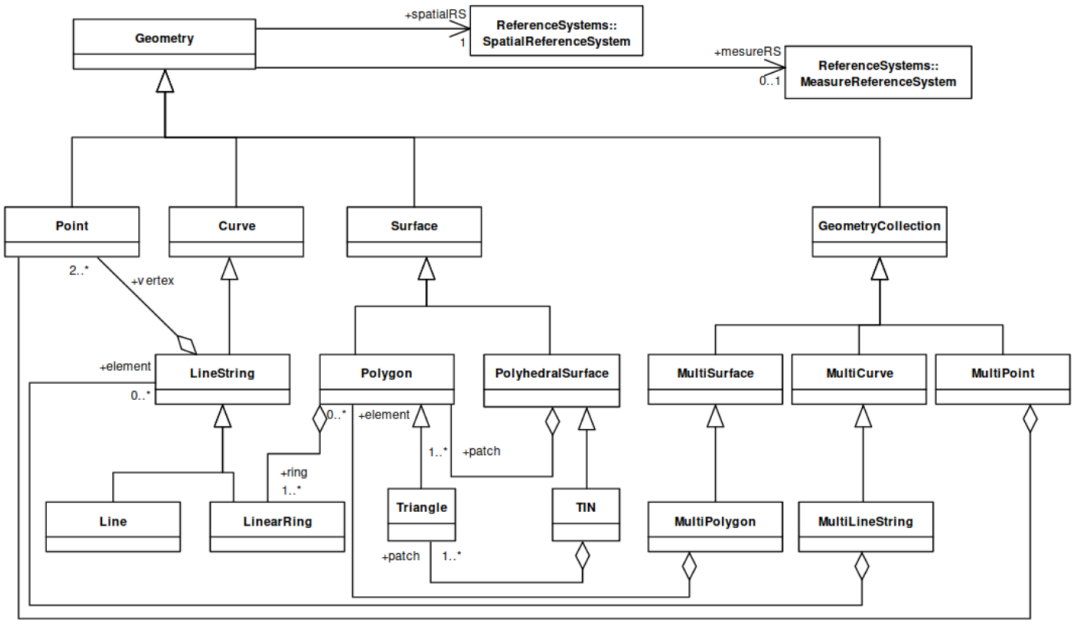


Figure 3.3: Geometry class hierarchy OGC-2006 [17]

Reference System (CRS) used, such as Longitude and Latitude. The CRS accurately describes locations on the Earth's surface, and adding a Z value can help indicate elevation above sea level for example.

2. **Polyline Features**: A polyline consists of two or more vertices connected in a continuous path and is used to represent linear features such as roads, rivers, and flight paths. Special rules can be applied to polylines to ensure they conform to specific requirements, such as contour lines not crossing

each other or road networks connecting at intersections. The appearance of polylines can vary with scale, so they should be digitized with appropriate vertex spacing. Attributes of polylines describe their characteristics, like surface type or number of lanes, and can be used by GIS to symbolize these features appropriately.

3. **Polygon Features**: Polygons represent enclosed areas such as dams, islands, and country boundaries. They are created by connecting a series of vertices in a continuous line, with the first and last vertices being the same to enclose the area. Polygons often share boundaries with neighboring polygons, and many GIS applications ensure these shared boundaries coincide precisely. Like points and polylines, polygons have attributes that describe their characteristics, such as temperature of a region.

Data Formats

- **WKT (Well-Known Text)**: A human-readable text format that describes vector geometries.
- **WKB (Well-Known Binary)**: A compact binary format for vector geometries, suitable for efficient storage and processing.

Vector Data in Layers

Vector data in a GIS environment is typically managed and organized into layers. Each layer groups features with the same geometry type, such as points, lines, or polygons, and similar attributes. This organization allows for efficient management, as entire layers can be shown or hidden with a single click.

GIS applications provide tools for creating and modifying vector data, a process known as digitizing. When editing a layer, the application enforces rules to ensure data integrity, such as requiring lines to have at least two vertices or only allowing polygons in a polygon layer.

Utility and Symbology of Vector Data

One of the significant advantages of using a GIS is the ability to modify the appearance of vector layers to create personalized maps. Initially, vector layers are drawn with random colors and basic symbols, but GIS applications allow users to customize these visualizations to suit the data type. For instance, you can set water bodies to be drawn in blue or represent tree positions with small tree icons instead of basic circles. This is known as symbology, it enhances the interpretability and aesthetics of the maps, making the data easier to understand and more visually appealing. Symbology settings can be adjusted using dedicated panels within the GIS application, allowing for detailed and meaningful representations of various data layers.

Beyond visual customization, the utility of vector data in GIS is immense, particularly in spatial analysis. Vector data allows for complex queries and analyses

such as determining which houses are within a flood zone, identifying optimal locations for facilities like hospitals, or mapping the distribution of students in a suburb. These capabilities leverage vector data to provide insights and answers to geographic questions, enhancing decision-making processes. However, vector data must be accurate and reliable, which requires meticulous data capture and maintenance. Common problems like slivers, undershoots, and overshoots can occur if data is not carefully digitized. These errors, visible at larger scales, can undermine the quality and reliability of the spatial analysis. The OGC's DE-9IM (Dimensionally Extended nine-Intersection Model), shown in Figure 3.4, provides a framework to understand and manage the topological relationships between geometric entities, ensuring the integrity of vector data in GIS applications.

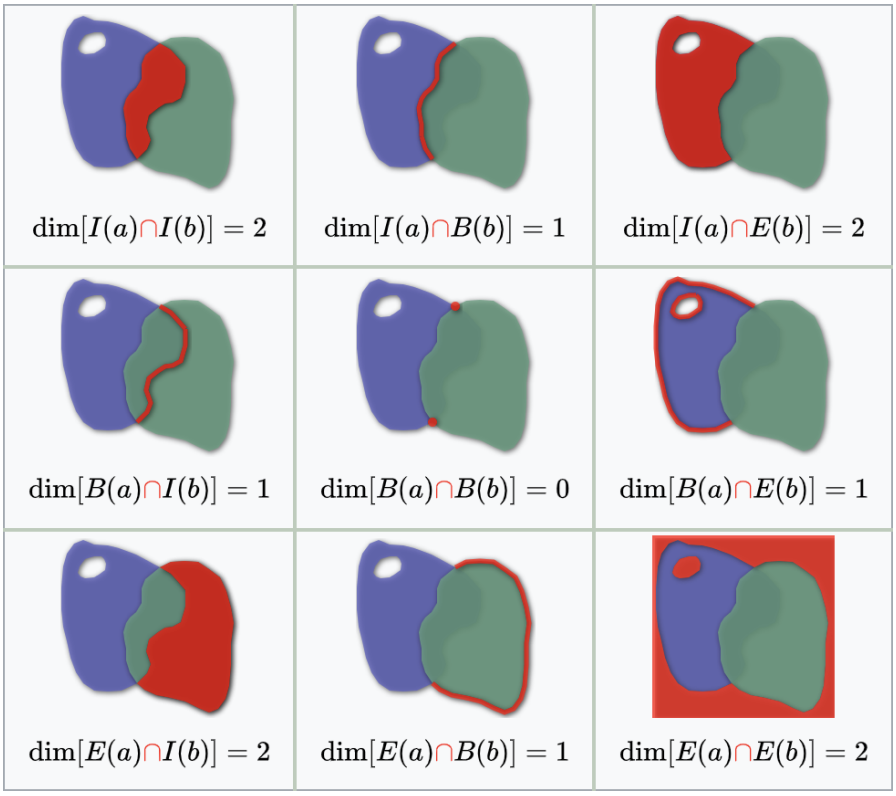


Figure 3.4: DE-9IM from OGC [4]

Vector Attribute Data

Vector data in GIS is greatly enhanced by the use of attribute data, which allows for detailed and informative maps. This attribute data is stored in tables within the GIS, with each row representing a feature and each column representing a property of the feature. As shown in Figure 3.5, if every line on a map had the same color, width, thickness, and label, it would be very difficult to understand the map and extract useful information. Symbology, which involves using different colors

[4] *https://postgis.net/docs/manual-2.5/using_postgis_dbmanagement.html*

and symbols to represent various features, makes maps informative and easier to interpret. For instance, using distinct colors and symbols for rivers, roads, and contours allows users to differentiate between these features effortlessly, enhancing the map's clarity and usability.

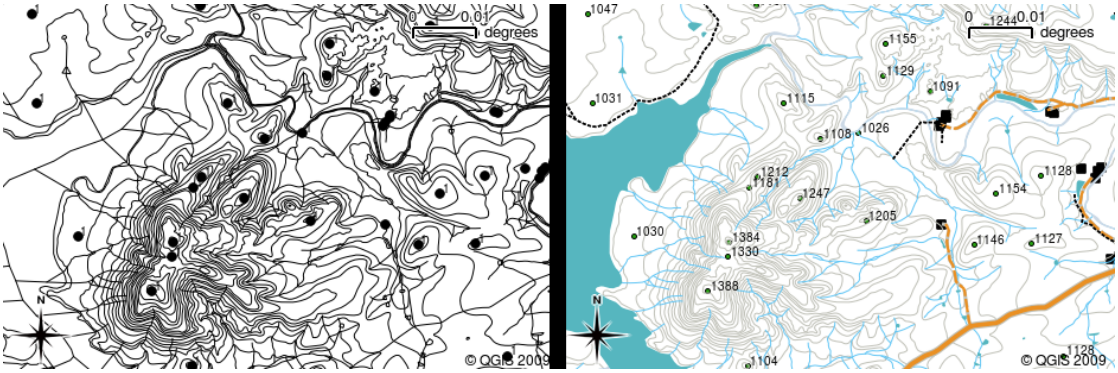


Figure 3.5: Impact of symbology to better understand a map [5]

Attributes enable symbology, where features can be drawn with colors and symbols based on their properties, making maps visually appealing and easier to interpret. For instance, houses can be color-coded by roof type or age, and contour lines can be shaded based on elevation.

3.1.2 Topology

Topology in GIS defines the spatial relationships between connecting or adjacent vector features such as points, polylines, and polygons. This is crucial for detecting and correcting digitizing errors and performing spatial analysis. It ensures that features like road intersections and property boundaries are accurately represented without gaps or overlaps. Tools and rules in GIS applications help maintain correct topology by preventing errors such as unclosed polygons and intersecting contour lines. Parameters like snapping distance and search radius aid in precise digitizing, ensuring features align correctly. Accurate topological data is essential for reliable spatial analyses, such as network routing and area measurement, enhancing the overall utility of GIS data.

3.1.3 Coordinate Reference Systems

Understanding how to accurately represent the Earth's surface on a flat map is crucial in GIS, and this is where Coordinate Reference Systems (CRS) come into play. Map projections transform the Earth's three-dimensional spherical shape into a two-dimensional plane, making it possible to display on paper or computer screens. A CRS defines how the two-dimensional, projected map relates to real-world locations on Earth. The choice of CRS and map projection depends on various factors, including the regional extent of the area, the type of analysis being conducted, and the availability of data.

[5] *https://docs.qgis.org/3.34/en/docs/gentle_gis_introduction/vector_attribute_data.html*

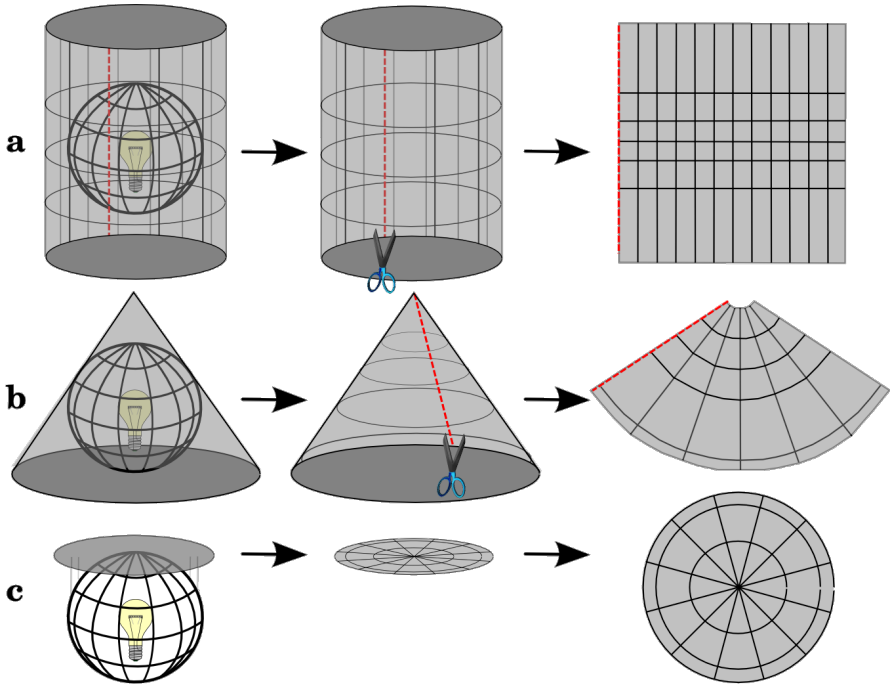
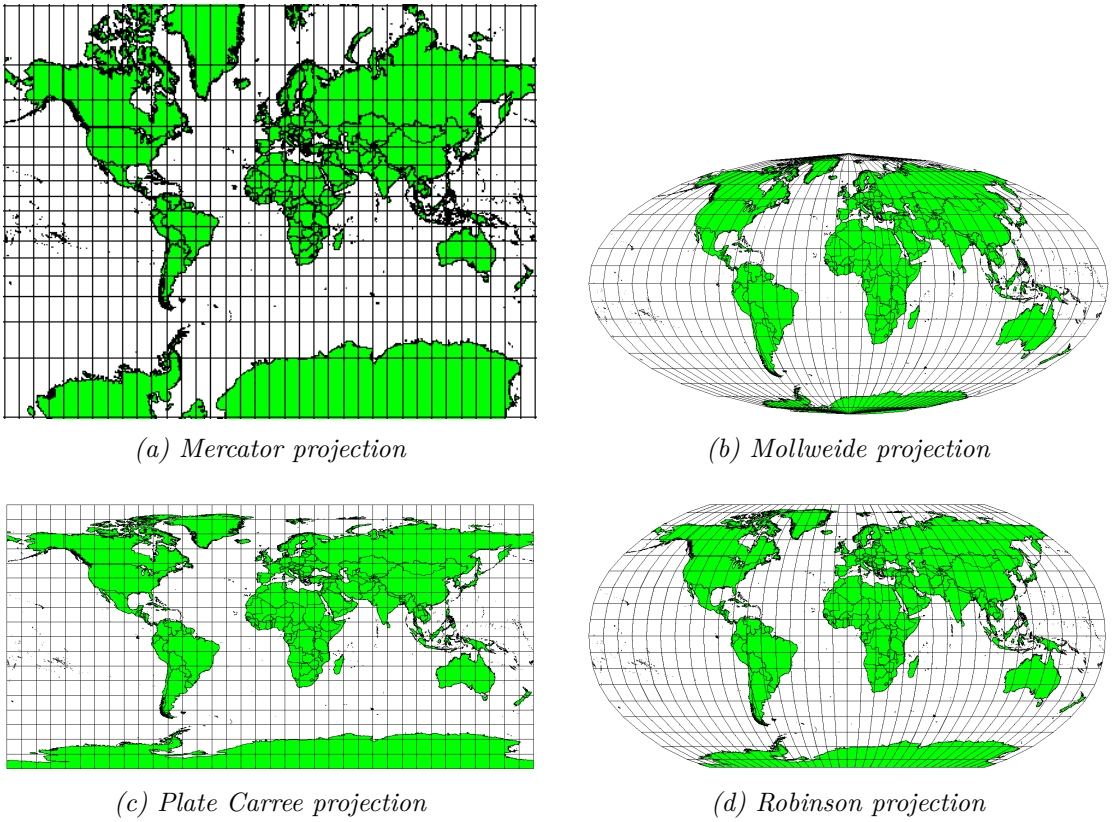


Figure 3.6: Different families of projections : a) cylindrical projections, b) conical projections, and c) planar projections [6]

As shown in Figure 3.7, map projections are categorized into different families based on how they project the Earth's surface onto a flat plane: cylindrical, conical, and planar projections (see Figure 3.6). Each type has its strengths and weaknesses, often trading off between preserving angular conformity, distance, or area. For example, the Mercator projection (Figure 3.7a) is well-known for preserving angles, making it useful for navigation, but it distorts areas, especially near the poles. On the other hand, equidistant projections, such as the Plate Carree (Figure 3.7c), maintain consistent distances from the center of the map, which is crucial for applications requiring accurate distance measurements. Equal area projections like the Mollweide (Figure 3.7b) ensure that all mapped areas retain their proportional size, making them ideal for analyses involving area calculations.

The accuracy of a map projection is never perfect due to the distortions that occur when representing a curved surface on a flat plane. These distortions can affect angular relationships, distances, and areas. Some map projections, like the Robinson projection (Figure 3.7d), attempt to balance these distortions, providing a compromise that makes them suitable for world maps. However, choosing the right projection is critical for specific tasks; for instance, using a conformal projection like Mercator for detailed area measurement would lead to significant errors. Thus, the selection of a map projection should align with the map's intended purpose to ensure the accuracy and reliability of the spatial analysis.

[7] *https://docs.qgis.org/3.34/en/docs/gentle_gis_introduction/coordinate_reference_systems.html#map-projection-in-detail*



(a) Mercator projection

(b) Mollweide projection

(c) Plate Carree projection

(d) Robinson projection

Figure 3.7: Various map projections [7]

3.1.4 Representation of Earth's Shape

The Earth's shape is highly irregular due to variations in density and gravitational forces, which makes it challenging to represent with simple mathematical formulas. To address this, two main reference surfaces are used: the ellipsoid and the geoid, as shown in Figure 3.8. The ellipsoid is a mathematically defined, smooth, oblate spheroid that approximates the Earth's shape and is used as a reference for mapping and navigation because of its simplicity. However, it doesn't account for the Earth's irregularities. The geoid, on the other hand, is a more accurate but complex representation of the Earth's shape, defined as the hypothetical sea level surface where the gravitational potential is constant. It reflects the Earth's true shape by accounting for variations in the gravitational field caused by factors like mountain ranges and ocean trenches. The difference between the geoid and a reference ellipsoid is known as geoid undulation. The choice between using an ellipsoid or geoid depends on the application: the ellipsoid is typically used for most mapping and GPS tasks, while the geoid is crucial for high-precision tasks such as surveying, where gravitational variations must be considered for accurate height measurements.

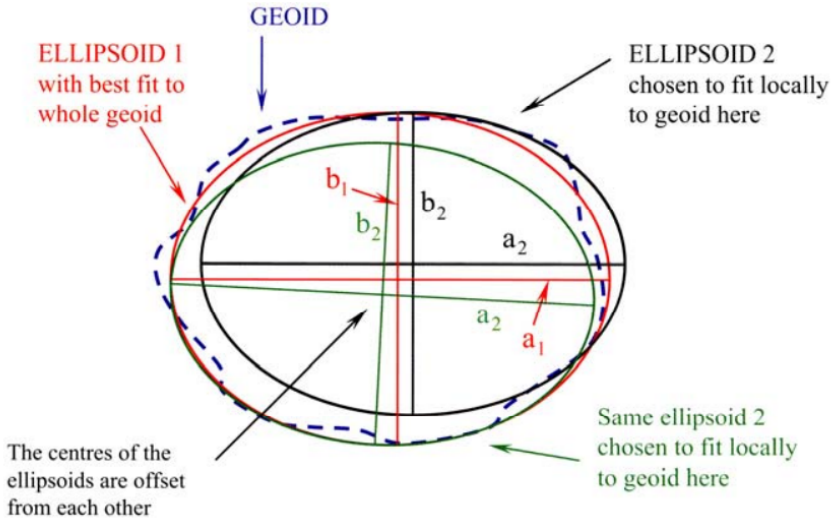


Figure 3.8: The two main reference surfaces: ellipsoid and geoid [18]

Geographic Coordinate Systems (GCS)

A Geographic Coordinate System (GCS) defines locations on the Earth's surface using a three-dimensional spherical surface. It is based on an ellipsoid, which approximates the shape of the geoid. The ellipsoid is mathematically defined, and its flattening factor f describes the compression of the symmetry axis relative to the equatorial radius. For Earth, the flattening factor is around $\frac{1}{300}$, leading to a difference of approximately 21 km between the major and minor axes. Locations on the ellipsoid are measured using latitude and longitude, which are angular measurements. Latitude measures angles in the North-South direction, with the Equator at 0°, while longitude measures angles in the East-West direction, with the Prime Meridian at 0°. An example of a GCS is the WGS84, commonly used for GPS.

