Université Libre de Bruxelles
Faculty of Sciences
Computer Sciences department

Visualizing Moving Objects using MobilityDB, Leaflet, React and pg_tileserv

Florian Baudry

Promotor: Prof. Esteban Zimányi

Acknowledgement

I would like to express my deepest appreciation to all those who provided me the possibility to complete this master's thesis.

First and foremost, I would like to thank my parents. Their unwavering support and sacrifices have given me the ability to pursue my studies and reach this pivotal moment in my life. Without their love and encouragement, this journey would have been significantly more difficult.

I am deeply grateful to my friends, so sure of me when I am not even sure of myself. Their constant support and reassurances were invaluable in helping me overcome numerous obstacles along the way.

I am particularly thankful for Syti.io, who generously provided me with their synthetic dataset, `Persona`. This dataset played a vital role in the testing phase of my implementations, and their contribution significantly elevated the quality of my work.

I would also like to express my profound gratitude to my supervisor, whose guidance and expertise were indispensable throughout this process. His insights and dedication to my project have made a lasting impact on my academic and professional career.

Lastly, a special thanks to my girlfriend for her patience, love, and understanding during the difficult moments of this journey. Your unwavering support has been a source of strength that I cannot thank you enough for.

Contents

List of Figures

List of Tables

Abstract

This master thesis delves into the visual exploration and analysis of spatiotemporal data, specifically focusing on the dynamic visualization of moving objects using MobilityDB and Leaflet. In our increasingly interconnected and data-driven world, tracking and understanding the movement of objects in space and time is paramount in various domains such as transportation, logistics, wildlife monitoring, and urban planning. While numerous tools and methods have been developed to manage and analyze spatiotemporal data, visualizing this data type remains a challenging task due to its complexity and the need for interactive, real-time displays.

Our work begins with a comprehensive state-of-the-art survey, providing an overview of existing spatiotemporal data visualization techniques and tools. We then provide an in-depth review of several key technologies, including MobilityDB for managing spatiotemporal data and Leaflet for map-based visualization.

Building on this foundation, we design and implement a web-based visualization tool capable of rendering animations of moving objects over time. The developed tool leverages the capabilities of MobilityDB and Leaflet, offering an interactive interface for users to explore and understand their moving object data. Specifically, two different implementations, namely MVT and GeoJSON, are tested on two distinct datasets, demonstrating the flexibility and adaptability of the tool.

Lastly, we carry out an extensive evaluation of the tool, showcasing its utility in handling real-world and synthetic spatiotemporal datasets and providing a comparative analysis of the performance of MVT and GeoJSON implementations. Suggestions for future work include the exploration of other visualization techniques and the integration with other mapping libraries, demonstrating the potential for further improvements and broader applicability in the field of spatiotemporal data visualization.

Chapter 1

Introduction

1.1 Background

The advent of technologies such as Global Positioning System (GPS) and Internet of Things (IoT) has resulted in an explosion of spatiotemporal data, which represents the movement of objects over time. This data has a wide range of applications, from transportation and logistics to wildlife tracking and disaster management. As the volume of this data increases, so does the need for effective tools to store, manage, and visualize it.

Spatiotemporal databases, like MobilityDB [1], an extension of PostgreSQL and PostGIS, offer robust data storage and querying capabilities for handling moving object data. Although advancements in spatiotemporal data visualization have been made, limitations in interactivity, user-friendliness, and handling of large, complex datasets still exist. This thesis targets these specific areas for improvement.

Interactive maps are a common tool for visualizing spatial data, and libraries such as Leaflet have made it relatively easy to create custom maps for web applications. Yet, when it comes to visualizing moving objects, these libraries often fall short, requiring developers to create custom solutions.

This thesis aims to bridge this gap by developing a visualization tool that integrates MobilityDB with Leaflet, served by pg_tileserv and built on a React application, to enable the efficient visualization of moving objects. Two implementations, one based on MVT and another on GeoJSON, are developed and compared to provide a comprehensive understanding of the system's performance under various conditions.

This project distinguishes itself through a commitment to open-source technologies. The cornerstone technologies used in this thesis – MobilityDB for the database layer, PostGIS for spatial data processing, and Leaflet for interactive mapping – are all open-source. Furthermore, the visualization tool developed as a part of this thesis will also be open-source.

The choice of open-source technologies not only aligns with principles of transparency, accessibility, and collaboration, but it also encourages reproducibility and continued community-driven development. By contributing to the open-source ecosystem, this project aims to provide valuable tools and insights to developers, researchers, and organizations working with moving object data.