Projected Coordinate Systems (PCS)

A Projected Coordinate System (PCS) transforms the Earth's curved surface into a flat, two-dimensional plane, essential for creating maps. This transformation is achieved through map projections, which convert geographic coordinates (latitude and longitude) into Cartesian coordinates (x, y). Different projections preserve different geometric properties. Conformal projections, such as the Mercator projection, preserve shapes and angles, making them suitable for navigational charts and topographic maps. Equal area projections, like the Mollweide projection, preserve the relative size of areas and are ideal for thematic mapping. Equidistant projections, such as the Azimuthal Equidistant projection maintain accurate distances from a point, which is useful for applications like air routes and radio propagation.

The accuracy of a map projection is never perfect due to the inherent distortions introduced when representing a curved surface on a flat plane. These distortions

can affect angular relationships, distances, and areas. Some map projections, like the Winkel Tripel and Robinson projections, provide a compromise that balances these distortions, making them suitable for general world maps. However, choosing the right projection is critical for specific tasks. For instance, using a conformal projection like Mercator for detailed area measurement would lead to significant errors. Therefore, the selection of a map projection should align with the map's intended purpose to ensure the accuracy and reliability of the spatial analysis. For example, Belgium uses the Lambert 2008 projection (seen on Figure 3.9), a conical projection that is secant and conformal, based on the GRS80 ellipsoid. This projection minimizes distortion for the region, providing accurate spatial representation for analysis and mapping.

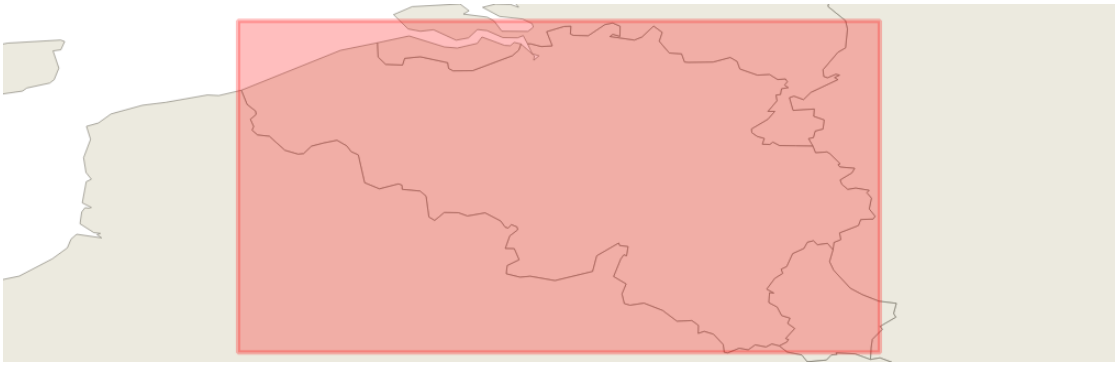


Figure 3.9: Belgium's projection lambert 2008 (EPSG:3812) [8]

3.1.5 Data Coordinate Systems

Concretely, we have the EPSG codes, developed by the European Petroleum Survey Group (EPSG), such as EPSG:4326 for the WGS 84 geographic coordinate system used by GPS, or EPSG:3857 for the Web Mercator projection commonly used in web mapping applications. These codes standardize the way spatial reference systems are referenced and used globally. When it comes to spatial and spatiotemporal databases, we have the equivalent SRID, which serves to link spatial data to its corresponding coordinate system within database environments like PostGIS and MobilityDB. SRIDs are the actual database implementations of their equivalent EPSG code (SRID 4326 - EPSG:4326), ensuring that spatial data stored in databases is accurately interpreted and manipulated, facilitating precise spatial operations and seamless integration across various platforms.

[8] *https://epsg.io/3812*

3.2 Advanced Mobility Data Management

3.2.1 MEOS: The Core Engine of MobilityDB

MEOS (Mobility Engine, Open Source) [26] is the core engine that powers the spatiotemporal database MobilityDB(see Section 2.1), designed for managing and querying mobility data. MEOS provides the fundamental functionalities for handling temporal and spatiotemporal data, facilitating advanced analyses and efficient data management. A key aspect of MEOS is its separation between data storage and processing. This decoupling allows for a more flexible system architecture, enabling efficient data handling and querying without being tightly coupled to the database storage mechanisms. MEOS supports various temporal data types and enables the creation, storage, and manipulation of trajectories, which are sequences of spatiotemporal points representing the paths of moving objects. Additionally, MEOS facilitates complex spatiotemporal queries, combining spatial and temporal conditions to extract meaningful insights from mobility data.

3.2.2 PyMEOS: Python Binding for MEOS

PyMEOS is the Python binding for MEOS, designed to integrate the advanced spatiotemporal functionalities of MEOS with the flexibility and ease of use of Python. This binding allows Python developers to interact directly with MEOS, facilitating the incorporation of mobility data processing and analysis into Python-based workflows.

Capabilities and Examples of PyMEOS

PyMEOS provides a range of features that allow users to perform various operations on mobility data. Here are some key capabilities and examples:

- **Trajectory Data Handling** : PyMEOS allows users to create, store, and manipulate trajectory data. Users can define trajectories with spatial and temporal components, enabling precise tracking of moving objects. Listing 3.1 shows an example of PyMEOS code in Python.
- **Spatiotemporal Queries** : Users can perform complex spatiotemporal queries to analyze mobility patterns, detect hotspots, and optimize routes.
- **Integration with Data Analysis Libraries** : PyMEOS can be integrated with popular Python data analysis libraries such as Pandas, NumPy, and MovingPandas, facilitating advanced data manipulation and visualization.
- **Visualization** : Users can visualize trajectories and query results using Python libraries like Matplotlib or Plotly, enabling better understanding and presentation of mobility data. An example is provided with the Code example 3.2, the data is from the Danish AIS dataset, covered in section 4.3.1, the result can be seen in Figure 3.10.

```python
from PyMEOS import PyMEOS_initialize, PyMEOS_finalize, TGeogPointInst,
    TGeogPointSeq

# Important: Always initialize MEOS library
PyMEOS_initialize()

sequence_from_string = TGeogPointSeq(
    string='[Point(10.0 10.0)@2019-09-01 00:00:00+01,
    Point(20.0 20.0)@2019-09-02 00:00:00+01,
    Point(10.0 10.0)@2019-09-03 00:00:00+01]')
print(f'Output: {sequence_from_string}')

sequence_from_points = TGeogPointSeq(instant_list=
[TGeogPointInst(string='Point(10.0 10.0)@2019-09-01 00:00:00+01'),
TGeogPointInst(string='Point(20.0 20.0)@2019-09-02 00:00:00+01'),
TGeogPointInst(string='Point(10.0 10.0)@2019-09-03 00:00:00+01')],
lower_inc=True, upper_inc=True)

speed = sequence_from_points.speed()
print(f'Speeds: {speed}')

# Call finish at the end of your code
PyMEOS_finalize()
```

Listing 3.1: Example of PyMEOS code

```python
from PyMEOS.db.psycopg import MobilityDB

PyMEOS_initialize()
connection = MobilityDB.connect(
    host=host, port=port, dbname=db, user=user, password=password
)
cursor = connection.cursor()

cursor.execute("SELECT * FROM public.PyMEOS_demo WHERE MMSI = 97000050;")
mmsi, trajectory, sog = cursor.fetchone()

fig, axes = plt.subplots(1, 2, figsize=(20, 10))
trajectory.plot(axes=axes[0])
sog.plot(axes=axes[1])
plt.suptitle(f"Ship {mmsi}")
plt.show()

connection.commit()
cursor.close()
PyMEOS_finalize()
```

Listing 3.2: Example code for plotting PyMEOS objects

[9] *https://pymeos.readthedocs.io/en/latest/src/examples/AIS.html*

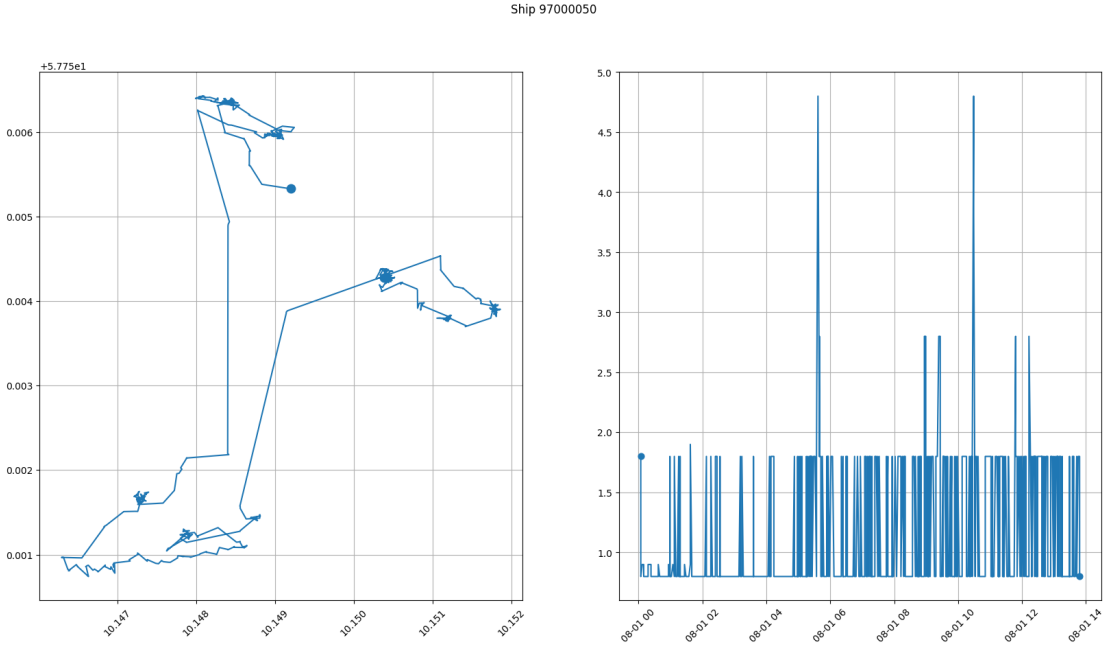


Figure 3.10: Illustration of PyMEOS data [9]

3.2.3 Accessing Temporal Values

A crucial operation in MEOS and MobilityDB is the ability to retrieve the value of a given temporal object for any timestamp within the valid time range of the object. This operation is done by the value_at_timestamp function, which utilizes various interpolation methods to provide the most accurate result. The Tgeompoint and Tgeogpoint types are a collection of location-time pairs that represent the trajectory of a moving object through space and time. When a value_at_timestamp call is made on a Tgeompoint or Tgeogpoint, MEOS examines all the timestamps(which are ordered in increasing direction by default) and applies a binary search to find the two pairs between which the target timestamp lies. It then applies the appropriate interpolation to determine the value at the specified time. By using a binary search, the time complexity is bounded by $O(\log n)$, while the space complexity remains $O(1)$.

Conclusion

PyMEOS serves as a powerful tool for Python developers working with mobility data. By providing an interface to MEOS and MobilityDB, it bridges the gap between advanced spatiotemporal data management and the extensive ecosystem of Python data analysis libraries. Extensive research has also been conducted to create bindings for other popular programming languages, such as JMEOS, the Java binding developed by Mareghni Nidhal [19], and ongoing projects like MEOS.NET, the .NET binding being developed by our colleague at ULB, Thomas Dudziak [9].

3.3 Vector Tiles Server

A Vector Tiles Server efficiently serves large amounts of vector data over the web. Unlike traditional raster tiles, which deliver pre-rendered images, vector tiles provide geographical data as vector graphics, allowing for dynamic styling and interaction within the client application. This approach reduces bandwidth usage and enhances rendering performance by only sending the necessary data for the current view. Vector tiles are typically encoded in formats like Mapbox Vector Tile (MVT) and can be integrated with mapping libraries such as Leaflet.

The XYZ format defines the resolution and coordinates of the tiles. The Z component specifies the zoom level, determining detail and resolution, while the X and Y components represent the tile's coordinates at that zoom level. Figure 3.11 illustrates the pyramid structure of vector tiles, demonstrating how zoom levels affect detail and coverage.

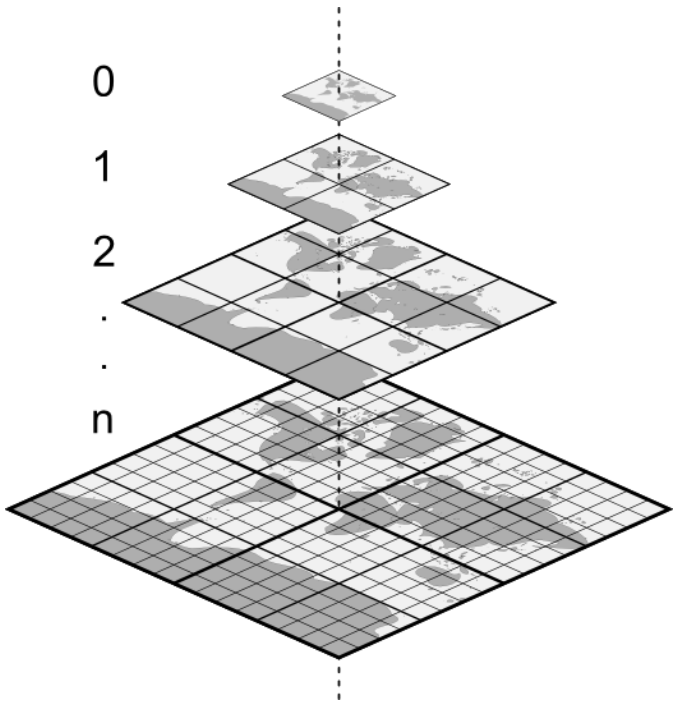


Figure 3.11: Illustration of the pyramid structure of vector tiles with zoom levels [10]

PgTileServ

Developed by Crunchy Data, `PG_tiles server`is an open-source tool designed to seamlessly integrate with PostgreSQL and PostGIS) [11]. It supports a wide range of geospatial functions and makes it easy to serve vector tiles via a simple and

[10] *https://docs.qgis.org/3.34/en/docs/user_manual/working_with_vector_tiles/vector_tiles.html*

[11] *https://access.crunchydata.com/documentation/pg_tileserv/1.0.8/introduction/*

efficient server setup. By utilizing SQL queries, the server can dynamically generate vector tiles on-the-fly, ensuring that only the necessary data for the current view is sent to the client. PgTileServ supports the XYZ tiling scheme and can serve tiles in the MVT format. This integration with PostgreSQL and PostGIS provides a robust and flexible solution for managing and serving spatial data. The dynamic generation of vector tiles enables real-time updates and interactions with the geographical data.

One of the key advantages of using PgTileServ is its ability to leverage the powerful spatial querying capabilities of PostGIS. This allows for complex spatial operations and filtering to be performed directly within the database, ensuring that the vector tiles generated are both accurate and efficient. Additionally, PgTileServ supports the use of SQL functions to customize the data served, providing a high degree of flexibility in how spatial data is presented. PgTileServ also offers features such as caching to improve performance and reduce the load on the database. By caching frequently requested tiles, the server can quickly respond to client requests without repeatedly querying the database. This can significantly enhance the performance of applications that rely on high-frequency tile requests.

3.4 PyQGIS

QGIS has a wide user base. With its open-source nature, it provides a Python API that allows anyone to build their unique plugin for their particular use case. This Python API gives powerful tools and access to QGIS's features. In our goal of creating a temporal-spatial visualization, we will cover specific features provided by PyQGIS. We invite curious readers to have a look at the well-writtent PyQGIS plugin cookbook [12].

3.4.1 Temporal Controller

QGIS introduced a built-in Temporal Controller in its 3.14 release [24], this feature was previously implemented as a separate plugin. Our aim is to develop a plugin that work hand in hand with this feature of QGIS to provide user with the possibility of viewing the spatiotemporal objects from MobilityDB.

QgsTemporalController

The QgsTemporalController [13] class creates the base structure of the temporal controller, and very importantly, implements the updateTemporalRange signal, it is emitted when a user interacts with the temporal controller GUI.

[12] *https://docs.qgis.org/3.34/en/docs/PyQGIS_developer_cookbook/index.html*

[13] *https://qgis.org/PyQGIS/3.34/core/QgsTemporalController.html#qgis.core.QgsTemporalController*

QgsTemporalNavigationObject

The QgsTemporalNavigationObject class is the actual implementation of a QgsTemporalController based on a frame by frame navigation and animation. This class gives QGIS users three different navigation methods:

- NavigationMode.Animated 3.12: Users can view an animation frame by frame of the temporal data, set the time ranges, the duration between each frame(with the time granularity), the frame rate of the animation, buttons to play, move by 1 frame or move to the end on either lower upper bounds.
- NavigationMode.FixedRange : The temporal data contained between the selected date ranges.
- NavigationMode.Movie : Animation for a given numer of frames, this mode does not take into account temporal properties but enables custome geometries modifications through the animation by using `@frame_number` expressions

It also provides a "Cumulative Range" button, when selected, users are able to view the temporal data from the beginning up to the current frame.

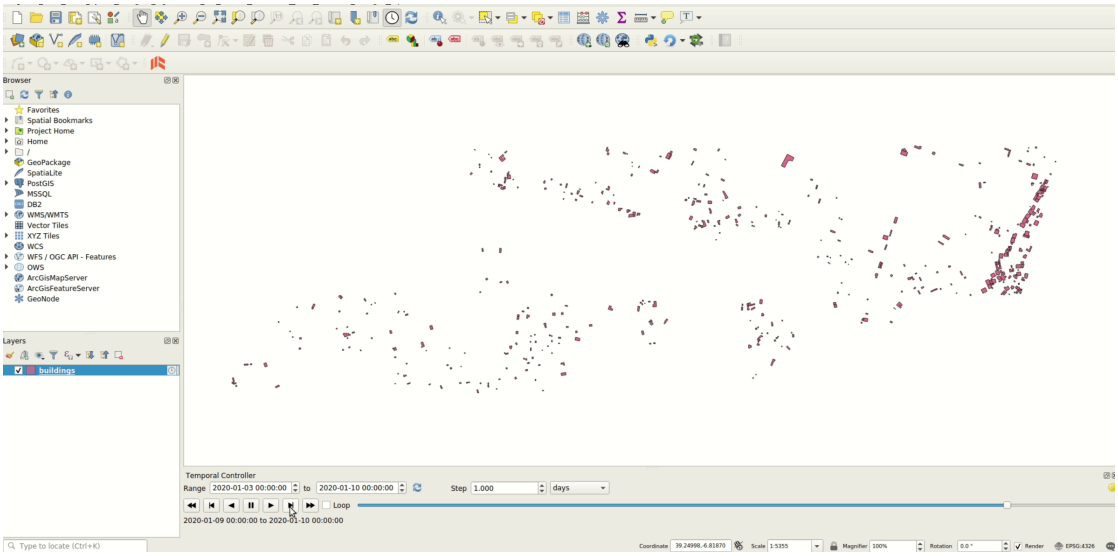


Figure 3.12: QGIS interface of the Temporal Controller in the Animated mode [24]

The Temporal Controller, just like most of PyQGIS's UI elements, provide built-in signal functions that can be connected to custom function in Python. For example we mentionned `updateTemporalRange`, this signal is emmited whenever the user interacts with the Tempor Controller UI Or when the state of the Temporal Controller changes. These signals are very useful and can be handled directly within the Python script.

3.4.2 Vector Layer

The QgsVectorLayer [14] class forms the core structure for handling vector data in QGIS. This class is crucial for creating, modifying, and visualizing vector data, which includes points, lines, and polygons. It encapsulates the data and provides a comprehensive set of methods for managing attributes, geometries, and spatial indices. The QgsVectorLayer class supports various data sources and formats such as shapefiles, GeoJSON, and memory-based layers. Since QGIS does not support MobilityDB data types, to visualize spatiotemporal types using this method requires a transformation into regular PostGIS spatial geometries, like it is done in the materialized views of Move. In our research project, we will be focusing on the memory-based Vector layer as it gives us full liberty in how we design our solutions inside Python using PyMEOS to handle MobilityDB data types. The PyQGIS developer cookbook [15], contains a well-written chapter on Vector Layers introducing various aspects of this class.

Creating a New Memory-based Vector Layer

The memory data provider, which is designed for in-memory data handling, does not have inherent persistent storage, making it ideal for temporary data layers used primarily by plugin or third-party application developers. To create a new memory-based vector layer, we need to define the data source URI, which specifies the geometry type (e.g., "point", "linestring", "polygon") and optionally includes parameters for the coordinate reference system (CRS), spatial index, and layer attributes. Listing 3.3 shows an example of how this is done in Python. Using memory-based vector layers provides flexibility and performance advantages for handling temporary spatial data, which is ideal for our scenario as we update geometries at every new frame of the temporal controller.