1.2 Problem Statement

While there has been substantial progress in the storage and analysis of spatiotemporal data, its efficient visualization, especially concerning moving objects, remains a challenging problem. Visualizing the movement of objects over time on a map is essential for understanding and interpreting the data, but existing solutions often lack the flexibility and efficiency required for large datasets.

In particular, there are two main challenges that this thesis addresses:

1. **Efficient data representation:** The representation of moving object data in a format that allows for efficient transfer and rendering on a web application is a key issue. We explore the use of two data formats, MVT and GeoJSON, each with their advantages and trade-offs, to address this challenge.

2. **Scalability:** As the number of moving objects increases, the performance of the visualization tool can degrade. We aim to investigate the performance of our tool under varying conditions, including different numbers of moving objects (100, 250, 500, 1000, 2000, 5000, 10000) and datasets, and explore techniques to improve scalability.

By addressing these challenges, this thesis seeks to contribute to the field by developing a visualization tool that can efficiently display moving objects on a map for various applications.

1.3 Objectives

The primary objective of this thesis is to design and implement a visualization tool for efficiently displaying moving objects stored in a MobilityDB database on a map, enhanced with a Least Frequently Used (LFU) caching system for optimized tile loading. This tool aims to provide an intuitive interface for users to track the movements of different objects interactively.

In order to achieve this primary objective, the following specific objectives have been identified:

1. **Design and Implement the Visualization Tool:** Design a system architecture that integrates MobilityDB, pg_tileserv, Leaflet, and React. Implement two versions of the tool, one using MVT and the other using GeoJSON as input data. Develop an LFU caching system to optimize the loading of map tiles.

2. **Caching Strategy Evaluation:** Evaluate the performance and effectiveness of the LFU caching strategy. Assess the improvement in loading times and efficiency in managing large amounts of data.

3. **Performance Evaluation:** Conduct a comprehensive performance evaluation of the two implementations under different conditions, including varying numbers of moving objects, different datasets, and the impact of the caching system.

4. **Compare and Analyze the Implementations:** Analyze and compare the performance of the MVT and GeoJSON implementations with and without the LFU caching system. Identify the trade-offs, advantages, and disadvantages of each implementation and caching's impact.

By accomplishing these objectives, this thesis aims to make a significant contribution to the field of spatiotemporal data visualization and offer a solution for various applications that require the visualization of moving objects with improved efficiency.

1.4 Limitations

Despite the careful design and implementation of the study, there are some limitations:

1. **Data limitations:** The study is limited to the two datasets provided. While efforts were made to ensure these datasets represent a range of conditions, they may not cover all possible scenarios.

2. **Scalability:** Although the tool is designed to handle a varying number of moving objects, extreme scales were not tested due to resource constraints. The maximum number of moving objects tested is 10000. Some tests have been made using larger number of moving objects but as you will see, the loading times are too high for a good user experience.

3. **Technology constraints:** The tool is implemented using specific technologies (MobilityDB, pg_tileserv, Leaflet, and React), and the findings may not be directly applicable to other technology stacks.

1.5 Significance and Applications

The effective visualization of moving object data has wide-ranging implications across numerous fields. For instance, in transportation and logistics, it can assist in tracking and optimizing vehicle routes for improved efficiency. In wildlife research, it enables scientists to study animal migration patterns and make informed conservation decisions. For disaster management, tracking the real-time movement of storms, wildfires, or other hazards can aid in timely response and recovery efforts.

The visualization tool developed in this thesis will, therefore, serve as a valuable resource for developers, researchers, and organizations in these and many other fields. By enhancing the capabilities of existing open-source technologies, it will foster further innovation and development in the broader community.

1.6 Key Technologies and Concepts

This thesis involves a range of technologies and concepts fundamental to the design and implementation of our visualization tool. MobilityDB, a spatiotemporal database extension of PostgreSQL and PostGIS, serves as the storage and management

system for moving object data. Leaflet, a leading open-source JavaScript library for mobile-friendly interactive maps, is used to create the interactive map interface. MVT and GeoJSON, two key formats for representing and transmitting spatial data, are employed and compared in this study. Additionally, a LFU caching system is introduced to optimize tile loading in the application. Each of these elements is critical to the system's performance and will be explored in detail in the subsequent chapters.

1.7 Thesis Overview

The rest of this thesis is organized as follows:

1. **Chapter 1 - Introduction:** Provides an overview of the thesis, including the background, problem statement, objectives, and scope of the study.
2. **Chapter 2 - State of the Art:** Presents an in-depth review of the latest advancements and trends in the field of spatiotemporal data management and visualization.
3. **Chapter 3 - Tool Review:** Discusses existing tools and technologies that are used in the visualization and management of spatiotemporal data.
4. **Chapter 4 - Project Description:** Describes the project in detail, elaborating on its aim, design, technologies used, and overall project plan.
5. **Chapter 5 - Implementation:** Details the design, architecture, and implementation of the visualization tool using MVT and GeoJSON.
6. **Chapter 6 - Testing and Evaluation:** Describes the testing methodology, datasets used, and presents a comprehensive evaluation of the tool's performance.
7. **Chapter 7 - Discussion and Future Work:** Provides an analysis of the results, discussing insights, limitations, potential improvements, and outlining directions for future research.
8. **Chapter 8 - Conclusion:** Summarizes the key findings and contributions of the thesis, and reflects on the overall project.

Chapter 2

State of the Art

2.1 Introduction

The discipline of spatio-temporal data visualization has its roots in the pioneering work of Charles Joseph Minard. His iconic depiction of Napoleon's ill-fated Russian campaign of 1812 (see Figure 2.1) remains a testament to the power of effective visual storytelling in the context of geo-spatial information [2]. Embodying Tufte's principle that 'Graphical elegance is often found in simplicity of design and complexity of data' [3], Minard's work underscores the critical role that visualization plays in turning raw data into insightful and actionable information.

Building on such foundational works, more recent developments have leveraged technological advancements to provide ever more sophisticated tools for the visualization of movement data. For instance, Tatem et al. [4] examined the role of expanding global transport networks in the spread of pathogens, indicating the necessity for robust visualization tools in tracking these complex movements and patterns. Similarly, the tool developed by Xavier and Dodge [5] offers an exploratory framework to map relationships between animal movement and the environment.

Advances in the field have been further bolstered by the work of Andrienko et al. [6], who proposed a novel visual analytics approach. Their framework synergistically combines interactive visual displays with computational methods to handle and analyze large amounts of movement data. This represents a significant step forward in understanding movement behaviors and mobility patterns.

We will now navigates through the various techniques and methods used for visualizing spatio-temporal data, moving from traditional static methods such as Geographical Information Systems (GIS) to more dynamic and interactive visualization techniques. The wide array of applications and use-cases where spatio-temporal data visualization plays an indispensable role are also explored, traversing diverse fields like transportation, logistics, environmental monitoring, and urban planning.

2.2 Spatiotemporal Databases

Spatiotemporal databases are a type of database that is capable of storing, indexing, and querying both spatial and temporal data [7]. Spatial data refers to

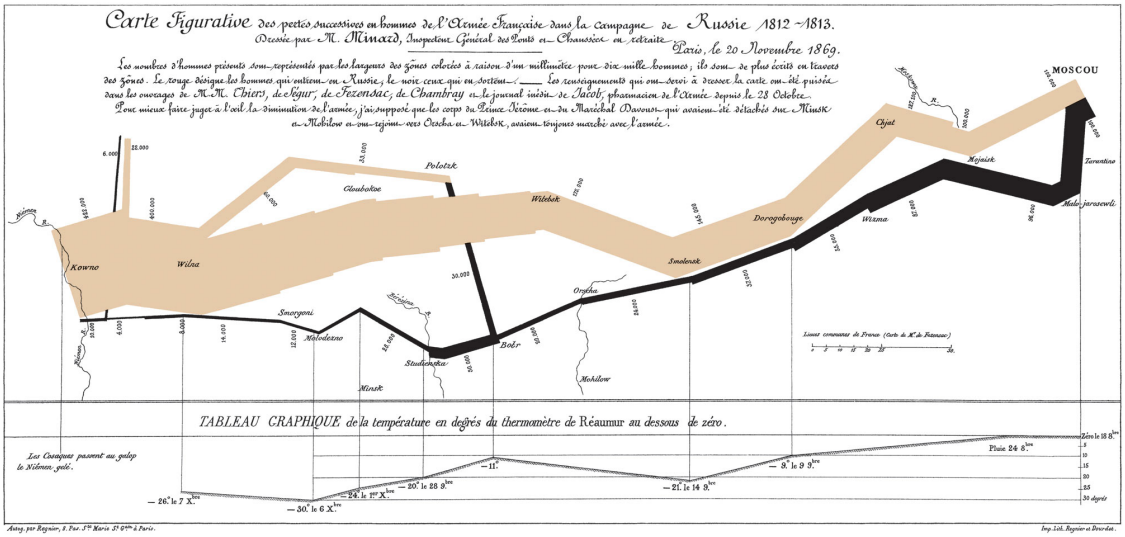


Figure 2.1: Minard, C. J. (1869) *Carte Figurative des pertes successives en hommes de l'Armée Française dans la campagne de Russie 1812–1813*, 'a portfolio of his work held by the Bilbiothèque de l'Ecole Nationale des Ponts et Chaussées, Paris'[3]

data associated with geographical or spatial information, such as coordinates or polygons, while temporal data refers to data associated with time.