Attributes and Features

The QgsVectorLayer class in QGIS provides robust capabilities for retrieving and managing attributes of vector data. It allows users to access field definitions, retrieve attribute values for specific features, and modify the schema by adding or removing fields. Additionally, attributes can be updated programmatically, facilitating dynamic and flexible data manipulation within the vector layer. The QgsVectorLayer class also supports iterating over features, enabling users to traverse and process each feature within the layer efficiently. This is particularly useful for tasks such as spatial analysis, data validation, and attribute updates. By providing an iterator, QGIS allows developers to loop through the features, access their geometries and attributes, and perform necessary operations. This capability is integral for automating workflows and conducting comprehensive data assessments within the QGIS environment.

[14] *https://qgis.org/PyQGIS/master/core/QgsVectorLayer.html*
[15] *https://docs.qgis.org/3.34/en/docs/PyQGIS_developer_cookbook/tasks.html*

```
vlayer = QgsVectorLayer(f"Point?crs=epsg:{srid}", "MobilityBD Data", "memory")
# Define the fields
fields = [
    QgsField("id", QVariant.String),
    QgsField("start_time", QVariant.DateTime),
    QgsField("end_time", QVariant.DateTime)
]
vlayer.dataProvider().addAttributes(fields)
vlayer.updateFields()
# Define the temporal properties
tp = vlayer.temporalProperties()
tp.setIsActive(True)
tp.setMode(
QgsVectorLayerTemporalProperties.ModeFeatureDateTimeStartAndEndFromFields
)
tp.setStartField("start_time")
tp.setEndField("end_time")

vlayer.updateFields()
QgsProject.instance().addMapLayer(vlayer)
```

Listing 3.3: Code for creating a memory-based vector layer

QGIS Feature

A key component in handling features within QgsVectorLayer is the QgsFeature class. The QgsFeature class represents a single feature within a vector layer, encapsulating its geometry and attributes. It provides methods for accessing and modifying both the spatial and non-spatial properties of the feature. This allows for detailed manipulation and analysis of individual features, making it an essential part of the QGIS data handling framework. Using QgsFeature, developers can create new features, set their geometries, and assign attribute values. This class enhances the flexibility and control developers have over the vector data, ensuring that features can be dynamically created, updated, and managed within the QGIS environment.

Spatial Index

A direct quote from the cookbook: "A layer without a spatial index is a telephone book in which telephone numbers are not ordered or indexed. The only way to find the telephone number of a given person is to read from the beginning until you find it.". For this reason, Spatial indexes [2] [21] are very important for the performance of spatial operations; they are not created by default for a QGIS vector layer but it is very easy to create one as shown in Listing 3.4. Creating a spatial index improves the performance of spatial queries and operations by allowing quick retrieval of features based on their spatial location. This is particularly useful for large datasets where spatial searches would otherwise be slow.

QGIS provides two spatial indexes be default for vector layers :

```
index = QgsSpatialIndex(layer.getFeatures())
```

Listing 3.4: Code for creating a new spatial index for a vector layer

1. QgsSpatialIndex [16]: This class provides an R-tree-based spatial index, which is a tree data structure used for storing spatial objects. The R-tree allows efficient querying of spatial data, such as finding all features within a certain area or finding the nearest feature to a point.

2. QgsSpatialIndexKDBush [17]: This is an alternative spatial index implementation based on the KDBush algorithm, which is a fast static spatial index for 2D points. The KDBush index is particularly efficient for nearest neighbor searches and intersection queries [18].

In general, using spatial indexes can greatly enhance the performance of spatial operations in QGIS by reducing the number of features that need to be examined for spatial queries. This is crucial for applications that require real-time spatial analysis or handle large amounts of geospatial data.

Symbols and Renderers

The QgsVectorLayer class in QGIS provides extensive support for customizing the visualization of vector data through symbols and renderers. Symbols define the visual appearance of features, such as points, lines, and polygons, while renderers determine how these symbols are applied based on the feature attributes or other criteria. This flexibility allows users to create detailed and meaningful maps that effectively communicate spatial information. With various types of renderers, such as single symbol, categorized, graduated, and rule-based renderers, QGIS enables sophisticated and dynamic cartographic representations.

3.4.3 Vector Tiles Layer

The QgsVectorTileLayer class in QGIS represents a specialized type of layer designed for displaying vector tiles. As discussed in Section 3.4.2, vector tiles deliver geographic data in small chunks, which can be styled and rendered on the client side. This method offers advantages in flexibility, performance, and dynamic styling without the need to re-fetch tiles from the server. The QgsVectorTileLayer supports various vector tile formats, including Mapbox Vector Tiles (MVT), and can handle sources like local files, web services, or databases. Key properties include defining tile resolution, cache settings, and coordinate reference systems

[16] *https://qgis.org/PyQGIS/3.34/core/QgsSpatialIndex.html#qgis.core.QgsSpatialIndex*

[17] *https://qgis.org/PyQGIS/3.34/core/QgsSpatialIndexKDBush.html#qgis.core.QgsSpatialIndexKDBush.intersects*

[18] *https://github.com/mourner/kdbush*

(CRS). Users can configure the layer by specifying a URL template for tile requests and styling it using QGIS style files for consistent and customizable map rendering.

A notable feature of QgsVectorTileLayer is its performance optimization. Vector tiles are bandwidth and storage-efficient, as they transmit vector data instead of images. This efficiency benefits large-scale maps and interactive applications, ensuring responsiveness and smooth user experiences. The layer integrates seamlessly with QGIS's symbology and rendering engine, supporting advanced styling capabilities like categorized, graduated, and rule-based renderers. Additionally, QgsVectorTileLayer supports querying and interaction, allowing users to perform feature identification and selection for detailed exploration and analysis. The class also offers tile caching methods to improve performance by storing fetched tiles locally, reducing server load and load times.

3.4.4 Background Tasks

A QGIS Task [19] is a framework designed to manage and execute long-running operations without freezing the user interface. This is crucial for maintaining a responsive application, especially when dealing with complex spatial analysis or large data processing tasks. There exists multiples ways a A QGIS Task can be created, with the main method being by subclassing the QgsTask class and overriding its run() method, where the task's logic is implemented. Custom signals can be defined to communicate the task's progression or completion status. In the Listing 3.5, MyTask simulates a long-running process by sleeping for a specified duration. The task is then added to the QGIS application's task manager, which handles its execution.

```
from qgis.core import QgsTask, QgsApplication

class MyTask(QgsTask):
    def __init__(self, description, duration):
        super(MyTask).__init__(description, QgsTask.CanCancel)
        self.duration = duration

    def run(self):
        QgsApplication.processEvents()
        for i in range(self.duration):
            # Simulate a long-running task
            QThread.sleep(1)
            self.setProgress(100 * i / self.duration)
        return True

task = MyTask("My Custom Task", 10)
QgsApplication.taskManager().addTask(task)
```

Listing 3.5: Code for creating a basic QGIS Task

[19] *https://docs.qgis.org/3.34/en/docs/PyQGIS_developer_cookbook/tasks.html*

QGIS Task Limitations

Since PyQGIS is based on PyQt5, creating QGIS GUI objects inside a QGIS task thread is prohibited because PyQt5 does not support creating GUI objects in non-main threads. This means that the initialization process of our solution, such as creating layers and features, cannot be offloaded to a QGIS Task. Another major limitation is the inherent design of multithreading in Python, which we discuss in detail in Chapter 7. The Global Interpreter Lock (GIL) in Python means that the QGIS task does not run in true parallelism but concurrently with the main QGIS thread. This allows long-running tasks to run without blocking the UI. However, performing heavy computation tasks inside the QGIS Task thread can impact the performance of the main UI thread, leading to slowdowns. In the case of a temporal controller animation, this can result in frame rate instability.

3.4.5 Optimizing PyQGIS Code

We list below some useful optimization aspects related to developing an animation with PyQGIS and Python. As these concerns are purely related to the technical development of the solutions presented in the next chapters, we list them in this section.

1. **STDOUT print calls**: These have a significant time cost. We recommend using the QGIS logging system 3.6 if there is a need to print a message during the animation run. We used the logs to gather information regarding FPS, variable states, etc.

```
1  def log(msg):
2      QgsMessageLog.logMessage(msg, 'LOG_NAME', level=Qgis.Info)
```

Listing 3.6: Logging with QGIS's built-in logging system

2. **Updating geometries at each frame**: We compared the performance of deleting all Qgis Features and only creating QgisFeatures for points to show in the new frame, versus keeping the same Qgis Features for all tgeompoints in the dataset throughout the animation. Our results showed a noticeable advantage for the latter approach.

3. **Using the Layer's data provider**: This helps to avoid unnecessary errors related to the visual PyQT components. For example, using addFeatures() from the Vector layer's data provider is more efficient and does not result in warnings. compared to the layer's addFeatures() method.

3.5 Summary of the Technologies

This chapter provides readers with the essential knowledge required to understand the tools and technologies underpinning the solutions presented in subsequent

chapters. It begins with a comprehensive introduction to Geographical Information Systems (GIS), explaining how real-world data is transformed into spatial information that computers can process. Key concepts such as the vector representation of spatial data and EPSG codes, which represent the Earth's surface on 2D maps, are thoroughly defined.

Following this, we describe MEOS, the engine behind MobilityDB, which facilitates the management of spatiotemporal data directly in C++ without the need for a MobilityDB database interface. QGIS offers developers a powerful Python library, PyQGIS, which enables the development of custom plugins. This chapter covers the essential elements of PyQGIS used in our visual solution, including vector layers, the Temporal Controller, and vector tiles layers. Additionally, we discuss PyMEOS, the Python binding for MEOS, which plays a crucial role in building our final solution. Finally, we introduce the concept of a Vector Tiles server using pg_tileserv, a PostGIS-based vector tiles server. Having covered these core components, the next chapter will detail the computer hardware configurations and various datasets utilized to test our implementations.

Chapter 4

Setup Overview

In the introduction chapter, we outlined our objectives concerning performance. Our goal is to develop a solution that operates seamlessly across a variety of hardware configurations. To achieve this, we need to carefully define the metrics used to compare different approaches, ensuring fair assessments across architectures.

4.1 Performance Metrics

Developing robust performance metrics for a visualization method is essential yet challenging. We will utilize a combination of quantitative and qualitative metrics to comprehensively evaluate our solution:

1. **Frame Rate**: This metric measures the smoothness of the animation during interactive visualization. A higher frame rate indicates a smoother and more responsive visual experience, which is crucial for real-time applications. This is measure in Frames Per Second(FPS), maintaining a consistent FPS is vital to ensure that users can interact with the visualization without latency or lag.

2. **Memory Usage**: This metric monitors the amount of RAM consumed during the visualization process, measured in megabytes (MB). Efficient memory usage is critical for running visualizations on devices with varying resources. High memory efficiency ensures that the application remains responsive and stable, even on hardware with limited memory capacity.

3. **Accuracy**: This metric assesses the correctness of the visualization in representing the underlying data. Ensuring high accuracy is vital for maintaining the integrity of the data being visualized. Accurate visual representations allow users to derive valid and reliable insights, which is particularly important in data-driven decision-making processes.

4. **Scalability**: This metric evaluates the system's performance as the size of the data and the duration of the animation increase. A scalable system can handle growing datasets and longer animations without significant performance degradation. This ensures that the visualization method remains effective and efficient, regardless of the scale of the data being processed.

5. **Responsiveness**: This metric measures the system's ability to quickly respond to user interactions, such as zooming, panning, or adjusting visualiza-

tion parameters. High responsiveness enhances user experience by providing immediate feedback, making the visualization more intuitive and engaging.

4.2 Hardware Configurations

To ensure our animation runs efficiently across various hardware setups, we tested all our solutions on a broad spectrum of configurations. This approach allows us to assess performance across different computational environments, providing insights into both the upper and lower bounds of performance. We focus on two primary configurations: a laptop with modest computational capabilities and a high-end desktop.

The specifications for these configurations are summarized in Table 4.1.

Component	**Laptop Configuration**	**Desktop Configuration**
CPU	Intel Core i7-6700HQ 2.60GHz	Ryzen 7800X3D 4.20GHz
RAM	16GB DDR4 2133 MHz	32GB DDR5 5600 MHz
GPU	Intel HD Graphics	RTX 3090 24GB VRAM
OS	Ubuntu 22.04	Ubuntu 22.04

Table 4.1: Specifications of the hardware configurations

Laptop Configuration

The laptop configuration represents a typical consumer-grade setup with modest computational capabilities. It is equipped with an Intel Core i7-6700HQ processor, 16GB of 2133 MHz DDR4 RAM, and integrated Intel HD Graphics. This configuration is suitable for assessing the performance of our solution on hardware commonly found in everyday use, providing a baseline for minimum performance expectations.

Desktop Configuration

The desktop configuration represents a high-end setup with significant computational power, as of August 2024. It features a top of the line Ryzen 7800X3D processor, 32GB of fast 6000 MHz DDR5 RAM, and an RTX 3090 GPU with 24GB of VRAM. This configuration is intended to test the upper limits of our solution's performance, ensuring that it can take full advantage of advanced hardware for optimal performance.

Rationale for Hardware Selection

Testing on these two configurations allows us to evaluate the scalability and efficiency of our solution across different hardware environments. By comparing

performance metrics such as frame rate, memory usage, and responsiveness, we can identify potential bottlenecks and optimize our solution for a wide range of users. This dual-approach testing ensures that our solution is robust, versatile, and capable of delivering consistent performance across varying levels of computational resources.

4.3 Datasets

To stress-test our solutions, we selected two distinct datasets that will help us measure performance accurately. These datasets provide a comprehensive evaluation of our methods across different types of spatiotemporal data.

4.3.1 Danish AIS Dataset

The Danish Maritime Authority provides extensive AIS (Automatic Identification System) data, essential for monitoring ship traffic within Danish waters [1]. This data offers a real-time snapshot of maritime traffic, valuable for maritime authorities, shipping companies, and researchers. AIS allows ships to broadcast their identity, position, course, and speed, enhancing maritime safety and security by providing real-time data on vessel movements.

Users can access continuously updated historical AIS data for free. The data is stored in CSV files and can be downloaded and unpacked from zip files. This dataset is the primary one used in our research project due to its scalability, extensive coverage, and the long trajectories of most boats, providing a robust test for our solutions.

Statistics

This dataset has the following characteristics:

1. Total number of individual trajectories: 5821
2. Total size of the temporal point table: 193 MB
3. Time range for the entire dataset: 2023-06-01 00:00:00 to 2023-06-01 23:59:59

Static View of the Data

The initial data contained in the downloaded file requires some data preprocessing. We direct readers to MobilityDB's workshop [25] for detailed explanations on this part as it is not the focus of the thesis. Figure 4.1 shows the the large number of points representing all the GPS positions in the dataset. Figure 4.2 shows the static view of all the ship's trajectories.

[1] *https://www.dma.dk/safety-at-sea/navigational-information/ais-data*

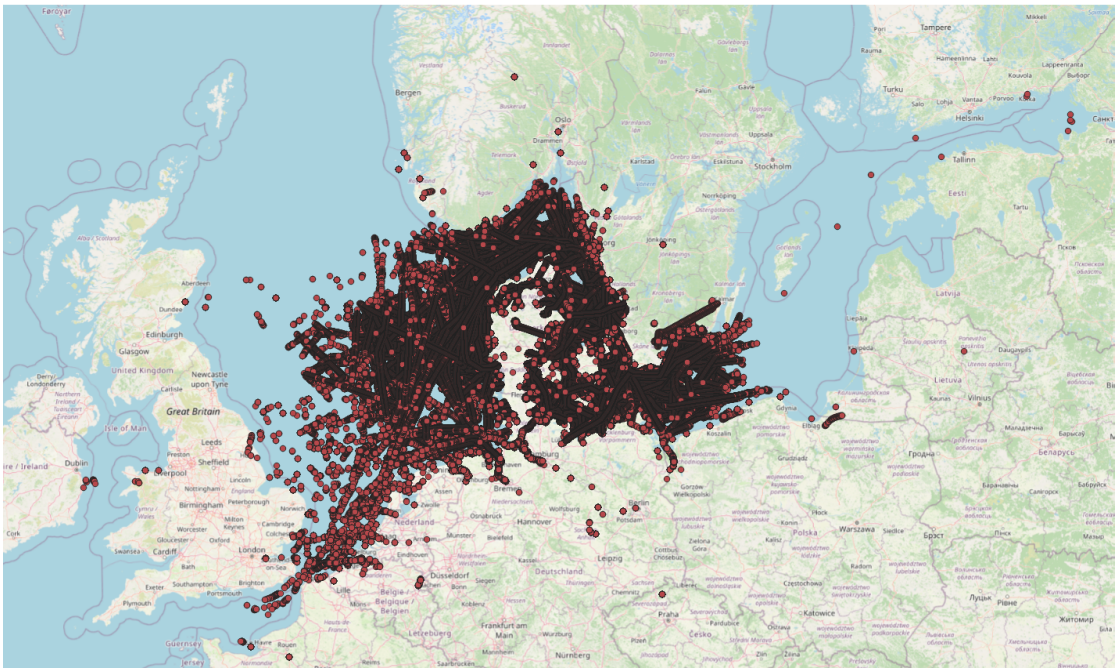


Figure 4.1: Danish AIS dataset : all the GPS positions

Figure 4.2: Danish AIS dataset : trajectories of ships

4.3.2 STIB Dataset

The General Transit Feed Specification (GTFS) establishes a standardized format for public transportation schedules and their associated geographic information, with GTFS-realtime used for real-time transit data. Many transit agencies worldwide publish their data in these formats, making it publicly accessible.

The STIB-MIVB (Brussels Intercommunal Transport Company) operates 4 metro lines, 17 tram lines, and 55 bus lines (along with 11 "Noctis" bus lines) in Belgium's capital [23]. For our research, we will use this GTFS data from STIB to stress-test our solutions with a extremely large dataset containing numerous individual trips for visualization. Figure 4.3 shows the routes and stops for the GTFS data.

Statistics

This dataset has the following characteristics:

1. Total number of individual trajectories: 497,000
2. Total size of the temporal points table: 6000 MB
3. Time range for the entire dataset: 2024-05-23 00:00:00 to 2024-05-23 23:59:59

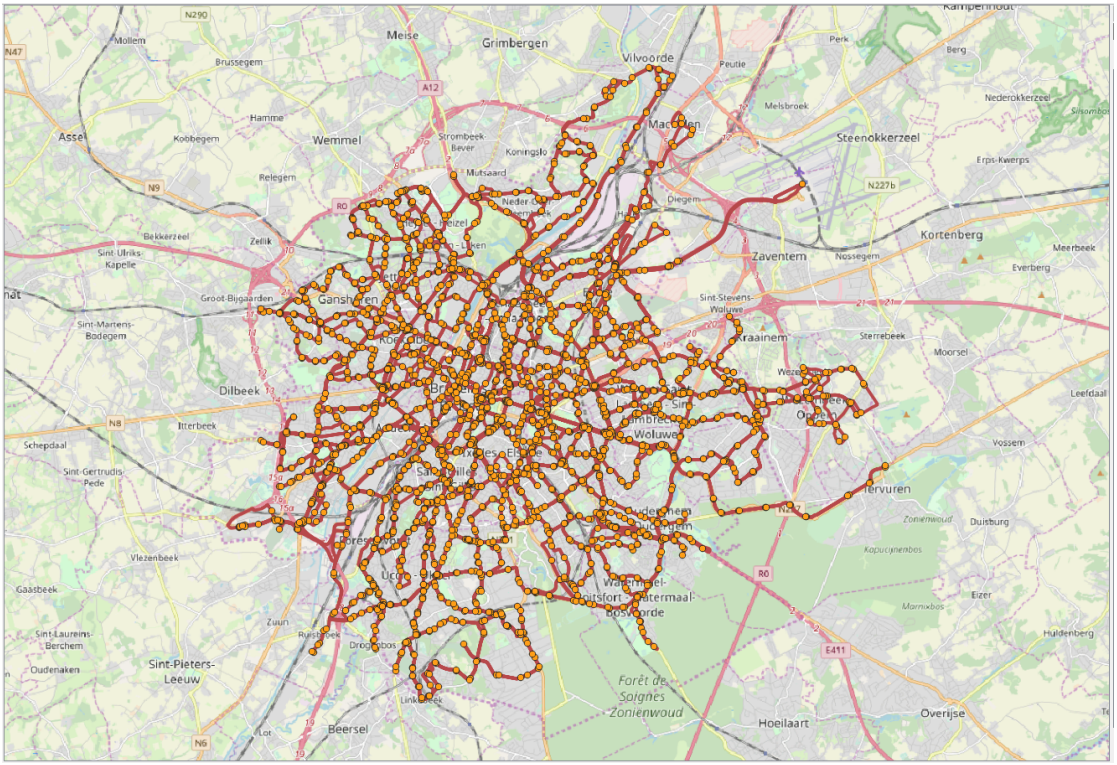


Figure 4.3: STIB dataset : the routes and stops for the GTFS data from Brussels [2]

Reading and loading GTFS data into a MobilityDB database is enabled thanks to the brilliant work of different contributors to this open-source project. Research projects like "Mobility Data Exchange Standards in MobilityDB" by Iliass El Achouchi [1] have significantly advanced this field.