The field of spatiotemporal databases has seen substantial growth over the past few decades due to the increase in availability of spatial and temporal data. These databases are used in a wide range of applications, from GIS and transportation systems to social media analysis and environmental monitoring [8].

One of the most popular spatial databases is PostGIS, an open-source software program that adds support for geographic objects to the PostgreSQL relational database [9]. PostGIS is used in combination with PostgreSQL to create a spatial database for storing and querying spatial data.

MobilityDB is a recent extension of PostgreSQL and PostGIS, which adds support for temporal and spatiotemporal objects, making it a powerful tool for managing moving object data [10]. MobilityDB introduces new data types and functions that facilitate the handling of trajectories of moving objects, which are typically represented as a sequence of time-stamped locations.

While these advancements have significantly improved the storage and querying capabilities for spatiotemporal data, visualization remains a challenge, particularly when it comes to moving objects. As the volume of moving object data continues to grow, the need for effective visualization tools becomes increasingly apparent, thus the motivation for this thesis.

2.3 Visualizing Spatiotemporal Data

Visualizing spatio-temporal data can be challenging due to the multiple dimensions involved. Despite this, several techniques have emerged, ranging from static methods such as GIS, to dynamic, non-interactive visualization methods, and interactive techniques [11].

2.3.1 GIS-based Static Methods

GIS offer a static method for visualizing spatio-temporal data. Using these systems, data can be represented as points, lines, or polygons, overlaid on maps. For instance, the trajectories of moving objects can be represented as lines on a map, where the thickness, color, or pattern of the line can be used to represent other dimensions of the data, such as speed or direction of movement.

The choice of color schemes in these visualizations plays a critical role in their effectiveness. As highlighted by Harrower and Brewer [12], appropriate color schemes can significantly enhance the readability and interpretability of the map. They developed ColorBrewer, an online tool that assists in selecting optimal color schemes for maps based on the nature of the data (e.g., sequential, diverging, or qualitative data) and the number of data classes. This tool helps to mitigate common issues in map color selection such as accommodating colorblind viewers and ensuring legibility on screen and in print. A well-chosen color scheme can not only make the visualization more aesthetically pleasing but also convey the underlying data more effectively, facilitating a deeper understanding of the spatio-temporal patterns.

2.3.2 Dynamic, Non-interactive Visualization Methods

Dynamic visualization methods provide a valuable way to depict the temporal dimension of spatio-temporal data. For instance, non-interactive animations can effectively showcase how data points change over time on a map [13]. However, the interpretation of such animations can be challenging due to rapid screen changes, which may lead to missed information if the animation speed is not properly adjusted.

Space-time cubes They offer an alternative by representing temporal data through three dimensions: two for geographical space and one for time [6]. Trajectories of moving objects are shown as lines or curves within this cube. However, due to its 3D nature, space-time cubes may suffer from occlusion problems, where some data points or trajectories are hidden behind others, making it difficult to fully perceive the data.

Stacked graphs They serve as an effective tool for handling high-density trajectory attribute data [14]. The thickness of each layer in the graph corresponds to the value of a data attribute at a given point in time, creating a landscape-like shape. Despite its strength in showing overall patterns, stacked graphs can sometimes be misleading in terms of relative comparisons, especially when layers are stacked on top of one another, distorting the perception of the lower layers.

Composite density maps They combine the densities of multiple variables into a single visual representation, can help identify correlations between different attributes [15]. However, the clarity of these maps can be compromised when dealing with high-dimensional data, making the interpretation of relationships between variables more complex.

Heatmaps They are beneficial for visualizing dense datasets, as they transform data points into a grid of colored cells to represent the intensity or frequency of occurrences [16]. Despite their usefulness in identifying patterns, heatmaps can be less effective when the dataset has outliers with extreme values, which can skew the color scale and make it harder to differentiate between the more common values.

2.3.3 Interactive Techniques

In the domain of spatiotemporal data visualization, interactive techniques play an integral role, particularly when it comes to managing the complexity inherent in the data. One of the primary uses of interactivity is the effective management of time as a filter, helping users to navigate the data in a more intuitive and user-friendly manner [17].

Animation can also serve as an interactive tool that allows users to visualize the evolution of spatiotemporal phenomena, as they can control aspects such as the speed or range of the animation. However, its usage should be carefully considered. Tversky et al. [18] found that animated graphics are not always superior to static ones for conveying complex systems, mainly due to their potential to overload the viewer with too much information at once. On the other hand, judicious use of interactivity can overcome these disadvantages and make animations more effective, especially for real-time reorientations in time and space.

Some advanced visualization techniques go beyond basic animation and interactivity to allow users to interact with complex visual elements such as spatiotemporal graphs and density maps [15] [19] . These interactive visualizations help users to understand and explore intricate patterns and relationships in the data.

A key aspect of interactivity in visualization is the ability to modify the visual representation to highlight different aspects of the data. This is often achieved by adjusting graphical parameters like color schemes [12], or by stacking graphic elements to avoid over-plotting [14] [20].

2.4 Use Cases of Spatiotemporal Data Visualization

Spatiotemporal data visualization has found valuable applications in numerous fields, particularly in relation to the movement of entities. These applications leverage various visualization techniques to uncover patterns and relationships in data, facilitating deeper understanding and informed decision-making. This section highlights three applications in the context of animal movement data, demonstrating the diversity and the potential of spatiotemporal data visualization methods.

Mapping Animal Movement and Environment Relationships An exploratory visualization tool was introduced by Xavier and Dodge to map relationships between animal movement and the environment [5]. The tool, developed in Processing 2.0 with OpenGL, provides a dynamic and interactive way to visualize the trajectories of animal movement in their geographic context, allowing the user to visually explore the interactions between animals and their environment. The tool

also enables the exploration of various factors influencing the movement, such as topography and vegetation cover, enhancing the understanding of animal behavior and habitat usage.

DynamoVis: Visualizing Movement in Relation to Factors Expanding on the previous work, Dodge, Toka, and Bae presented DynamoVis 1.0, an open-source software for creating custom animations and multivariate visualizations from movement data [13]. Written in Java, this software enables the visual exploration of patterns that capture the associations between animals' movement and internal and external factors affecting it. With its grounding on cartographic principles and intuitive visual forms, DynamoVis provides a useful tool for researchers, enhancing scientific discovery and communication of findings.

MoveApps: Analysis Platform for Animal Tracking Data In a different approach, Kölzsch and colleagues developed MoveApps, a serverless no-code analysis platform designed to handle the increasing volume and complexity of animal tracking data [21]. MoveApps allows users, even those without coding skills, to apply a wide variety of analysis methods on their data. The platform's interface uses visual elements to represent data, enabling users to visually inspect the patterns and relationships in the movement data. Moreover, MoveApps supports interactivity, allowing users to manipulate the visualization and extract more detailed insights.

2.5 Web-based Visualization of Spatiotemporal Data

The proliferation of web technologies has greatly impacted the field of spatiotemporal data visualization, offering tools and platforms for creating interactive and accessible visualizations. This paradigm shift has led to a new range of possibilities, especially in the context of visualizing moving objects. Web-based visualization allows users to explore data in an intuitive and user-friendly manner, enhancing the comprehension of complex spatiotemporal patterns. The ability to interact with data directly through a web interface bridges the gap between technical complexity and user accessibility. This section will focus on an application of web-based visualization for traffic dynamics.

Interactive Visualization of Traffic Dynamics Gomes, Santos, and Vidal developed a web-based interactive visualization system for traffic dynamics based on trajectory data [22]. Their system, built with WebGL and JavaScript, allows users to visually explore traffic patterns over time, enabling the identification of congestion points and the study of flow dynamics. The system represents vehicle trajectories as lines in space and time, changing in color and thickness to indicate vehicle density and speed. This dynamic and interactive visualization approach greatly aids in understanding and managing complex traffic systems.

2.6 Visualization Specific to MobilityDB

In addition to the more generic applications-based and web-based visualization techniques, there are a few projects that focus specifically on visualizing data from MobilityDB. These efforts have resulted in integrations with several different visualization libraries and tools.

DeckGL Integration The first notable work on visualization linked to MobilityDB was conducted by Fabrício Ferreira da Silva [23]. In his master's thesis, Fabrício developed a solution that integrates MobilityDB with DeckGL, a WebGL-powered framework for visual exploratory data analysis of large datasets. This work represents an important step towards bringing rich, interactive visualization capabilities to the MobilityDB platform.

QGIS Integration In a preliminary study carried out by Ludéric Van Calck during his master's thesis preparatory work, an exploration of techniques for the integration of MobilityDB with QGIS was conducted [24]. This endeavor resulted in an implementation that was able to achieve a frame rate of 15 FPS for visualizing 100 moving objects. This initial work laid the groundwork for further investigations and advancements in the field of integrating MobilityDB with QGIS for efficient visualization of moving objects. Schoemans et al. [25] developed a plugin called MOVE that integrates MobilityDB with QGIS. The plugin allows users to perform rich transformations and visualizations on moving object data through an interactive interface, as well as to query and display such data in a simple, user-friendly manner. Moreover, MOVE provides the ability to visualize both static and animated spatial data, thereby greatly enhancing the scope and flexibility of the visual exploration process.