[2] *https://docs.mobilitydb.com/MobilityDB-workshop/master/ch04.html*

4.4 Summary of the Setup Overview

This chapter details the hardware configurations used to test our design implementations, ensuring optimal performance for a wide range of users. We present the different metrics measured: Frame rate, memory usage, accuracy, scalability, and responsiveness. By utilizing two diverse datasets—one representing boat movements and the other public transport data, we comprehensively evaluate the robustness, efficiency, and accuracy of our solutions across different types of spatiotemporal data. The Danish AIS dataset tests our methods against large-scale, continuous data, while the STIB dataset challenges our solutions with a high density of trajectories. This approach ensures that our methods are versatile and effective in various real-world scenarios. With these important details established, we will describe our initial solutions in the next chapter, utilizing the tools from Chapter 3. These solutions will be tested using the hardware configurations and datasets presented in this chapter.

Chapter 5

Vector Layer

This chapter covers two distinct solutions using the in-memory vector layer of QGIS, providing exact data accuracy. The research presented here is built on top of previous experiments conducted by Ludéric Van Calck (see Section 2.3) [6] and Maxime Schoemans's MOVE plugin (Section 2.3) [20]. We utilize the power of PyMEOS combined with PyQGIS to extend MOVE to visualize and animate large quantities of spatiotemporal data inside QGIS.

5.1 Interactive Mode

PyMEOS provides the capability to handle MobilityDB objects directly in Python [26], particularly spatiotemporal types such as TgeomPoint and TgeogPoint, which represent trajectories of moving objects. This functionality allows us to develop a solution that eliminates the need for constant calls to a MobilityDB database whenever we interact with MobilityDB's custom objects and methods. Building on Ludéric Van Calck's research(detailed in Section 2.3), our base solution utilizes the following features provided by PyQGIS:

1. **In-Memory Vector Layer (QgsVectorLayer)**: This layer displays the geometries throughout the animation (detailed in Section 3.4.2).
2. **QGIS's Temporal Controller (QgsTemporalController)**: The temporal controller enables users to visualize temporal data by selecting the date, time, and granularity, and also provides the ability to animate the temporal data (detailed in Section 3.4.1). The QgsTemporalController class emits the updateTemporalRange signal whenever the user interacts with the temporal controller or when the animation is playing. Following Ludéric's approach, we use this signal to create a custom function, on_new_frame. This function, connected to the updateTemporalRange signal, updates the geometries visible on the vector layer on every new frame of the animation. A pseudocode is shown in Listing 5.1.

5.1.1 Design Architecture

The first solution is straightforward and consists of three main steps, as shown in Figure 5.1, we detail the three steps as follows :

1. **Fetching Trajectories**: First, all the trajectories are fetched from the MobilityDB database.

```
def OnNewFrame(self):
    current_frame = TemporalController.current_frame()
    new_geometries = {}
    for feature_id, trajectory in self.list_of_trajectories:
        new_geometries[feature_id] =
            trajectory.value_at_timestamp(Timestamps[current_frame])
    self.update_vector_layer_geometries(new_geometries)
```

Listing 5.1: Function called every new frame

2. **Initializing Vector Layer and Features**: Next, we create the Vector Layer and initialize a unique QGIS Feature for each trajectory. The temporal properties are activated alongside the Temporal Controller in the Animated mode.

3. **Animation and Visualization**: Finally, during the animation and visualization phase, the on_new_frame function updates the geometries of all QGIS Features to their new positions on each new frame signal emitted. These positions are obtained by calling the value_at_timestamp function (see Section 3.2.3) for the datetime associated with the new frame. When value_at_timestamp is called on a trajectory that doesn't exist for the given datetime, it will throw an error. In our implementation we put in place a try/catch block that continues forward when this happens.

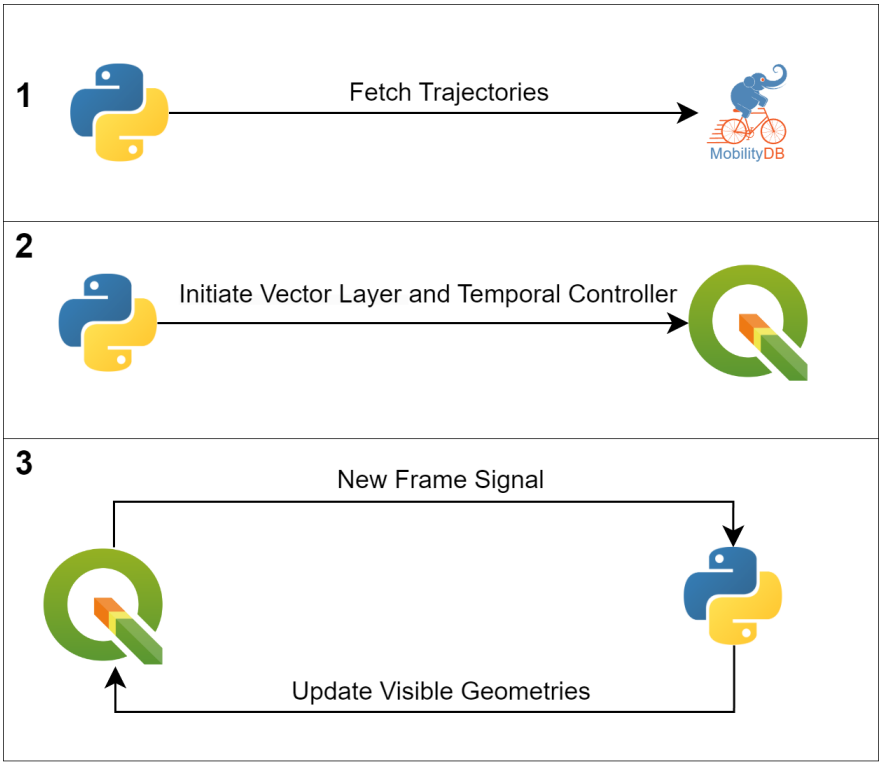


Figure 5.1: Design architecture the Interactive Mode

5.1.2 Key Performance Metrics

We will use the Danish AIS dataset from Section 4.3.1 and divide it into three subsets, containing 10%, 50%, and 100% of the entire dataset. The animation will run for the first 120 frames with a one-minute time granularity, displaying the first two hours of the dataset. It is important to note that the actual number of points shown on screen is different from the total number of boats considered (see Table 5.1). Some boats may only appear later after the initial 120 frames. We use the running average formula from Equation 5.1 to track the average number of objects shown on screen during the animation.

$$\textbf{running average}_{n+1} = \text{running average}_n + \frac{(x_{n+1} - \text{running average}_n)}{n+1} \quad (5.1)$$

Percentage of the Danish AIS dataset	Total Boats	Average Boats per Frame	Size in MB
10%	582	255.025	19.3
50%	2910	1549.299	96.5
100%	5821	3138.208	193

Table 5.1: Statistics for the danish AIS dataset subsets

Time to Fetch All the Trajectories

We can measure the time it takes to fetch the trajectories from the MobilityDB database for various combinations of dataset sizes and hardware configurations. Table 5.2 displays the results for both the lower-end laptop configuration and the higher-end desktop configuration. From these results, it is evident that the hardware configuration significantly impacts the initial load time.

Hardware Configuration	Laptop	Desktop
10%	16.270 s	4.495 s
50%	36.004 s	8.454 s
100%	74.700 s	16.236 s

Table 5.2: Interactive Mode results : time to fetch trajectories from MobilityDB, in seconds

Frame Rate

During the animation, we can continuously measure the frame rate by calculating the frames per second (FPS) at each frame using Equation 5.1.2. The ω parameter

in the formula represents the time to generate a new frame. This parameter is essentially the duration required to execute one instance of the on_new_frame function. The time to emit and catch the change of frame signal and the function call itself is negligible in our Python script. We use this value at each frame to calculate the average FPS over the test animation of 120 frames. The results can be seen in Table 5.3. As the dataset size increases, the average FPS takes a dramatic hit on both hardware configurations.

$$\text{FPS} = \frac{1}{\omega}$$

Percentage of the Danish AIS dataset	**Laptop**	**Desktop**
10%	20.938	85.776
50%	3.819	24.883
100%	1.954	13.136

Table 5.3: Interactive Mode results : average FPS over the initial 120 frames

To get a more in-depth analysis of the FPS value throughout the animation, we will us the following test bench : the laptop configuration with 10% of AIS boats(582 boats). Figure 5.2 displays in blue the computation time cost of the on_new_frame function calls (and by extension the value for the ω parameter in the FPS formula). The figure also displays the two main computational components inside on_new_frame: in orange the time to compute the new positions of all trajectories using PyMEOS's value_at_timestamp method, and in green the time to update the existing QGIS Features's geometries with the newly computed positions.

From these results, we can observe that the main computation time cost inside of the on_new_frame function can be attributed to the multiple value_at_timestamp calls that are made to compute the positions of each trajectory. We also provide the corresponding FPS per frame plot in Figure 5.3, this also shows that the frame rate of the animation fluctuates depending on the number of boats appearing on screen.

Memory Usage

We store the entirety of the trajectories in memory from the start. This results in a solution where the memory usage does not fluctuate very much, if at all. The trajectories are stored in memory using PyMEOS. In our code, as long as we do not instantiate certain aspects of a MobilityDB trajectory in Python (such as storing the instants as sets of Shapely Points and DateTime), the memory usage remains constant and low. Table 5.4 shows the approximate memory usage by the QGIS process during the animation of two subsets of the Danish AIS and STIB datasets. These measurements are taken with only the vector layer enabled, without any other layer that could impact this measure(such as the OpenStreeMap world map layer), the animations are run on the laptop hardware configuration.

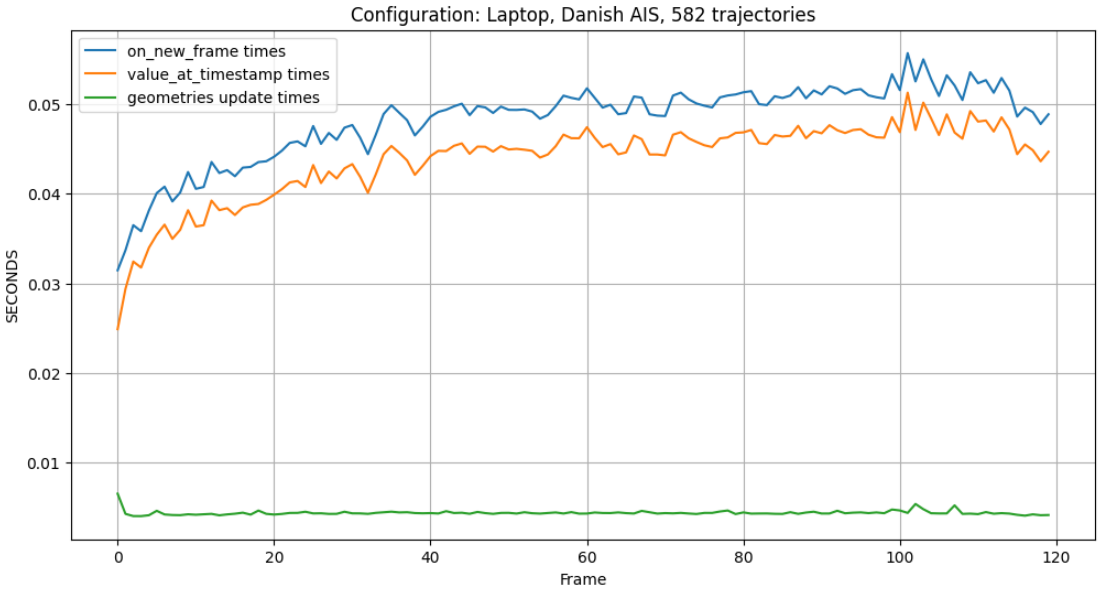


Figure 5.2: Interactive Mode results : frame generation time breakdown

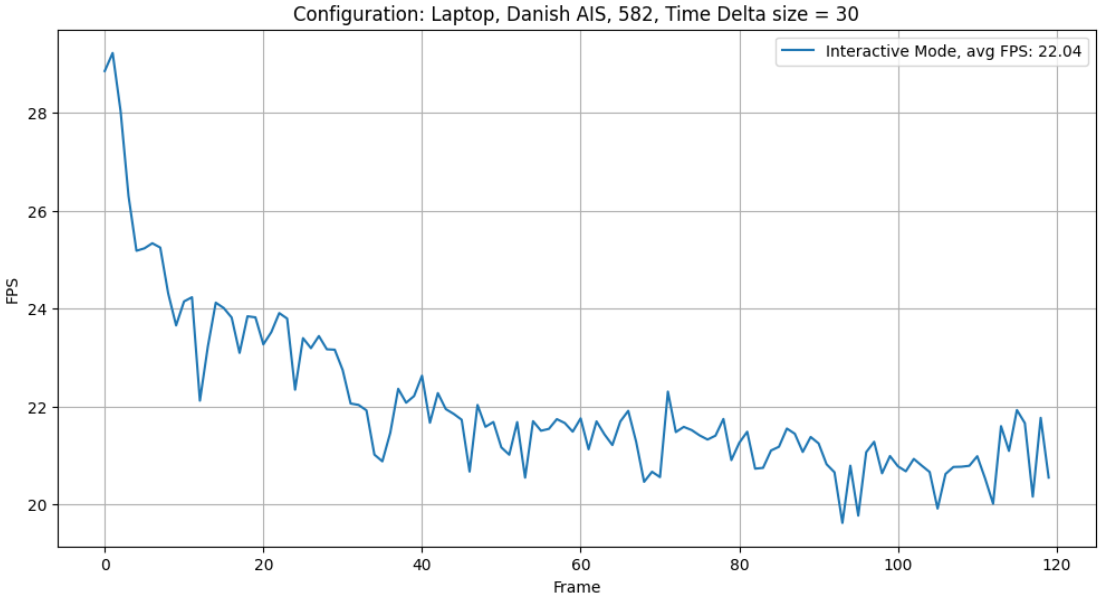


Figure 5.3: Interactive Mode results : frame rate stability

Dataset	**Interactive Mode**
Danish AIS : 582 trajectories	0.59
Danish AIS : 5821 trajectories	1.80
STIB : 100000 trajectories	8.00
STIB : 497609 trajectories	Not enough RAM

Table 5.4: Interactive Mode results : memory usage (in GB)

User Experience

A user can manually skip the timeline in the Temporal Controller. For example, if a user wants to go from frame 1 to 150, the sequence of frame change signals that PyQGIS will emit might look like this: [Frame 1, Frame 2, Frame 3, Frame 4, Frame 20, Frame 120, Frame 149, Frame 150]. Intermediate frame calls are unnecessary computations that slow down or even block the QGIS process. To counter this, our script only computes directly adjacent frames, avoiding unnecessary computations.

As we load the entirety of the trajectories in memory, the user can freely update the state of the Temporal Controller during the animation without requiring any additional load times. This means changing the frame duration, temporal granularity, temporal extents, or even navigation mode without any lag or delay. However, since we use the in-memory Vector Layer, all QGIS Features with their attributes and geometries are lost upon closing the QGIS app. This means that on every QGIS startup, the user will have to wait for all the trajectories to be loaded back into memory.

5.1.3 Conclusion

This first solution provides a concrete implementation of the idea proposed by Ludéric Van Calck's on-the-fly interpolation, using PyMEOS to perform MobilityDB specific operations without needing to make any database requests. By storing the entire trajectories during the animation, the user can modify the Temporal Controller's configuration without requiring any load time. This allows for a highly interactive visualization(hence the name *Interactive Mode*), in line with QGIS's Temporal Controller's philosophy. Users can switch the time granularity, temporal extents, and frame durations while maintaining a smooth running animation.

However, this solution struggles significantly in two areas when scaling up the dataset size. First, since we store the entire trajectories in memory, the actual cost of keeping the PyMEOS objects in memory is two to three times larger than the actual size of the trajectories in the database. For very large datasets, such as the entire STIB dataset (see Section 4.3.2), where even the 32GB of RAM in our high end desktop test machine weren't sufficient to display it. Secondly, the animation frame rate, measured in FPS is very poor for larger datasets. The low FPS values are directly correlated with the large computation time of the various calls to `value_at_timestamp` in `on_new_frame`. Although its time complexity is $O(\log n)$, which is efficient, as this operation is repeated for each trajectory, it creates a significant delay. Our main motivation behind the next iterations is to reduce or eliminate this time sink.

5.2 TimeDeltas Mode

In the previous solution, we observed that value_at_timestamp is a significant time sink within the on_new_frame function. In this section, we present a solution which decreases the time cost of each `value_at_timestamp` by reducing the size of the trajectories.

Instead of considering the entire time range all at once, we divide it into subsets of equal sizes, which we will call Time Deltas. To illustrate this idea, consider the trajectory of an object through time as shown in Figure 5.4. We can see the positions (longitude/latitude) of the moving object changing for each time tick on the z-axis. We will divide the z-axis into subsets called Time Deltas. In Figure 5.5, each rectangle of a different shade of blue represents one Time Delta. By doing this, if we call value_at_timestamp at $z = 3$, only the trajectory of the object contained in the blue rectangle at $[z = 0, z = 5]$ is considered. The binary search to find the bound will be done on a much smaller number of instants from the trajectory.

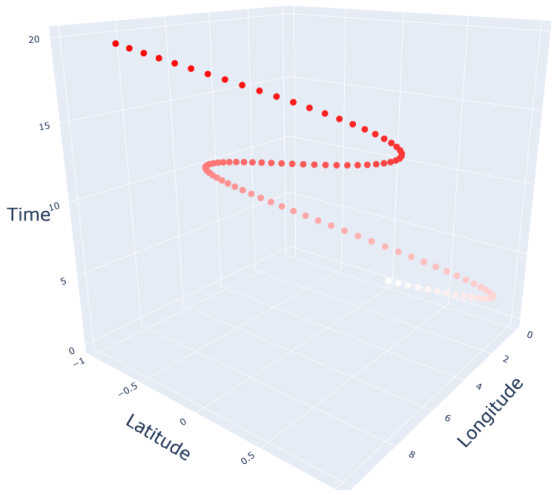


Figure 5.4: Illustration of a moving object through time

With this new configuration, we no longer fetch the entire trajectories from the MobilityDB database. Instead, we continuously fetch sections of all trajectories contained in the ongoing Time Delta of the animation. This operation is easily implemented in MobilityDB, as displayed by the Code example 5.2. This approach will, on average, drastically reduce the time required to compute the value_at_timestamp operation, as the binary search operation will be applied to a smaller number of instants. This will directly reduce the time to compute on_new_frame and, by extension, increase the FPS since it is inversely proportional to the on_new_frame time. The memory consumption throughout the animation will also be heavily reduced.

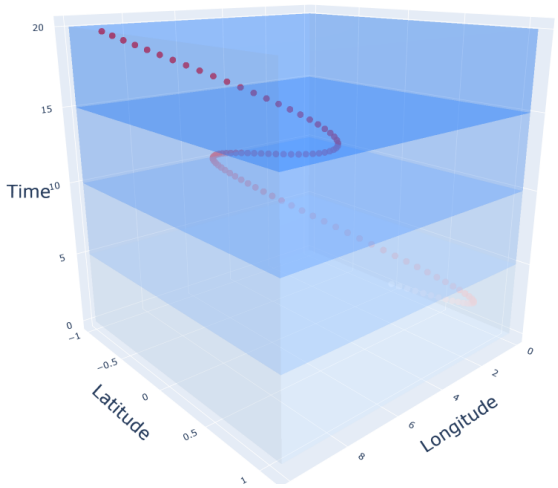


Figure 5.5: Illustration of the Time Deltas: time axis divided into subsets

```
SELECT attime(trip::tgeompoint,
             span('2023-06-01 00:00:00'::timestamptz,
             '2023-06-01 12:00:00'::timestamptz, true, true))
             FROM public.ships;
```

Listing 5.2: SQL query to fetch a Time Delta

5.2.1 Adding QGIS Tasks to the Design Architecture

Figure 5.6 shows the new design architecture that incorporates the Time Deltas. In this new design, we keep the initial step of initiating the vector layers with the QGIS Features associated with all trajectories, alongside the activation of the Temporal Controller and the vector layer's temporal properties. In the second step, the Python script will constantly update its buffer of Time Deltas. While the animation traverses the ongoing Time Delta, a QGIS Task (see Section 3.4.4) will create the next Time Delta in the background. When the animation reaches the current Time Delta's bound, the script will move on to the next Time Delta and start a new QGIS Task to fetch the following Time Delta. This operation repeats throughout the animation, allowing for an uninterrupted animation.

5.2.2 Size of an Individual Time Delta

The size of a Time Delta is defined in the number of frames that it covers, we define it with the following parameter in this thesis report : T_Delta_Size . Decreasing the T_Delta_Size parameter will lower the time cost of value_at_timestamp, but there exists a lower bound, a very small T_Delta_Size will lead to more frequent calls to the database, and even in the best case scenario, where the database call itself does not incur noticeable time cost, the basic operations needed to make this

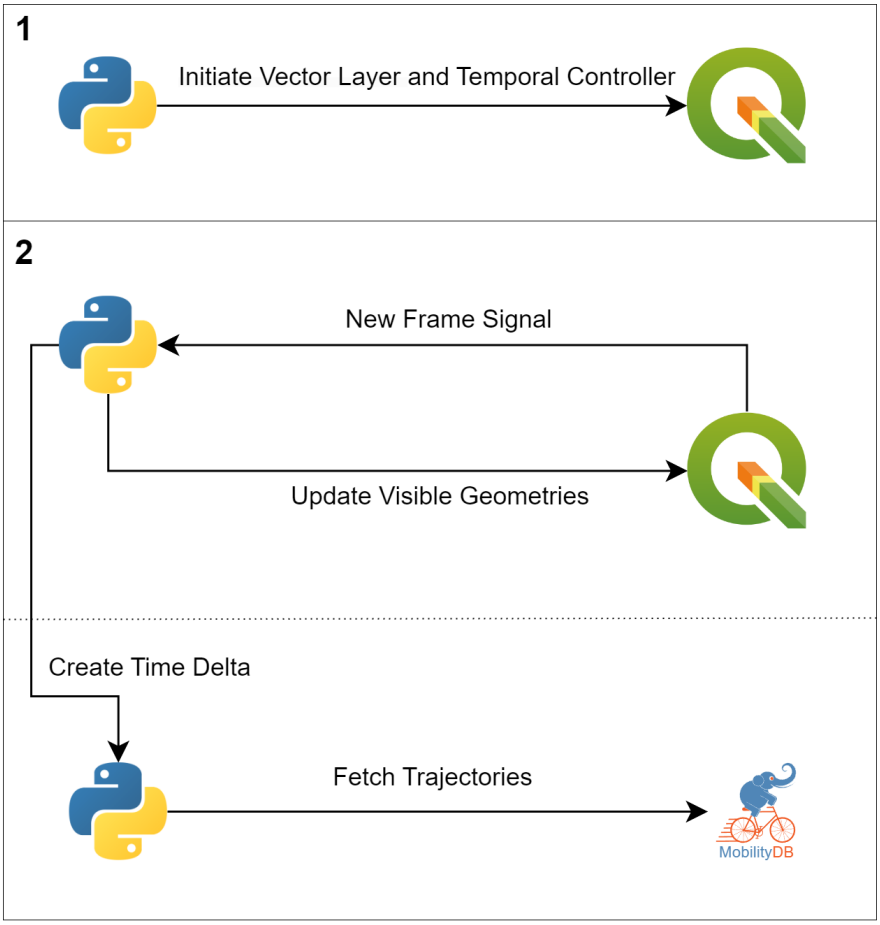


Figure 5.6: Design architecture of the TimeDeltas Mode

call and store the results have a constant overhead. This overhead can accumulate, resulting in a performance bottleneck. Therefore, it is crucial to find an optimal T_Delta_Size that balances the reduction in value_at_timestamp time costs and the frequency of database queries. Proper tuning of this parameter is essential to achieve efficient performance without incurring unnecessary overhead from too many database operations.

Buffering System

Storing all Time Delta creates a significant memory usage concern, but we can mitigate this with a buffering system by only keeping data associated with at most three time deltas at all times during the animation (previous, current, and next time deltas).

5.2.3 Time Delta Frame rate Cap

The FPS formula given in Section 5.1.2 relies entirely on the on_new_frame time cost. With a Time Deltas architecture, we also have to take into consideration the time it takes for a QGIS Task to fetch the upcoming Time Delta, such that

the animation can continue smoothly without interruptions. This adds another constraint when defining the frame rate: to have an uninterrupted animation, the QGIS Task fetching the next Time Delta must complete before the animation reaches the end of the current Time Delta.

This means that the frame rate is determined first by the on_new_frame function time, which defines the rate at which the script can generate a frame. Secondly, we have to consider the frame rate cap set by the QGIS Task runtime, which defines the maximum frame rate for an uninterrupted animation. We ultimately have to take the smallest of the two values as the final FPS value.

It is impossible to accurately predict the time that the QGIS thread will take to fetch the next Time Delta. For this reason, we will use the previous fetch time as our heuristic. This is not ideal, as there can be scenarios where the upcoming Time Delta might take significantly longer than the previous one, leading to the animation being paused to wait for the QGIS Task to complete. As an example, consider two Time Deltas spanning the start of the day for the STIB's metro, bus and tram lines in Brussels. The first Time Delta might have none or just a couple of trajectories in it, while the second might have substantially more trajectories, leading to a longer QGIS Task completion time.

We define the FPS value using Formula 5.2.3: Let ω be the time to compute on_new_frame, and δ be the compute time of the QGIS Task creating the upcoming Time Delta in the background. Since we cannot predict δ, we use $\delta_{T\Delta-1}$, which is the QGIS Task completion time for the previous Time Delta.

$$\text{FPS} = \min\left(\frac{1}{\omega}, \frac{\texttt{TIME\_DELTA\_SIZE}}{\delta_{T\Delta-1}}\right)$$

5.2.4 Computation Inside a QGIS Task

We tested two designs for this solution, each with a different computation load for both the QGIS Tasks and the on_new_frame. We describe below the trade-offs and how we made our final decision in the selection process.

Design 1: Heavy QGIS Task Computation

In Section 5.1, we stored the entire trajectories in a list or hashmap and then applied value_at_timestamp to the trajectories inside on_new_frame. However, with smaller Time Deltas, we can move the value_at_timestamp calls inside the QGIS Task and store the positions of each trajectory for all the frames of the Time Delta inside a matrix data structure. Figure 5.7 shows an example of how the data is structured inside the matrix, with the columns representing frames and the rows representing a unique trajectory. Each cell of the matrix contains either the position of trajectory i at frame j, or an Empty Point otherwise(this can be implemented in Python using Shapely Point or their WKT or WKB representations (see Section 3.1.1).

It is important to note that using Python lists to create the matrix is extremely inefficient. Python has a scientific library, NumPy [16], built specifically to han-

dle vector and matrix operations very fast with low memory usage, as shown in Table 5.5. During the animation, on_new_frame only has to take the column associated with the current frame and update the geometries of all trajectories, eliminating the time cost of computing the positions. In some scenarios, we expect many empty Points cells if the dataset is sparsely filled, where, just like sparsely populated graphs, we can shift from a matrix representation to an adjacency list or sparse matrix representations to optimize memory space even further.

	2023/06/01 00:01	2023/06/01 00:02	2023/06/01 00:03
Boat 1	EMPTY POINT	POINT(8.42335 55.4718)	POINT(8.42335 55.4718)
Boat 2	POINT(12.6059 55.6845)	POINT(12.606 55.6845)	POINT(12.606 55.6845)
Boat 3	EMPTY POINT	EMPTY POINT	EMPTY POINT

Figure 5.7: Illustration of a Matrix storing the positions from Solution 5.2

Percentage of the Danish AIS dataset	**T_Delta_Size = 30 Frames**	**T_Delta_Size = 60 Frames**	**T_Delta_Size = 120 Frames**
10%	0.400 MB	0.799 MB	1.598 MB
50%	1.998 MB	3.996 MB	7.992 MB
100%	3.997 MB	7.994 MB	15.988 MB

Table 5.5: Total size of a single Time Delta matrix

This design, although promising in theory, faced many challenges when we tried implementing it. As discussed in the limitations of running background tasks with QGIS (see Section 3.4.4), heavy computation in the QGIS Task thread impacts the performance of the main QGIS thread. This issue is exacerbated on lower-end hardware configurations, such as the Laptop configuration.

So even though it does help reduce the average on_new_frame time by a lot, by extension higher FPS values, it does so in exchange of a different FPS cap. The thread being computationally heavy, require more time to complete, creating a much lower FPS cap for a uninterrupted animation. On top of that, the QGIS Task being run in the background heavily impact the running animation, almost making it halt for the duration of the task on lower end hardware, or in the best scenario reducing the FPS values on higher end configurations.

Design 2: Balancing QGIS Task and on_new_frame load

In the second design choice, QGIS Task's only computation is the fetching of trajectories for the upcoming Time Delta and storing them in a hashmap. The po-

sitions are then computed directly on-the-fly in the on_new_frame function with value_at_timestamp calls on each of the smaller trajectories, just like in Solution 5.1. As we show in the results below, this approach provides a great mix of improvements to average FPS over the Interactive Mode and a lower memory consumption thanks to the Time Deltas architecture. By reducing the QGIS Task computation, we also increase the max FPS for uninterrupted animation.

Final Choice

Ultimately, the first design choice would have given us a better FPS bound if the computation of the positions could be done faster and without the limitations of multithreading code set by Python itself. We will continue forward with the second design; however, in Chapter 7, we explore ways of revisiting the first design choice.

5.2.5 Benchmarks

We will measure the performance of the Time Delta solution by looking at the average FPS, FPS stability, and memory consumption throughout the animation. We also provide direct comparisons with the Interactive Mode. For the benchmark of the TimeDeltas Mode, we will primarily focus on the lower end hardware laptop configuration since the QGIS Task architecture is more CPU intensive, it will give us better picture of the overall performance. The test bench for the following results is the lower-end laptop configuration and the Danish AIS boats dataset.

Tuning the size of a Time Delta

An important part of the Time delta design architecture is the size of a single Time Delta, that we note T_Delta_Size. Ideally we would like to have the smallest possible T_Delta_Size such that the value_at_timestamp calls are done on the shortest trajectory possible, but at a certain threshold, if we pick a T_Delta_Size too small, the constant overhead needed to create and run the QGIS task and then store its result, will force us to interrupt the animation.

We look at the impact of the T_Delta_Size on the average FPS, for this we isolate the run of the following test configuration : fifty percent of the Danish AIS dataset on the laptop hardware configuration. Figure 5.8 displays the average FPS value over the initial 120 frames of the animation with a 1 minute time granularity. From these results we observe that the optimal T_Delta_Size in this experiment is between 20 and 60 frames, a lower size results in the overhead impacting the possible max FPS, and going higher would lead to slower value_at_timestamp calls. In our research we opted to pick a static T_Delta_Size throughout the animation, but for future research it would be interesting to look at a dynamic T_Delta_Size parameter.

Average FPS

Table 5.6 provides the average FPS for the TimeDeltas Mode and directly compares them with the results of the Interactive Mode from Section 5.1. We use

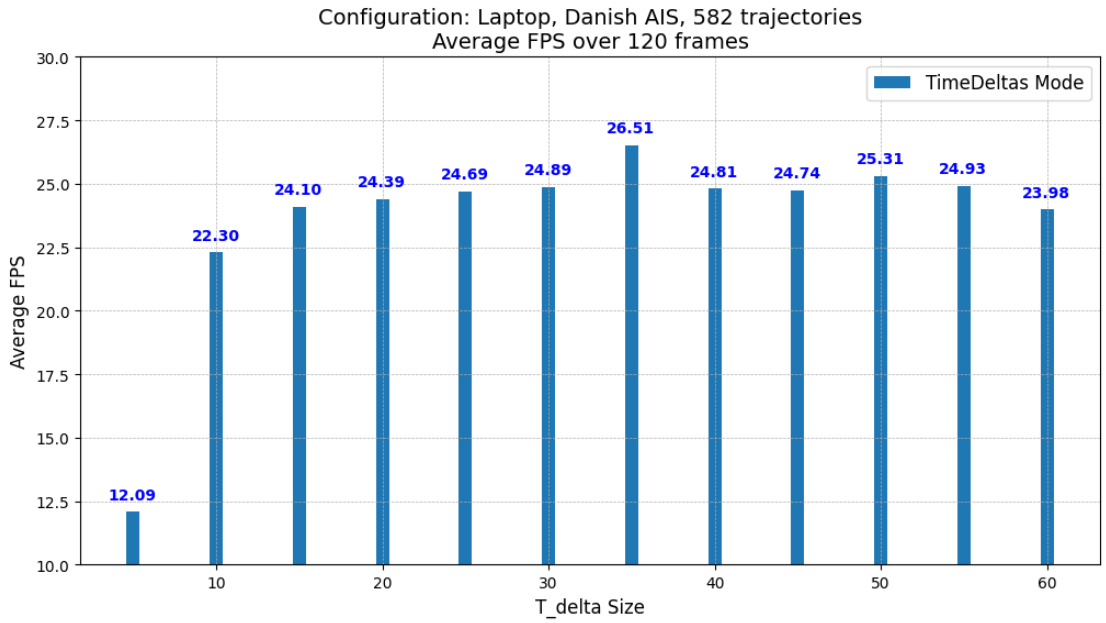


Figure 5.8: Average FPS(over 120 frames) for TimeDeltas Mode for different T_Delta_Size parameters

the exact same test bench configuration: average over the initial 120 frames, with T_Delta_Size set to 30, for three subsets of different sizes from the Danish AIS boats dataset. From these results, we see a clear but small improvement in the average FPS for all dataset sizes.

Percentage of the Danish AIS dataset	**Interactive Mode**	**TimeDeltas Mode**
10%	20.938	24.554
50%	3.819	5.419
100%	1.954	2.534

Table 5.6: TimeDeltas Mode : Average FPS of results

Frame rate Stability

We compare the frame rate stability of the TimeDeltas Mode with the Interactive Mode. The test configuration is the same for both, we use 10 percent of the DanishAIS dataset(582 boats), on the lower end laptop configuration, with a T_Delta_Size of 30 frames. Figure 5.9 displays the FPS value over the initial 120 frames of the animation(1 minute granularity), in orange the TimeDeltas Mode, in blue the Interactive Mode. From this plot, we observe that the average FPS does improve with the TimeDeltas Mode, however we can clearly see the moments where a QGIS task is working in the background with the visible dips in FPS at every 30 frames.

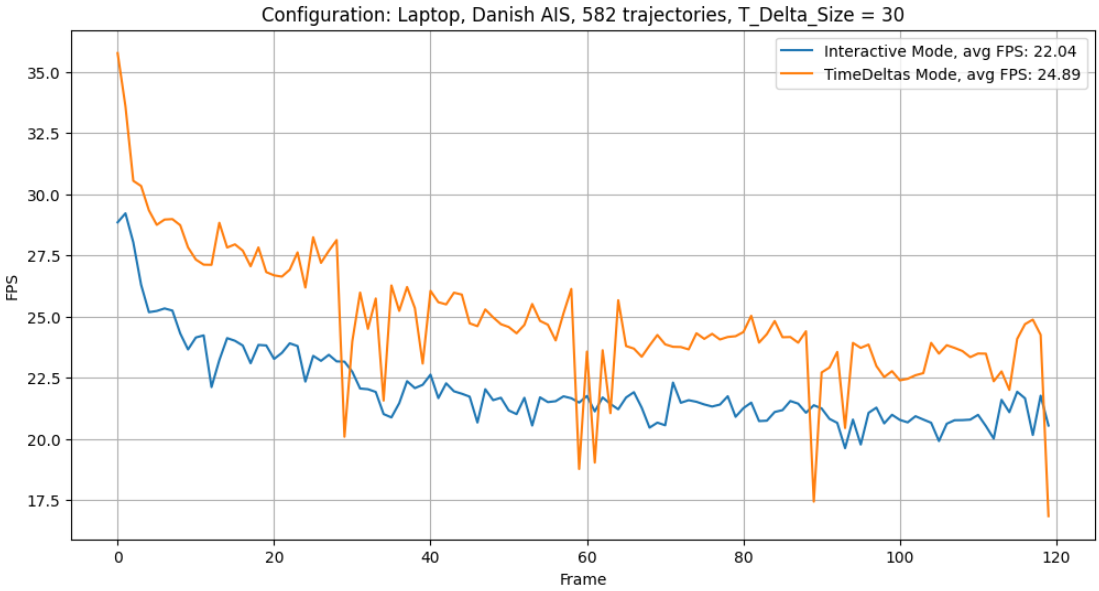


Figure 5.9: TimeDeltas Mode results : frame rate stability

Memory Usage

For a direct comparison with the Interactive Mode, we measure the memory usage for the TimeDeltas Mode with the following setup: T_Delta_Size of 30 frames, laptop hardware configuration, two subsets of the Danish AIS and STIB datasets. The animation is run with only the vector layer in the QGIS process, and the measurements are taken when the buffer is full. The results are shown in Table 5.7, we observe that for the Danish AIS datasets, we don't observe much of a difference, however the STIB dataset shows the power of this technique as most of the trajectories do not overlap, resulting in drastically lower memory consumption.

Dataset	**Interactive Mode**	**TimeDeltas Mode**
Danish AIS : 582 trajectories	0.59	0.58
Danish AIS : 5821 trajectories	1.80	1.80
STIB : 100000 trajectories	8.00	1.20
STIB : 497609 trajectories	Not enough RAM	4.50

Table 5.7: TimeDeltas Mode results : memory usage (in GB)

User Experience

The architecture of the Time Deltas is conceptually dependent on the frames of the animation. If a user modifies the configuration—such as the duration between

two frames, the time granularity, or the temporal extents—the script will require a reset of the Time Deltas buffer and will reload the animation. This introduces delays to any Temporal Controller configuration change, creating a potentially frustrating user experience. Additionally, there is a reload time when a user skips the timeline to a completely different Time Delta.

5.2.6 Final Remarks on the TimeDeltas Mode

In conclusion, we observe that the Time Deltas architecture provides a slightly higher frame rate in terms of average FPS, but this comes at the cost of lesser stability, which can be corrected by capping the FPS to a lower value. The biggest improvement is found in the minimal memory footprint throughout the animation, enabling users to create visuals for extremely large datasets. Figure 5.10 shows a running example of the entire STIB dataset (approximately 6 GB in the MobilityDB database) on a laptop with 16 GB of RAM.

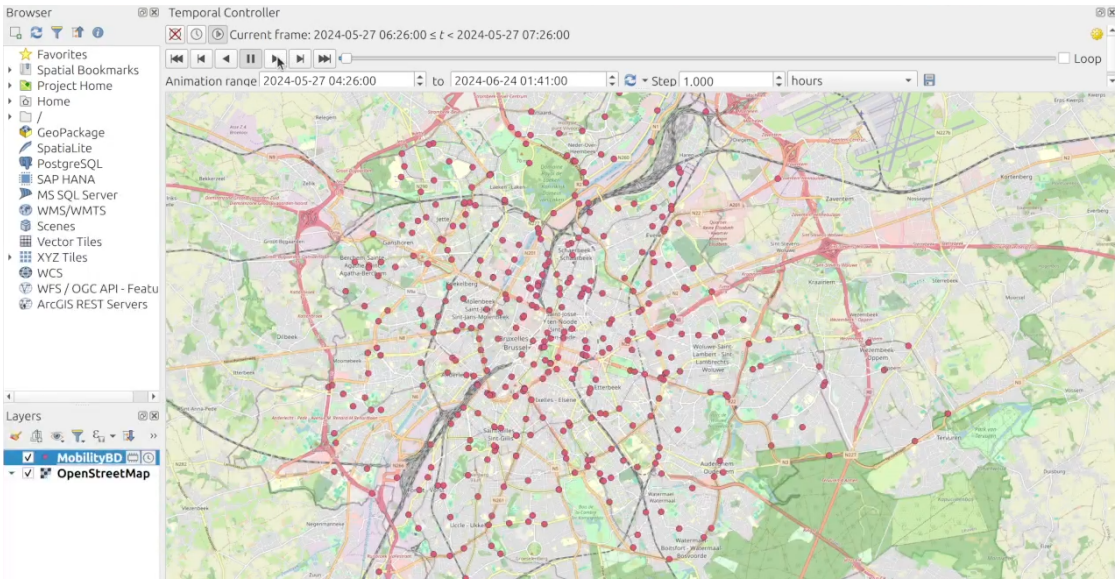


Figure 5.10: Visual of the entire STIB dataset inside the TimeDeltas Mode

However, the downside of the Time Deltas TimeDeltas Mode is the loss of seamless interaction with the Temporal Controller. Any modification to the Temporal Controller configuration requires reloading the animation, which creates a delay and can result in a frustrating user experience. In Chapter 6, we experiment with an entirely different approach, delving into the realm of vector tile servers. Additionally, in Chapter 7, we provide an alternative solution that builds upon the Time Deltas architecture. In theory, the Time Deltas architecture is very interesting and provides a blueprint that can be implemented in other GUI visualization tools. However, the limitations imposed by QGIS Task's overhead, Python's threading, and the interactive nature of the Temporal Controller unfortunately render this solution less practical than the initial Interactive mode proposed in Section 5.1.