OpenLayers Integration In a similar vein, El Bakkali Tamara [26] is working on an integration of MobilityDB with OpenLayers, a high-performance, feature-packed library for creating interactive maps on the web, as part of his ongoing master's thesis. Although the work is still in progress, it promises to be a valuable contribution to the field, given the widespread use and robustness of the OpenLayers library.

2.7 Summary

The field of spatio-temporal data visualization has seen significant advancements since its inception, rooted in the foundational work of Minard [2]. Modern developments, facilitated by technological advancements, have introduced a multitude of tools and methodologies to visualize movement data [4, 5]. Particularly notable is the proposal of a visual analytics approach by Andrienko et al. [6], which presented a novel fusion of interactive visual displays with computational methods for managing and analyzing large volumes of movement data.

A vast range of techniques for visualizing spatio-temporal data has been developed, extending from traditional static methods like GIS, to dynamic and non-interactive methods, and towards more advanced interactive techniques [11]. While

static methods provide effective initial insights into data, they lack the ability to display the temporal evolution of the data [12]. Dynamic non-interactive methods add a temporal dimension, yet may suffer from interpretability issues due to rapid screen changes or potential occlusion problems [6, 13]. More recently, interactive techniques have emerged, allowing users to manage complex data in a more user-friendly and intuitive manner [17]. Nevertheless, Tversky et al. [18] caution that animated graphics are not always superior to static ones, as they may overload the viewer with information, highlighting the need for continued exploration in this area.

Numerous applications of spatiotemporal data visualization have been introduced, particularly in the context of animal movement data [5, 13, 21]. The advent of web-based visualization platforms has led to even more accessible and user-friendly visualization methods, with various implementations specific to MobilityDB also being developed [22, 23, 24, 25, 26].

However, gaps remain in the current state of the art. Despite the abundance of techniques and applications, several challenges persist in the visualization of spatio-temporal data. For instance, handling high-dimensional data effectively, accommodating extreme values or outliers in the dataset, and resolving issues with occlusion in 3D visualizations continue to pose problems. Moreover, the proliferation of interactive visualization tools must be balanced with considerations for user cognitive load, as overly complex animations may impede rather than aid understanding. Additionally, while significant strides have been made in integrating visualization tools with MobilityDB, further research is needed to fully explore and optimize these possibilities.

The gaps identified in the current state of the art offer opportunities for novel contributions to the field, leading us to the subsequent section on the research questions and objectives of this study.

Chapter 3

Tool Review

3.1 Leaflet

Leaflet is a highly recognized, open-source JavaScript library that allows developers to create interactive maps that are mobile-friendly [27]. The library is known for its simplicity, performance, and usability, making it a popular choice for developers.

Leaflet provides a wide range of features that help developers build interactive map applications. These features include tile layer integration, marker placement, popups, and handling of various user interactions such as panning, zooming, and tapping. The tile layer integration allows developers to easily connect to various map providers, while marker placement and popups enable the representation of data points on the map [27].

Despite its feature-rich nature and ease of use, Leaflet does not have built-in support for MVT. MVT is a format utilized for encoding vector tile data that is transferred from server to client. It offers advantages such as smaller file sizes and better performance compared to other formats like GeoJSON, particularly for larger datasets [28]. However, in order to use MVT with Leaflet, developers must either depend on third-party plugins or write custom code, which presents a challenge in efficiently visualizing moving object data from a MobilityDB database on a Leaflet map.

One of the main objectives of this thesis is to tackle the aforementioned challenge and implement efficient MVT support for visualizing moving object data. The evaluation of this approach and its comparison with the traditional GeoJSON-based method forms a crucial part of our work. As we will explore in the subsequent sections of this thesis, this task comes with its own set of complexities and considerations.

For this project, Leaflet plays a central role as it is the core of the frontend implementation. By leveraging its features, we are able to create an interactive visualization of moving object data. However, the limitation in MVT support means that we need to devise our own solutions and approaches, thus leading us to a deeper understanding of both the potential and the challenges of using Leaflet for such a task. This will be discussed in further detail in the subsequent sections of this thesis.

3.2 Raster Tiles

Raster tiles are a common method for serving geographic data over the internet for use in maps and other visualizations. In this technique, a raster image of a map (usually a .png or .jpg file) is divided into square tiles at multiple zoom levels, forming a so-called "pyramid" structure (See Figure 3.1).

At the base of the pyramid (zoom level 0), the entire world is represented by a single tile. As you move up the pyramid, each tile is divided into four new tiles representing the same geographical area at a higher resolution. This division continues for as many zoom levels as are required for the data.