5.3 Limiting Geometries to Map View

One major point of contention with the previous solutions is that they do not consider the visible extent of the map on the screen. The script continuously updates the positions of all objects, even those no longer visible on screen, hindering performance without benefiting the user.

To illustrate why this is an issue, in Figure 5.11 we plot the FPS upper bound depending on the number of geometries to update at each frame. We obtained these results by removing all computation from on_new_frame and only keeping the geometries update operation. We then measured the FPS for different numbers of QGIS Feature geometries. The FPS values follow a logarithmic relation with the number of QGIS features's geometries to update. The results shown here are obtained in the Desktop configuration, but the relation remains consistent across all hardware configurations. Therefore, it is in our best interest to reduce the number of geometries that we need to update on every new frame for a better FPS values.

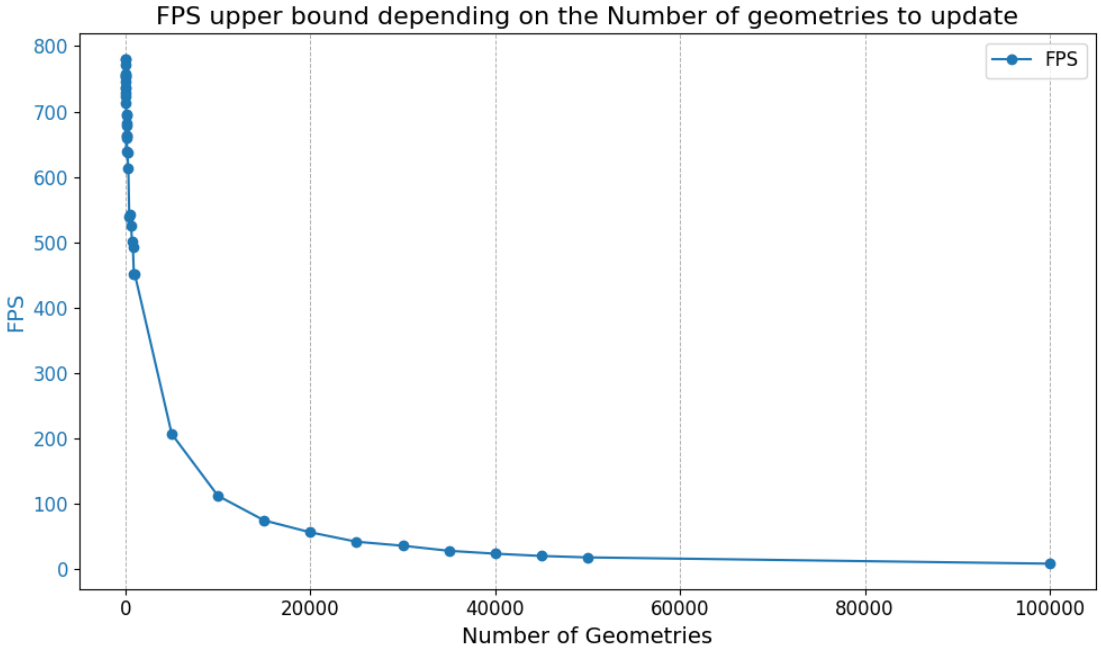


Figure 5.11: FPS limit by number of geometries per frame

In Chapter 6, we cover a vector tile layer approach that addresses this consideration (and more). In this section, we describe ways of overcoming this problem with the help of bounding boxes in both the Interactive 5.1 and TimeDeltas Modes 5.2. In spatial databases, we can restrict the geometries to a given area, notably a rectangle called a bounding box. In PostGIS, this can be done with the help of ST_extent, which returns a box2d, a two-dimensional bounding box. MobilityDB implements this concept for spatiotemporal types and proposes the atSTBox operation, which enables us to restrict the geometries to a given bounding box for a given time range.

We illustrate atSTBox with the Figure 5.12. We are able to restrict the points of the trajectory shown in Figure 5.4 inside a bounding box (represented by the yellow rectangular prism) for the entire time range. We can see a top-down view that shows exactly all the points that are outside the bounding box and that are never shown or loaded during the animation.

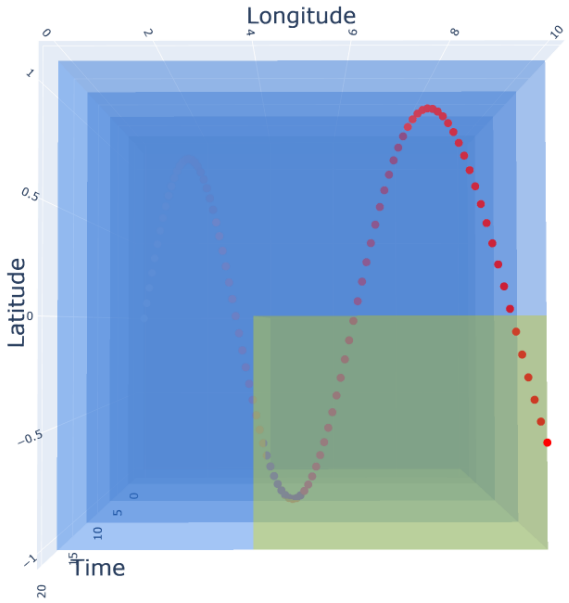


Figure 5.12: Illustration of atSTBox excluding points outside of the bounding box (represented by the yellow rectangular prism)

QGIS emits a signal whenever the user changes the visible extent by zooming in or out. We can attach this signal via PyQGIS in our Python script. In Solution 5.1, we load all trajectories at the start. It is not reasonable to automatically reload the entire trajectories any time the user changes the map's visible extent (as shown in Table 5.2, we already suffer from long wait times for the fetch operation). One solution can be to create a specific reload button that would allow users to choose the extent and then load all the trajectories.

In the TimeDeltas Mode, we can take advantage of the Time Deltas to adapt the extent to the visible extent when requesting a new Time Delta. This also has the consequence of reducing the QGIS Task time if the user chooses an extent that contains fewer trajectories. However, since we fetch the next Time Delta in the background, if an extent change occurs after the QGIS thread has started, the user will have to wait for the $t + 2$ Time Delta to see the change in extent visible on screen, becoming less optimal if the Time Delta size is large.

5.4 Summary of Vector Layer Implementations

This chapter details the two main solutions we propose to extend the MOVE plugin. The first implementation stores the entirety of the spatiotemporal objects in memory via PyMEOS. At each frame, the geometries associated with each trajectory are updated using the value_at_timestamp operation, with the timestamp taken from the Temporal Controller configuration. This means that users can modify the animation's configuration after loading the trajectories and see the changes applied during the next frame generation. This ability to interact without delay or load time with the Temporal Controller is the reason behind the name "Interactive Mode".

However, storing the entirety of the trajectories has a significant downside: the memory usage is extremely high. We are not even able to load the entire STIB dataset on 32 GB of RAM using the Interactive Mode. To tackle this issue and improve the average FPS by reducing the time cost of value_at_timestamp, we propose a design architecture called TimeDeltas Mode. It divides the timeline into subsets called Time Deltas. By only loading the portions of trajectories contained in the current Time Delta, we dramatically lower the memory footprint and, on average, improve the FPS. The trade off for these improvements is the loss of high interactivity with the Temporal Controller. Designing an architecture to manage the Time Deltas and the QGIS Task thread that loads the upcoming data in the background is tricky and limits how much a user can modify the configuration on the fly. For this reason, we propose both Interactive Mode for small to medium-sized datasets and TimeDeltas Mode to create visualizations for very large datasets.

In the final section, we cover the notion of restricting geometries to the visible extent only. We mention atSTBox and how it can enable such functionality in the vector layer solutions we propose in this chapter. In the next chapter, we will cover the vector tiles layer solution in which this consideration is inherently taken into account. However, unlike the vector layer solution, we will need to measure the accuracy of the data.

Chapter 6

Vector Tile Layer

In this chapter, we present a solution that incorporates PgTileServ (detailed in Section 3.3). Vector tile servers are highly efficient for visualizing large datasets, as they consider the visible extent and resolution at which the data is actively being viewed, ensuring that only the necessary data is processed and displayed. We present a design architecture using a PgTileServ to connect a mobilityDB database to QGIS's built-in vector tile layers (Section 3.4.3).

6.1 Design Architecture

Our solution integrates several key components that enable the visualization of spatiotemporal objects using vector tiles in QGIS:

1. **QGIS's Vector Tile Layer (QgsVectorTileLayer):** This tool is instrumental in displaying vector tiles within QGIS, we detail its working in Section 3.4.3.

2. **PgTileServ:** A robust server that generates and serves vector tiles from PostGIS-enabled databases.

3. **Temporal Controller:** Enables temporal navigation of the dataset, detailed in Section 3.4.1.

4. **Custom PgTileServ functions for MobilityDB:** Specialized functions designed to integrate MobilityDB with the PgTileServ, ensuring seamless data retrieval and display.

Figure 6.1 illustrates the architecture of our solution. Unlike the methods discussed in Chapter 5, this approach automates the management of visible geometries through the QGIS vector tile layer and the PgTileServ. In this iteration, the QGIS vector tile layer handles the rendering of vector tiles, which are served by the PgTileServ. The Temporal Controller emits a signal whenever its state is changed. The Python script then deletes the existing vector tile layer and creates a new one with a URI containing the correct temporal settings, dynamically updating the vector tiles. Custom functions in the PgTileServ interact with MobilityDB to fetch and process the required data efficiently.

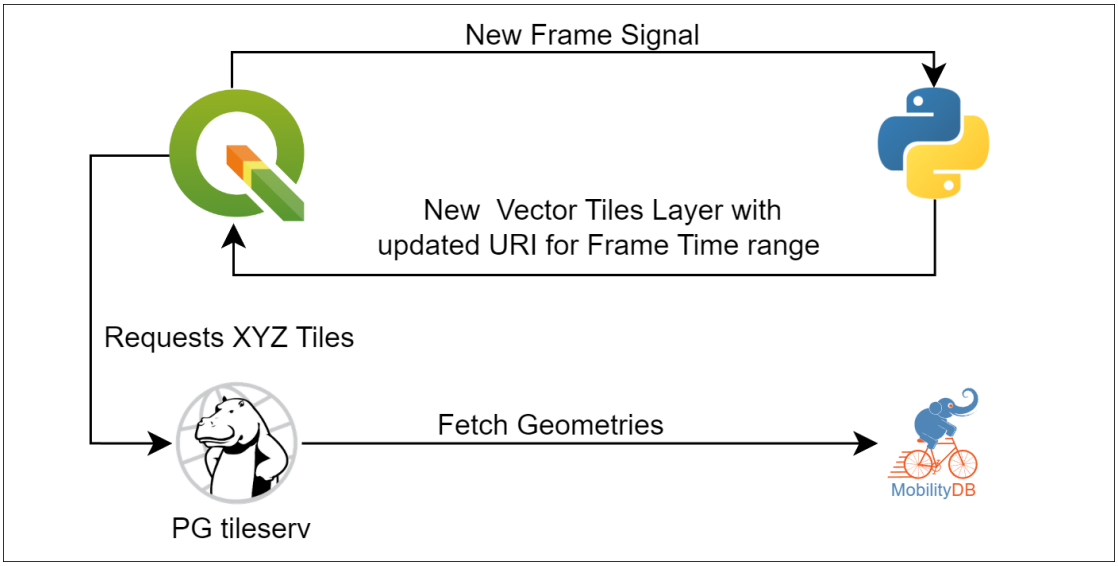


Figure 6.1: Design architecture of the vector tiles solution

6.1.1 Connecting PgTileServ with MobilityDB

Implementing this solution involves creating custom functions for the PgTileServ to interact with a MobilityDB database. Listing 6.1 demonstrates a SQL function designed to retrieve geometries from a spatiotemporal column within the MobilityDB database. This function transforms the data into a format suitable for rendering as vector tiles. First, the function defines the bounds of the tile based on the zoom level and x, y coordinates. It then retrieves a subset of trajectories from the dataset that are contained within the selected tile, applies temporal filtering, and converts the spatiotemporal geometries into regular PostGIS geometries with the asMVTGeom function from MobilityDB. These geometries are then converted into vector tile format using PostGIS's ST_AsMVT function. The resulting bytea output is then returned to the QGIS vector tile layer.

6.1.2 Performance Considerations

The performance of the vector tile server solution is influenced by several factors:

- **Zoom Level:** Higher zoom levels result in more tiles and higher resolution, increasing server load.
- **Caching:** Effective caching mechanisms can significantly improve performance by reducing the need to regenerate tiles for frequently viewed areas.
- **Request Frequency:** Managing the frequency of requests to the server helps prevent overload and ensures smooth performance.

In our implementation, we utilized various techniques to optimize performance, such as limiting the number of points retrieved, applying spatial and temporal filtering, and leveraging caching mechanisms.

```sql
CREATE OR REPLACE FUNCTION public.tpoint_mvt(
    z integer, x integer, y integer, p_start text, p_end text, number_of_points
        integer DEFAULT 500
) RETURNS bytea
AS $$
WITH bounds AS (
    SELECT ST_TileEnvelope(z, x, y) AS geom),
trips_ AS (
    SELECT * FROM PyMEOS_demo ORDER BY mmsi LIMIT number_of_points),
vals AS (
    SELECT mmsi, numInstants(trip) AS size,
        asMVTGeom(attime(trip, span(p_start::timestamptz, p_end::timestamptz, true,
            true)),
                transform((bounds.geom)::stbox, 3857))
        AS tgeom
    FROM (
        SELECT mmsi, trajectory::tgeompoint AS trip
        FROM trips_
    ) AS ego, bounds),
mvtgeom AS (
    SELECT (tgeom).geom, size, mmsi
    FROM vals)
SELECT ST_AsMVT(mvtgeom)
FROM mvtgeom;
$$
LANGUAGE 'sql'
STABLE
PARALLEL SAFE;
```

Listing 6.1: SQL function to fetch MobilityDB data with XYZ format

6.2 QGIS and PgTileServ Limitations

The vector tile layer's URL is the endpoint that connects directly to the PgTileServ. A significant limitation encountered was the inability to update the URL of a vector tile layer in QGIS once it is created. At every frame we need to update the time range parameters of the vector tile layer's URL in order to request the correct spatiotemporal data. To bypass this limitation, our implementation deletes and creates a new vector tile layer with the updated time parameters, this is done for each frame of the animation. A potential workaround involves creating a proxy server that handles requests from the QGIS layer, forwarding them to the PgTileServ with the correct time bounds. This method could manage request frequency and prevent server overload but adds complexity to the system architecture, adding another separate component on top of the PgTileServ itself.

Limitations

Several limitations were identified in using the PgTileServ as our final solution:

- **Customization Options:** For animations, entire layers must be deleted

and recreated for each frame, so user changes to symbology are always reset To counter act this, we can implement a custom symbology settings that is applied to every new layer.

- A proxy server might be necessary to manage requests, adding complexity and involving multiple components (QGIS, proxy, PgTileServ, MobilityDB).
- Accuracy is dependent on the zoom level. Higher zoom levels increase resolution and the number of tiles needed, leading to more server calls and potential overload.
- Since layers are recreated for each frame, any user modifications to the symbology will only last for a single frame. To address this, symbology settings could be saved in an XML file and reapplied to each new layer.
- The current implementation of MVTGEOM returns trajectories for the time range between frames, displaying lines instead of points.

Advantages of a Vector Tile Solution

Despite the limitations listed above, the vector tile server approach offers several significant advantages for visualizing mobility data:

1. **Efficient Data Handling:** The vector tile server automatically considers the visible extent, excluding unnecessary points and optimizing data handling without requiring manual creation of a bounding box in queries.
2. **Smooth Animation:** With an effective caching system, animations can be made smooth and responsive, with minimal delay when adjusting the temporal controller, thus enhancing interactivity.
3. **Interactive Controls:** Provides interactive controls for zooming, panning, and adjusting the temporal range.

6.3 Summary of the Vector Tiles Experiment

Considering the challenges and limitations encountered during the implementation of our architecture in QGIS, we were unable to develop a functioning proof of concept to implement in MOVE. For this reason, we do not include benchmarks for this solution. Nevertheless, a working demo is available on the GitHub repository of the research project for interested readers, with detailed instructions on how to replicate the setup. Figure 6.2 shows a running example of the demo.

We did not explore this research path further, as our initial aim remains to provide a self-sufficient, downloadable plugin from the QGIS market that doesn't require any other setup. However, we still wanted to lay the groundwork for future developments in the field. The insights gained from this work will help future efforts to develop a vector tiles solution to visualize MobilityDB data inside QGIS.

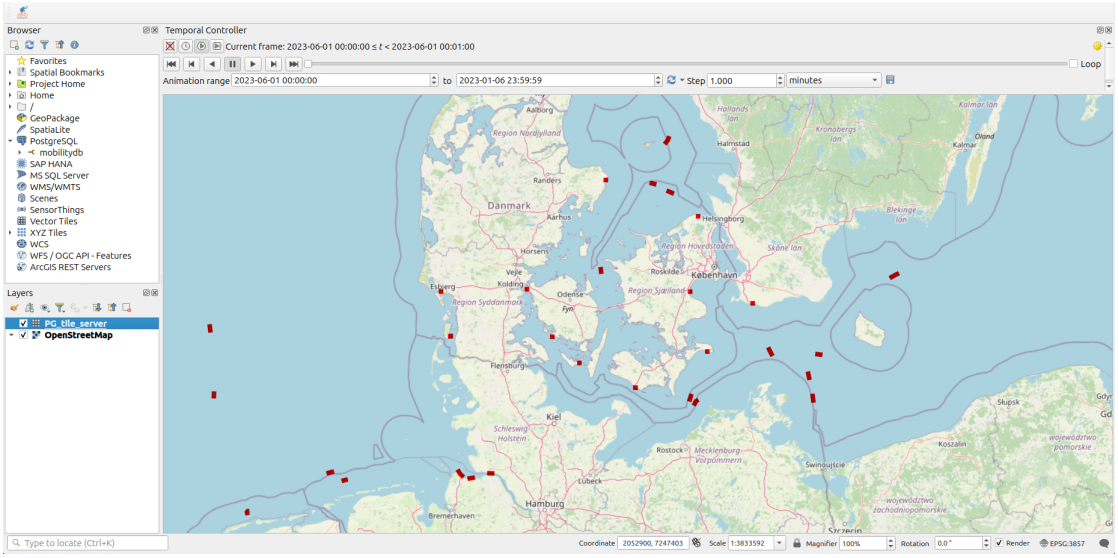


Figure 6.2: Illustration of the vector tile layer demo

Specifically, future work could explore the development of a proxy server to manage request frequency and improve the overall efficiency of the system. Additionally, further optimizations could be made to handle higher zoom levels and improve accuracy without overloading the server. Experimenting with different caching mechanisms could enhance performance and interactivity.

In the next chapter, we take a step back to revisit the TimeDeltas Mode. We described in the previous chapter how some of the limitations regarding multi-threading in Python and the computation of the trajectories' positions created significant challenges for the implementation of the time deltas idea. In the next chapter, we will delve deeper into the field of parallel processing in Python and deliver a solution that extends the TimeDeltas Mode.

Chapter 7

Resampling Trajectories

In chapter 5 we explored different paths to visualize and animate MobilityDB trajectories with the help of the value_at_timestamp method to compute the positions we display. This operation, even if done in O(log n) time complexity, can become a big time sink when applied to a huge number of trajectories. In this chapter, we detail an architecture that does not use value_at_timestamp to compute the positions.

7.1 Reconstructing Existing Trajectory

MobilityDB and MEOS provide a powerful operation called tsample [28]. This operation reconstructs the instants contained in a trajectory. This functionality is particularly useful in scenarios where data needs to be sampled at regular intervals. The tsample function samples a temporal value with respect to a given interval. The syntax is shown in Listing 7.1. If the origin is not specified, it defaults to Monday, January 3, 2000. The interval must be strictly greater than zero. Figure 7.1, illustrates the sampling of temporal floats with various interpolations. As shown in the figure, the sampling operation is best suited for temporal values with continuous interpolation.

```
tsample({tnumber,tgeompoint}, duration interval, torigin timestamptz='2000-01-03',
    interp='discrete') -> {tnumber,tgeompoint}
```

Listing 7.1: tsample syntax

7.2 Resampling Mode

For this new solution, we reuse the Time Deltas architecture presented in Section 5.2. In the previous chapter we discussed a design choice for the TimeDeltas Mode in which we would compute the positions of all the trajectories directly inside the QGIS Task and store them in a numpy Matrix. This design choice has two main limitations:

1. Python Limitation : Multithreading in Python isn't done in parallel and every QGIS Task thread impacts the performance of the main animation thread

[1]https://mobilitydb.github.io/MobilityDB/master/ch09s02.html

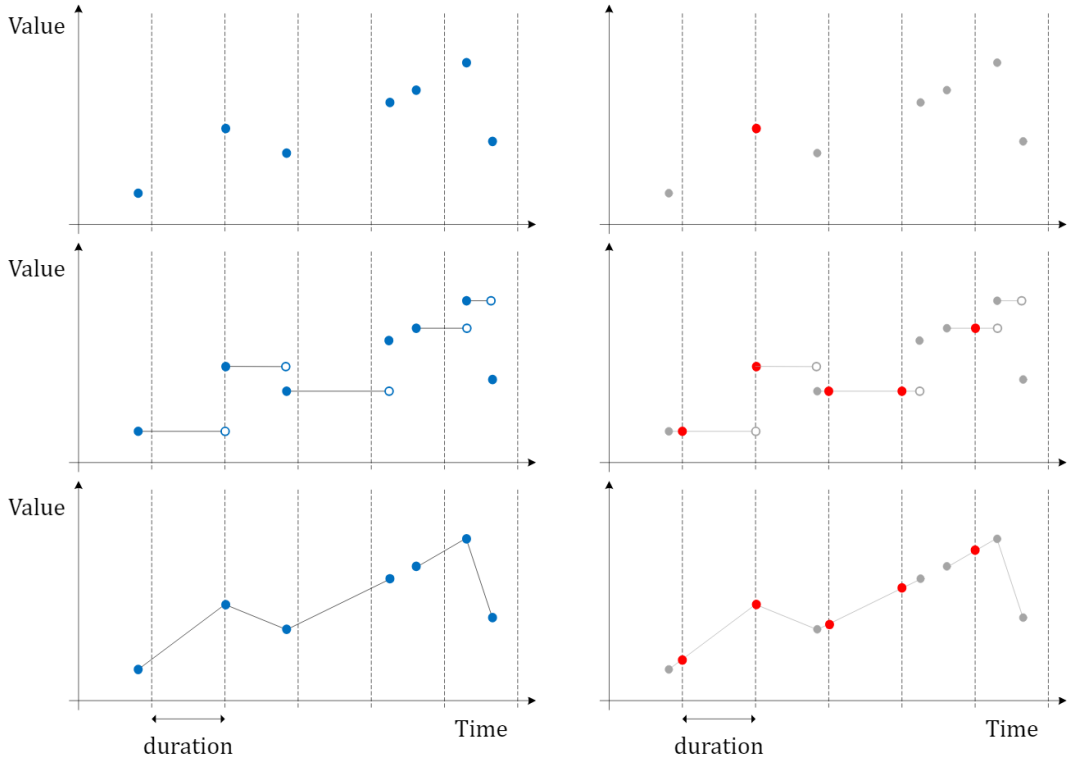


Figure 7.1: Sampling of temporal floats with discrete, step, and linear interpolation [1]

2. Position computations : The time cost for computing the positions of each trajectory using value_at_timestamp, even if done on smaller trajectories is still noticeably large on big datasets.

We address both of these issues in the Resampling Mode, firstly, to reduce the computation time of all the trajectories position's, we use the sampling functionality of MobilityDB, this enables us to create instants for the exact time interval selected by the user in the Temporal Controller. This completely removes the need to call value_at_timestamp, as each instant of the resampled trajectory corresponds to one frame of the Temporal Controller animation. The example Code 7.2 shows how easily this can be done through MobilityDB. Since we apply the resampling operation only on a subset of all the trajectories, the memory consumption is bounded by both the number of trajectories(rows) and the number of frames in a time delta(columns). The only computation inside the QGIS Task is now the assignment of all the points of each trajectory to its corresponding row in the Numpy Matrix, this is done very efficiently due to the slicing operations in Numpy, that are run using very fast low-level libraries capable of bypassing Python's GIL and are run in parallel. The start and end indexes of the example represent the bounding frames of the trajectory inside the Time Delta.

The design architecture for the Time Deltas with resampling solution is the same as shown in Figure 5.6. However, it differs in the implementation of the second step. In step 2, when creating the Time Deltas inside the QGIS Task, instead of storing the subsets of trajectories themselves, we create a Numpy Matrix

```
WITH trajectories as (
SELECT
    attime(trip::tgeompoint,
        span('2023-06-01 00:00:00'::timestamptz,
        '2023-06-01 12:00:00'::timestamptz, true, true)) as trip
    FROM public.ships),
    resampled as (
SELECT
    tsample(trip, INTERVAL '1 MINUTE', TIMESTAMP '2023-06-01 00:00:00') AS
        resampled_trip
    FROM trajectories)
SELECT
    EXTRACT(EPOCH FROM (startTimestamp(resampled_trip) - '2023-06-01 00:00:00'::
        timestamp))::integer / 60 AS start_index ,
    EXTRACT(EPOCH FROM (endTimestamp(resampled_trip) - '2023-06-01 00:00:00'::
        timestamp))::integer / 60 AS end_index,
    resampled_trip
    FROM resampled;
```

Listing 7.2: SQL query to fetch a Time Delta of resampled trajectories

in which we store the positions, in WKB, of all the trajectories for each frame of the Time Delta. This means that the on_new_frame function that computes the "Update Visible Geometries" part no longer needs to compute the positions, it will use the column associated to the current frame in the Time Delta, and update the geometries with the column's values.

7.2.1 Implementing Parallel Code in Python

Parallel processing in Python presents challenges due to the Global Interpreter Lock (GIL), which prevents multiple native threads from executing Python bytecodes at once. This section explores the impact of the GIL on multithreading and presents strategies for achieving parallelism inside QGIS Tasks using multiprocessing.

Global Interpreter Lock (GIL)

Python's design includes a Global Interpreter Lock (GIL) to simplify memory management and ensure thread safety within the CPython interpreter [3]. The GIL allows only one thread to execute at a time, even on multi-core systems, which prevents true parallel execution of threads. This limitation significantly affects CPU-bound programs, as the GIL becomes a bottleneck when multiple threads compete for execution.

Subprocess

Using the subprocess module, we create the Time Delta matrix within a separate Python process. After its creation, the matrix is stored locally on the disk and then read by the main QGIS Python process. This approach, however, is not

ideal due to the need for constant read and write operations to and from the disk. Additionally, access permissions can pose constraints, and as shown in Figure 7.2, the matrix files occupy a significant amount of space as we scale either the number of frames or the number of trajectories, creating substantial memory concerns.

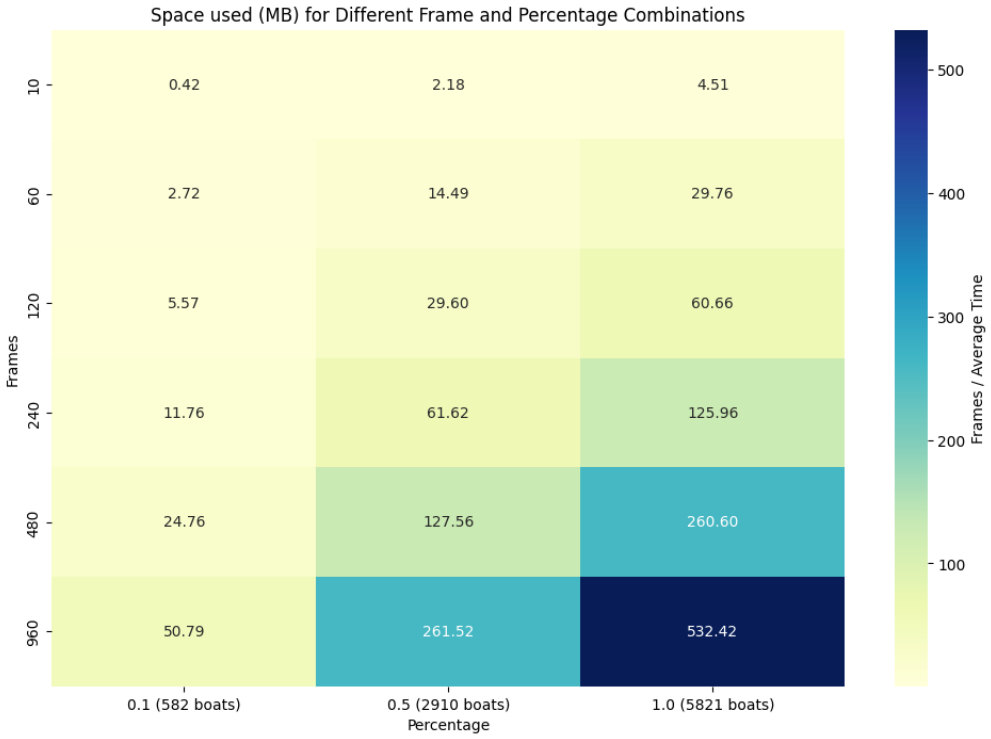


Figure 7.2: Matrix file sizes (in MB) for Resampling Mode with Subprocess library

It is important to note that the subprocess module can start processes in other languages as well. As future work, it is possible to envision a solution where the matrices are created in a more optimized language. However, this option was not considered within the scope of our research, as it would introduce additional requirements for the MOVE plugin.

Multiprocessing

Using the multiprocessing module, we can create new Python processes and even establish shared memory or queues, allowing direct passing of variables between processes. This enables a solution where a queue is used to pass the Time Delta matrix to the original QGIS Python process, eliminating issues related to local file read/write operations.

A significant advantage of multiprocessing is its ability to create true parallelism, allowing Python processes to run on multiple CPU cores (both logical and physical). Our task of creating a matrix containing the resampled positions for all trajectories can be easily divided into multiple sub-tasks. There are two possible ways to accomplish this subdivision:

- Dividing the task of creating one matrix among multiple workers, where each worker creates a submatrix with the positions for a subset of all the trajectories.
- Assigning the task of creating the next N upcoming matrices to N workers.

Both solutions have great potential to improve the frame rate of the animation. However, to enable faster switching between forward and backward animation, the first idea is easier to implement. From a purely non interactive animation standpoint, the second solution can definitely become more interesting as it would remove the pauses that can be created due to our uninterrupted animation FPS heuristic that is based off of the previous QGIS Task time. By having more multiple Time Deltas ready, there is more certainty that the animation won't pause. But the main reason why we pick the first idea is that using a multi-core solution in Python presents the same challenge as multithreading, we do not control which process will run on which CPU. If a worker process runs on the same CPU as the QGIS animation process, we will still experience animation slowdowns.

A potential solution involves using the multiprocessing library's ability to set CPU affinity, designating specific CPUs for particular processes. This feature, however, only works on Linux and Windows, as macOS does not support such operations. Even on Linux and Windows, the affinity parameter is more of a suggestion and does not guarantee that the QGIS animation will remain unaffected. Given these constraints, we decided not to implement any solution that relies on setting affinities. Nevertheless, we accept some degree of animation slowdown if it allows us to visualize a large number of trajectories at a reasonable frame rate.

7.2.2 Benchmarks

We implement this idea using the base code developed for the TimeDeltas Mode. In this section we provide the results of the multiple tests we conducted for both single-core and multi-core versions of the Resampling Mode. We measure the frame rate in terms of FPS and for its stability, the memory usage in MB throughout the animation and the overall user experience. In the Resampling Mode, choosing the correct T_Delta_Size(see Section 5.2.5) is just as important as in the TimeDeltas Mode, and the same considerations apply here, we redirect you to the Section 5.2.5 to read more on this.

We provided the low-end laptop configuration test results for the TimeDeltas Mode in Chapter 5 because we wanted to display the impact of the computationally intensive nature of Time Deltas architecture and the limitations of Python multithreading. To test the Resampling Mode, our aim is to provide the upper limit in terms of achievable performances for this architecture, as such we will use the high end Desktop configuration, we will also provide the Desktop results for both Interactive and TimeDeltas mode for comparison's sake.

Average FPS

In Tables 7.1 and 7.2 we display the average FPS values for all the different vector layer Modes we discussed up to this points : Interactive, TimeDeltas, single and multi-core Resampling Modes. The results are obtained by averaging the FPS values from the first 120 frames of the Danish AIS and STIB datasets (with 1 minute time granularity for both). From these results we can immediately see the impact of the multi-core solution, achieving the QGIS's FPS cap of 100 FPS for 1000 individual trajectories and almost reaching the 60 FPS for the **entirety** of the Danish AIS boats dataset.

Solutions	**AIS 1000 boats**	**AIS 5821 boats**
Interactive Mode (from Section 5.1)	58.80	12.43
TimeDeltas Mode (from Section 5.2)	60.80	14.54
Resampling Mode single-core	78.92	16.53
Resampling Mode multi-core	98.84	53.91

Table 7.1: Danish AIS dataset : average FPS results for all vector layer modes

Solutions	**STIB 100000**	**STIB 497609**
Interactive Mode (from Section 5.1)	2.03	Not Enough RAM
TimeDeltas Mode (from Section 5.2)	7.65	1.92
Resampling Mode single-core	10.50	1.99
Resampling Mode multi-core	13.72	2.43

Table 7.2: Stib dataset : average FPS results for all vector layer modes

Frame rate and stability

To delve deeper into the analysis of the frame rate, we display in Figure 7.3 the FPS values for the first 120 frames of the Danisha AIS and STIB dataset animation (with a 1 minute time granularity). We immediately notice the huge jump in performance brought by the multi-core resampling solution, with the occasional dips created by processes sharing the same CPU as the main QGIS animation process. We can also observe a notable improvement by the Resampling Mode single-core Mode over the TimeDeltas Mode, which gives a more stable frame rate throughout the animation on top of having a better average FPS. To push the test

even further, we display in Figure 7.3 the frame rate stability for the entire STIB dataset, which represents 497609 individual trajectories. This shows that even with the multi-core resampling solution we are still bounded by the FPS upper bound of on_new_frame (see Section 5.11).

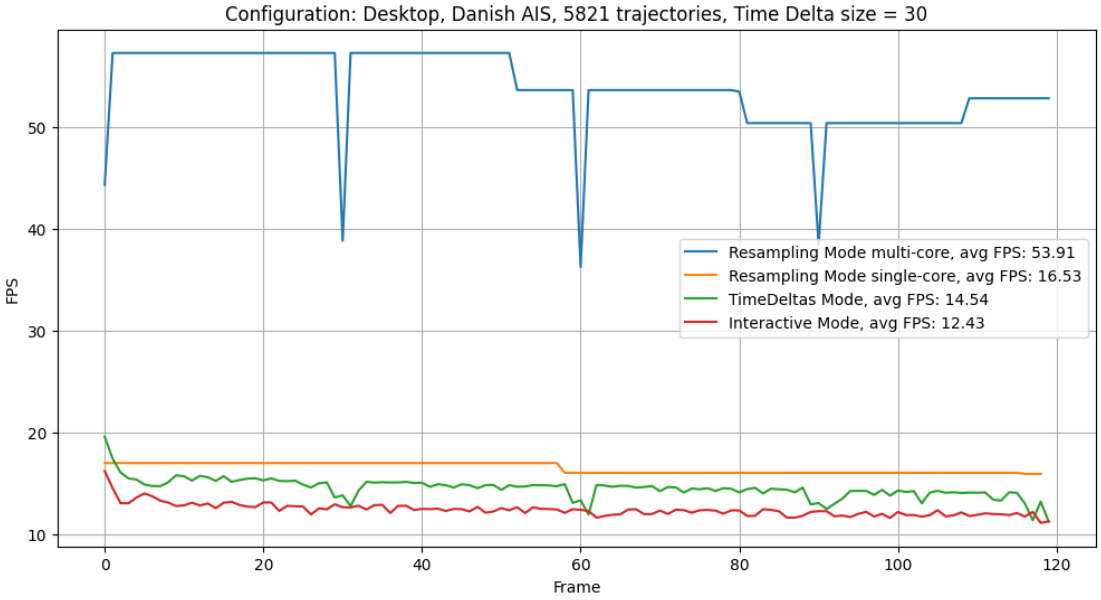


Figure 7.3: Danish AIS dataset : frame rate stability of all vector layer modes

Figure 7.4: Stib dataset : frame rate stability of all vector layer modes

Memory Usage

To create a direct comparison with the Interactive and TimeDeltas Modes results, we use the following setup : T_Delta_Size of 30 frames, laptop hardware configuration, two subsets of the Danish AIS and STIB datasets. The animation is run with only the vector layer in the QGIS process, and the measurements are taken when the buffer is full. We display the results in Table 7.3, we can clearly see the improvement in memory usage with regards to the Danish AIS dataset, this shows that storing positions in a numpy matrix is more efficient than storing the subset of trajectories themselves. However, in the case of the STIB dataset, this improvement is minimal since it contains a huge number of trajectories, and even when a trajectory does not appear in the Time Delta, the matrix still has to keep an empty row, increasing the memory usage.

Dataset	**Interactive Mode**	**TimeDeltas Mode**	**Resampling Mode single-core**
Danish AIS : 582 trajectories	0.59	0.58	0.33
Danish AIS : 5821 trajectories	1.80	1.80	0.39
STIB : 100000 trajectories	8.00	1.20	1.20
STIB : 497609 trajectories	Not enough RAM	4.50	4.10

Table 7.3: Resampling Mode single-core results : memory usage (in GB)

7.2.3 Final Remarks

Overall, even with only a single-core implementation, we improve upon the original TimeDeltas Mode from Chapter 5, with an animation that uses slightly less memory, and provide better average FPS values. When looking at the multi-core solution, it absolutely eclipses the previous solutions in terms of raw capabilities, however it reintroduces the slowdowns it set out to resolve with the Multiprocessing library. But even with the slowdowns, the improvement in frame rates are such that users can set a lower FPS cap to have a smooth animation and still enjoy higher FPS. During our research, we identified several promising ways to enhance the previous ideas by removing one or more of the initial constraints set for MOVE. These potential improvements are detailed below.

Focusing on One Animation Direction

In all previous solutions, we had to consider the possibility that the user might change the direction of the animation from forward to backward(or vice versa) at any time. In this section, we present a concept developed by Maxime Schoemans.

By restricting the animation to a single direction, we rework the value_at_timestamp method of MEOS to operate in O(1) time complexity instead of O(log n). As explained in section 3.2.3, a spatiotemporal trajectory in MobilityDB and MEOS is a sequence of (position, time) instants. When value_at_timestamp is called to fetch the position at a given timestamp, it first performs a binary search on the instants' timestamps to find the two bounding instants, and then interpolates to get the final position value. However, by considering an approach with only one moving direction (forward or backward), we can create an attribute that stores the last instant of the trajectory accessed in the animation. This allows the next value_at_timestamp call to skip the binary search and directly access the bounds for interpolation.

We can actually combine this "Iterative value_at_timestamp" with the existing value_at_timestamp whenever the user does change the direction. This iterative search method can be combined with resampling to completely eliminate the interpolation computation time as well. Both of these ideas show potential for improving performance. However, we did not implement a proof-of-concept for either solution in our research experiments, which presents an opportunity for future work.

Limited duration animation

Figure 7.5 displays the frame rate of the multi-core Resampling Mode, in blue the frame rate and in orange the upper bound for FPS value calculated by on_new_frame. We have purposefully selected the Laptop hardware configuration, since the CPU is less powerful, and with only 8 (logical and physical) cores, it exacerbates the issue we want to point out, which is that the background matrix generation QGIS Task is very slow, heavily capping the potential FPS in order to not break the animation. The size of a Time Delta is set to 30 in this example, we can also see the dips created by the background processes at every multiple of 30.