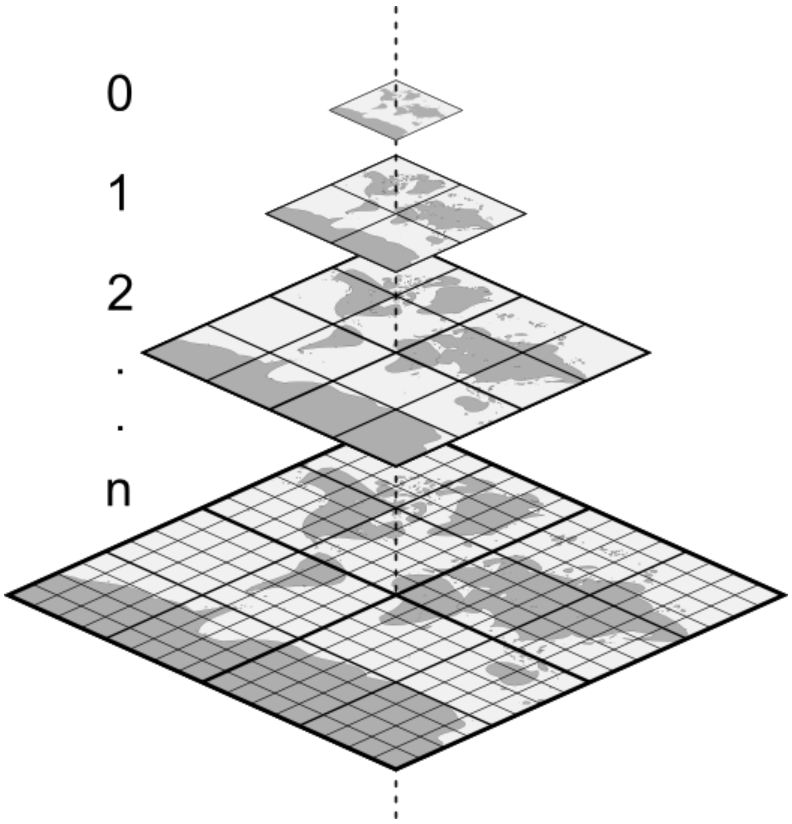


Figure 3.1: Pyramid structure of raster tiles at different zoom levels.

When a map is viewed in a browser, only the tiles corresponding to the current view and zoom level are requested from the server and displayed. This allows large and complex maps to be viewed and navigated smoothly, without the need to load the entire map into memory.

While raster tiles are simple and efficient to display, they have some limitations. The resolution of the data is fixed at the time the tiles are created, meaning that features may become pixelated when viewed at high zoom levels. Also, because the data is rendered into images, it is not possible to interact with individual map features on the client side. These limitations are addressed by the use of vector tiles, which are discussed in the following section.

3.3 Vector Tiles

Vector tiles represent a technology that enables efficient and flexible delivery of geographic data to client applications, such as web browsers or mobile apps. They represent geographic data as a collection of square-shaped tiles at multiple zoom levels, forming a pyramid-like structure. Each tile contains vector data of a specific portion of the map at a certain zoom level, which can be rendered on the client-side. This approach allows for highly interactive and customizable map visualizations, which can dynamically adapt to user interactions such as zooming and panning.

As shown in Figure 3.1, the entire map is divided into a grid of tiles at each zoom level. The tile at the top of the pyramid covers the entire world map at the lowest zoom level. As the zoom level increases, each tile is divided into four smaller tiles, thereby increasing the level of detail.

The key advantage of vector tiles over traditional raster tiles lies in their client-side rendering. While raster tiles are pre-rendered images, vector tiles contain raw vector data (such as points, lines, and polygons), along with attributes and styles. This means that the final rendering and styling of the map can be done on the client-side, allowing for greater customization, interactivity, and efficiency.

Moreover, vector tiles are typically smaller in size compared to raster tiles, leading to faster load times and better performance, especially on mobile devices. They also support interactive features such as hovering, clicking, and dynamic styling, which are not possible with raster tiles.

3.4 Mapbox Vector Tiles

Mapbox Vector Tiles (MVT) represents geographic data as a collection of vector geometries in binary format, primarily used for rendering maps in web browsers [28]. It was designed and developed by Mapbox, a leading provider of mapping and location-based services [29].

Despite its complexities and binary nature making it less human-readable than other geographic data formats like GeoJSON or KML, MVT has become a popular choice for web-based mapping applications. This popularity owes to its flexibility, efficiency, and performance advantages [30]. To work with MVT effectively, various libraries have been developed, including Mapbox GL JS for web applications and Mapbox GL Native for mobile and desktop applications [31].