If we let go of the unlimited duration animation constraint, we can propose an alternative in which we create a huge Numpy matrix for a short duration animation. This design heavily restricts the scale of the animation as the matrix size is directly correlated to the number of frames and the number of individual trajectories. We estimate the potential size of the Numpy matrix with Equation 7.2.3, as an example, let's take the Danish AIS boat dataset of 5821 individual trajectories, if we cap the animation to 30 FPS, a 3 minutes animation will take : $(5821 \times (30 \times 60 \times 3) \times 21 = 629.5$ MB). This heavy cost in memory space in return gives us the best and most stable frame rate even for very large amounts of individual trajectories, giving users a really clean animation.

$$\text{Matrix Size} = \text{Rows} \times \text{Frames} \times \text{Bytes per cell}$$

However, there is also an upper bound limit in the performance of on_new_frame, if we run the entire STIB dataset with the multi-core Resampling Mode, Figure 7.6 shows that if the number of geometries to update on each frame becomes extremely

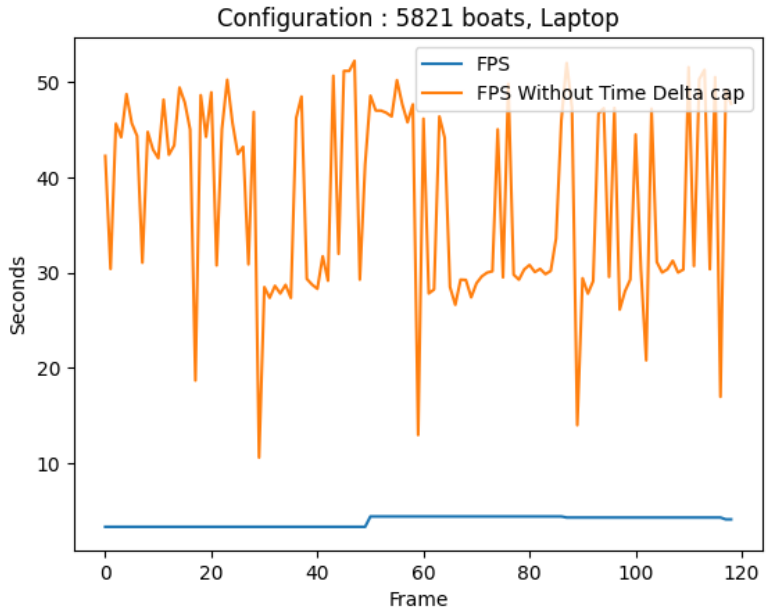


Figure 7.5: Frame rate for Danish AIS dataset animation in Resampling Mode on laptop configuration

large (497609 individual trajectories in this case), on_new_frame becomes the bottleneck.

A successful implementation must have a system to ensure that geometries which remain unchanged from one frame to another are not updated. Numpy is extremely useful for this purpose, as it allows us to exclude rows containing empty geometry WKBs in both columns t and $t-1$ very efficiently. Additionally, using a sparse matrix can reduce memory usage. These optimizations will mitigate some of the performance issues, although they will not completely eliminate them, as we show in Figure 5.11, the maximum FPS value is bounded by a logarithmic relation with the number of geometries to update in on_new_frame.

7.3 Summary of Resampling Trajectories

In this chapter we detail a new solution built entirely on the Time Deltas design architecture from the Chapter 5. By using MobilityDB's tsample operation that allows the restructuring of trajectories and Python's built-in multiprocessing library, we successfully bypassed the limitations set by the Python's multithreading and the time cost of the Iterative value_at_timestamp. Multiprocessing doesn't just allow running new process within the existing Python process, it also enables the use of all the computer's CPU cores. This capability enabled us to build a multi-core version that divides the task of building the matrix containing the positions for a Time Delta to multiple workers. This eliminated the FPS cap put in place for uninterrupted animation, bringing us closer to the real maximum FPS value calculated by the time generation of a single frame of the animation.

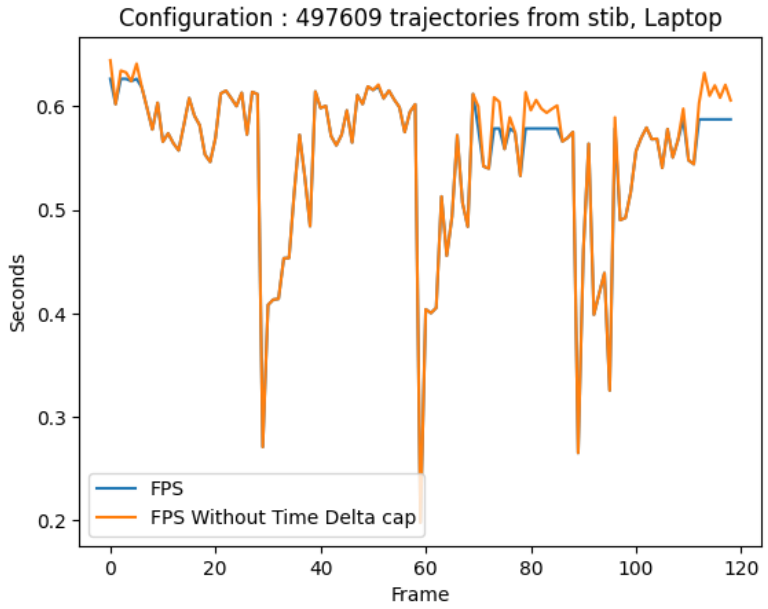


Figure 7.6: Frame rate for STIB dataset animation in Resampling Mode on laptop configuration

In the results we show how both the single-core and multi-core versions of this Resampling Mode beat the previous two solutions from Chapter 5, with the multi-core solution almost enabling a constant 60 FPS with the entirety of the Danish AIS dataset. The downside of this extreme uptick in performance is the usage of the entire CPU during the animation, meaning that this solution can be vulnerable to external loads hogging the CPU cores or even creating FPS dips when creating the matrix in the background if a worker process runs on the same CPU as the QGIS animation process. Even with the amazing performances shown in this chapter, we did not include the Resampling Mode in the final release. Our reasons for this decision are explained in the next and final chapter, which also covers the details of the new MOVE plugin and potential future improvements.

Chapter 8

Conclusions

To conclude this thesis report, we detail the state of the final beta release of the upgraded MOVE plugin. Throughout the thesis report, we discussed various design ideas and provided benchmarks from the actual implementations using proof-of-concept (POC) codes (these POCs are available on the GitHub repository of the research [1]). In this chapter, we start by covering the actual implementation of these ideas in the final deliverable that MobilityDB users will have access to at the time of writing this thesis. We then discuss the possible future improvements and end with the final conclusion.

8.1 MOVE plugin upgrades

We covered MOVE in Section 2.3 of the state of the art. This is an elegant plugin that enables users to write an SQL query to visualize both traditional PostGIS spatial columns and MobilityDB's spatiotemporal columns by creating materialized views. To develop our final solution, we will use the code of MOVE developed by Maxime Schoemans [20] as our base. It implements a well-optimized structure for a QGIS plugin and already includes features that will be essential to the final product, namely the code for managing the user's SQL queries and traditional PostGIS columns. We replace the materialized views with the in-memory QGIS vector layers.

8.1.1 Design Choices

We have mentioned multiple ideas that enable the visualization of MobilityDB's spatiotemporal types inside QGIS, each with its pros and cons. Ideally, we would rework QGIS's geometry types to support MobilityDB's temporal types by default, but this is not feasible within the scope of our work. Therefore, we have to rely on the tools QGIS provides like PyQGIS. When developing a plugin for QGIS's temporal controller, we must consider the philosophy behind the tool itself. The temporal controller expects user interaction with its interface, meaning that any solution that requires disabling buttons during an animation would not be ideal. Similarly, solutions that work entirely only with incremental time evolution or a specific direction (forward/backward in time) face challenges since users should be able to move the time slider freely. Additionally, we aim to provide the best user experience, which means avoiding solutions requiring additional setup (such as a local vector tiles server) or long loading times, that would create frustration

[1] *https://github.com/amanzer/mobilitydb-qgis-visualization*

points The plugin should be self-contained, allowing users to visualize trajectories immediately after downloading it.

By taking in all these considerations, we narrowed down our choice to the Interactive Mode presented in Section 5.1 This solution stores the entire spatiotemporal trajectories in memory, this imposes a clear performance upper bound in terms of the size of the dataset that can be viewed. But by storing all trajectories at the start, two major issues are addressed. Firstly, users do not need to wait for database interactions afterward; all operations on trajectories are done through the PyMEOS library in Python. Secondly, users can interact seamlessly with the temporal controller functionalities without needing to reload data for new configurations. So the limitations are acceptable because this solution enables a highly interactive experience, fitting right in with the Temporal Controller's philosophy.

Time Deltas architecture

Here are some final remarks regarding the solutions we developed using the Time Deltas architecture (TimeDeltas Mode from Section 5.2), including both the single-core and multi-core versions of the Resampling Mode from Chapter 7. These methods facilitate the visualization of large datasets with a very small memory footprint, enabling visualization on hardware with limited memory capacity. However, they introduce latency when modifying temporal configurations, thereby diminishing the interactive capabilities of the Temporal Controller.

Our decision to exclude these solutions from the final plugin release was primarily influenced by the challenges encountered in developing a production-ready tool for public use. Although the proof-of-concept implementations performed adequately in a development environment, building a robust and maintainable code structure that effectively handles QGIS task threads, manages the creation of new processes via Multiprocessing, and accommodates the unpredictable and sometimes erratic signals from the Temporal Controller proved to be both complex and time-consuming.

8.1.2 Graphical User Interface (GUI) of MOVE

We keep the base user interface of MOVE, it provides an area for users to write their SQL queries, a database selector drop down menu, and we added an automatic FPS checkbox, which, if selected, automatically selects the best FPS for the animation, alongside an FPS cap selector. After writing the SQL query, users can click the run button, and MOVE builds the layers accordingly:

1. **PostGIS spatial geometry column**: Creates a default vector layer with all SQL query attributes.
2. **MobilityDB spatiotemporal column**: Creates an in-memory vector layer with all SQL query attributes.

As in-memory vector layers in QGIS do not retain their data after closing the QGIS app, users will have to wait for the spatiotemporal data to be loaded on every startup of QGIS. Figure 8.1 displays the final interface of MOVE.

8.2 Future Improvements

There is a lot of work that can be done to improve the current state of the final release we deliver at the time of writing this report. The obvious improvements are the inclusion of the Resampling Modes to enable the visualization of extremely large datasets and better animation frame rate. We also need to incorporate a strategy to store the MobilityDB layer's data on disk so that users can keep the same layer through multiple runs of QGIS (currently, a new layer has to be created when QGIS is closed due to the in-memory nature of the vector layer). Another aspect not covered in the thesis report is the inclusion of the rest of MobilityDB's temporal types, like tfloat and tint. These temporal types can serve as temporal attributes. For example, in the case of the Danish AIS dataset, there could be a column for SOG (Speed Over Ground) that describes the speed of each boat at any time during its trajectory.

By adding the support for the rest of the temporal types as attributes alongside the spatiotemporal types, users will be able to change the visual aspect of the data points depending on the temporal value of their attributes. Figure 8.1 shows an example run of the Danish AIS dataset where the size of the points representing the boats is directly correlated to their ongoing SOG.

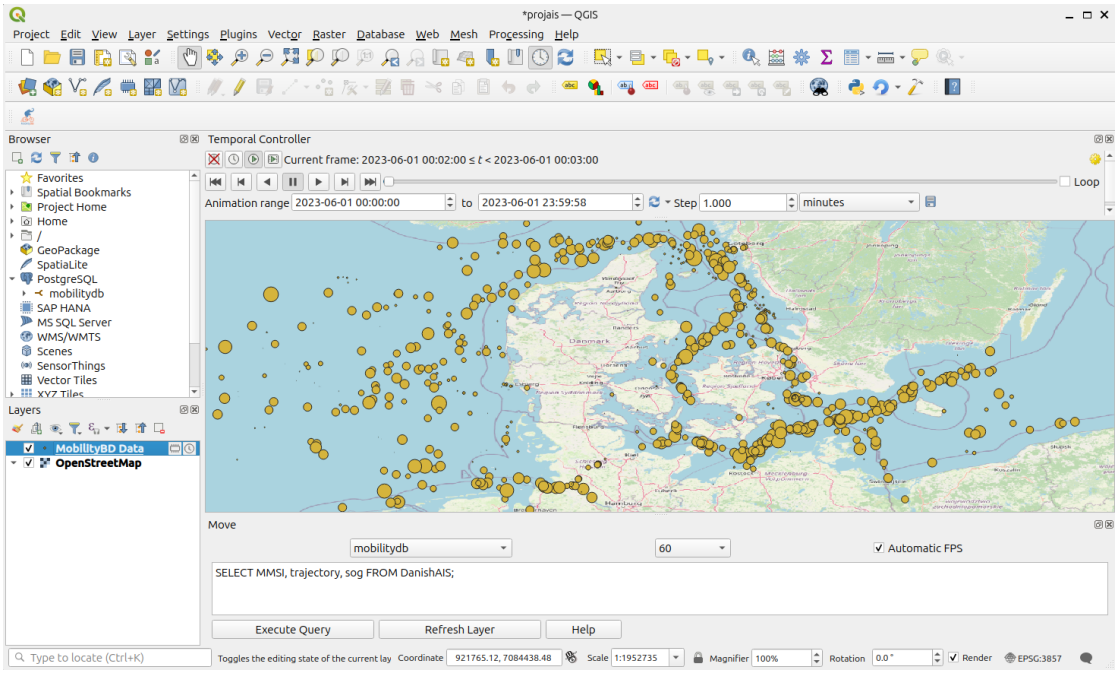


Figure 8.1: Animating boat size based on SOG

The Temporal Controller contains three distinct modes as of version 3.36 of QGIS, our entire focus throughout this research was on the Animated mode. Future work can easily implement all of its functionalities:

1. **FixedRange mode:** MOVE creates a separate linestring in-memory vector layer, at each temporal extents change signal, we update the visible

linestrings by taking the sections of all trajectorites contained in the new temporal extents.

2. **Cumulative Range:** This is an option alongside the FPS button in the Temporal Controller, it allows for the visualization to show not just the position at the selected timestamp but rather the entire trajectory from the initial to the selected timestamp. Incorporating this would be simply reusing the linestring in-memory vector layer of the FixedRange mode.

3. **Movie mode:** This does not take into account the temporal properties of the data, but is rather used to build custom animation with `@frame_number` expressions. We should disable the geometries update on frame update when this mode is selected.

And of course when it comes to future work, we can mention multiple areas of research that can still be done to develop new ideas and incorporate them into MOVE. We have mentioned our failed experiment with the vector tile layer, which could be an area of research that provides great results if we can bypass the limitations mentioned. We can also mention everything we discussed in Section 7.2.3, from the iterative `value_at_timestamp` to the limited duration animations.

8.3 Conclusion

This thesis has delved into the visualization and animation of MobilityDB's spatiotemporal objects within QGIS, a leading open-source Geographic Information System (GIS) platform. We have provided a thorough exploration of the essential tools and concepts in this domain, including a brief introduction to QGIS, MobilityDB and its underlying engine, MEOS, as well as a detailed examination of the Python libraries PyQGIS and PyMEOS, which are instrumental in the development of the new MOVE plugin.

Throughout this research, we have introduced several innovative solutions for visualizing MobilityDB trajectories within QGIS. Through the Interactive, TimeDeltas, and Resampling Modes, our work enables QGIS users to interact with their spatiotemporal data using the Temporal Controller. By addressing key challenges such as frame rate, memory usage, and data scalability, this thesis lays a solid foundation for future advancements in this field. The solutions proposed here not only advance the capabilities of QGIS but also present novel ideas that could be incorporated into other GIS systems.

Looking ahead, there are numerous opportunities for further research. One promising direction is the optimization of these solutions to better handle even larger datasets and more complex queries that include temporal attributes. Additionally, exploring new visualization techniques could further enhance the MOVE plugin's functionality. In summary, this thesis has made significant contributions to the field of spatiotemporal data visualization within QGIS, offering both practical tools and theoretical insights that will support future research and development.

Bibliography

[1] Iliass El Achouchi, "Mobility Data Exchange Standards in MobilityDB", MA thesis, Université Libre de Bruxelles, Aug. 2023.

[2] Vladimir Agafonkin, *A Dive into Spatial Search Algorithms: Searching through millions of points in an instant*, 2017.

[3] Abhinav Ajitsaria, "What Is the Python Global Interpreter Lock (GIL)?", in: (2018).

[4] Florian Baudry, "Visualizing Moving Objects using MobilityDB, Leaflet, React and pg tileserv", MA thesis, Université Libre de Bruxelles, 2023.

[5] Adam Broniewski et al., "Using MobilityDB and Grafana for Aviation Trajectory Analysis", in: vol. 28, Jan. 2023, p. 17.

[6] Ludéric Van Calck, *Visualizing MobilityDB Data using QGIS*, INFO-H402: Computing Project, 2021.

[7] K.T. Chang, *Introduction to Geographic Information Systems*, McGraw-Hill higher education, McGraw-Hill, 2008, ISBN: 9780073312798.

[8] M.N. DeMers, *Fundamentals of Geographic Information Systems*, Wiley, 2008, ISBN: 9780470129067.

[9] Thomas Dudziak, "MEOS.NET : Manipulating Mobility Data in C#", Master's thesis, Brussels, Belgium: Université Libre de Bruxelles, 2024.

[10] Soufian El Bakkali Tamara, "Visualization of Mobility Data on Openlayers", MA thesis, Université Libre de Bruxelles, 2023.

[11] S.R. Galati, *Geographic Information Systems Demystified*, Artech House communications library, Artech House, 2006, ISBN: 9781580535335.

[12] Juan Godfrid et al., "Analyzing public transport in the city of Buenos Aires with MobilityDB", in: *Public Transport* 14.2 (June 2022), pp. 287–321, ISSN: 1613-7159.

[13] Anita Graser, *Detecting Close Encounters using MobilityDB 1.0*, 2022.

[14] Anita Graser, *Visualizing Trajectories with QGIS and MobilityDB*, 2022.

[15] R.H. Güting and M. Schneider, *Moving Objects Databases*, The Morgan Kaufmann Series in Data Management Systems, Elsevier Science, 2005, ISBN: 9780080470757.

[16] Charles R. Harris et al., "Array programming with NumPy", in: *Nature* 585.7825 (Sept. 2020), pp. 357–362.

[17] John R. Herring, *OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 1: Common architecture*, OpenGIS Implementation Specification OGC 06-103r3, version 1.2.0, Open Geospatial Consortium Inc., Oct. 2006.

[18] Volker Janssen, "Understanding coordinate reference systems, datums and transformations", in: *International Journal of Geoinformatics* 5 (Jan. 2009).

[19] Nidhal Mareghni, "JMEOS: A Java binding of the MEOS spatiotemporal library", Master's thesis, Brussels, Belgium: Université Libre de Bruxelles, 2024.

[20] Maxime Schoemans, Mahmoud Attia Sakr, and Esteban Zimányi, "MOVE: Interactive Visual Exploration of Moving Objects", in: *Proceedings of the Workshops of the EDBT/ICDT 2022 Joint Conference, Edinburgh, UK, March 29, 2022*, ed. by Maya Ramanath and Themis Palpanas, vol. 3135, CEUR Workshop Proceedings, CEUR-WS.org, 2022.

[21] Shashi Shekhar and Sanjay Chawla, *Spatial Databases: A Tour*, Upper Saddle River, NJ: Prentice Hall, 2003, chap. 4.

[22] Fabrıcio Ferreira da Silva, "Visual Analytics for Moving Objects Databases", MA thesis, tu-berlin, 2021.

[23] Société des Transports Intercommunaux de Bruxelles (STIB) / Maatschappij voor het Intercommunaal Vervoer te Brussel (MIVB), *STIB-MIVB Open Data*, Accessed: 2024-08-06, 2024.

[24] Underdark, *QGIS 3.14 Pi Release Notes*, June 2020.

[25] MobilityDB Workshop, *Chapter 1. Managing Ship Trajectories (AIS)*, Accessed: 2024-07-28, 2024.

[26] Esteban Zimányi, Mariana MG Duarte, and Víctor Diví, "MEOS: An Open Source Library for Mobility Data Management.", in: *EDBT*, 2024, pp. 810–813.

[27] Esteban Zimányi, Mahmoud Sakr, and Arthur Lesuisse, "MobilityDB: A Mobility Database Based on PostgreSQL and PostGIS", in: *ACM Trans. Database Syst.* 45.4 (Dec. 2020).

[28] Esteban Zimányi et al., "MobilityDB: hands on tutorial on managing and visualizing geospatial trajectories in SQL", in: *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on APIs and Libraries for Geospatial Data Science*, 2021, pp. 1–2.

[29] Esteban Zimányi et al., "MobilityDB: A Mainstream Moving Object Database System", in: Aug. 2019, pp. 206–209, ISBN: 978-1-4503-6280-1.