A significant aspect to consider while working with MVT is the concept of a 'buffer'. Buffers are additional areas around the tile's edge that contain data from the neighboring tiles. This buffering technique is essential because it helps avoid rendering issues at tile boundaries, such as point features appearing cut in half or line features ending prematurely.

As Figure 3.2 demonstrates, the absence of a buffer can lead to a disjointed visualization where features at the boundaries are cut off. The tile borders have also been included in the image to highlight the point at which these features are cut off. Thus, while handling MVT, accounting for buffers is crucial to ensure smooth, consistent visualizations across tile boundaries.

The challenge of buffering is addressed in the PostGIS function 'ST_AsMVTGeom' and the MobilityDB function 'asMVTGeom', both of which include a buffer para-

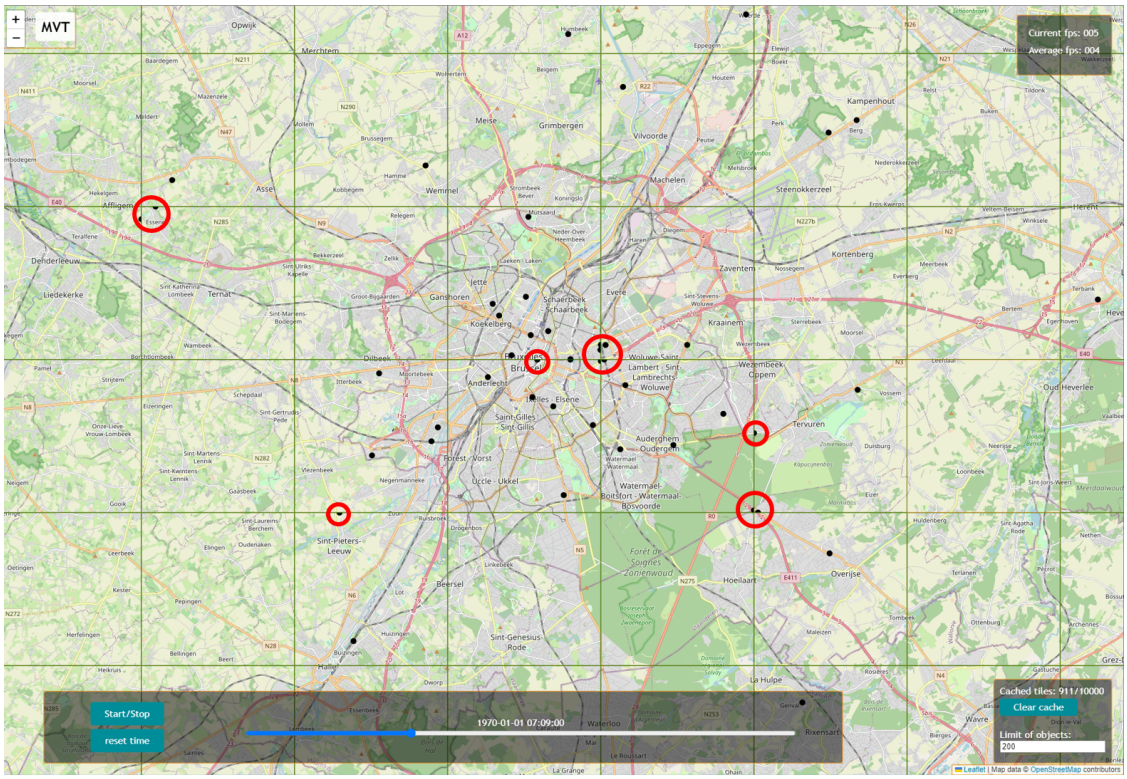


Figure 3.2: Rendering issues due to lack of buffer. The red circles highlight where point features are cut in half at tile boundaries. Tile borders are also shown to illustrate the issue more clearly.

meter. By properly setting this parameter, developers can avoid the visual discontinuities observed when the buffer is not considered, thus ensuring a more coherent and effective visualization of the data.

3.5 Pg_tileserv

Pg_tileserv [32] is a lightweight, open-source server for serving vector tiles dynamically from a PostGIS database. It is developed by Crunchy Data, a company specializing in providing enterprise open-source PostgreSQL and PostGIS technology solutions. Despite its simplicity, pg_tileserv is highly configurable and supports advanced features such as custom SQL queries, per-table configuration settings, support for tileset metadata, and built-in cache system.

Unlike traditional tile servers that require pre-generation of tile sets, pg_tileserv generates tiles on-the-fly based on the current data in the PostGIS database. This dynamic approach is particularly suitable for datasets that are frequently updated or have a high level of detail. Pg_tileserv automatically detects PostGIS tables and views in the database and exposes them as tileset sources, making it very easy to serve new data. Each table can be queried for tiles in MVT format, either for a specific tile or for a tile covering the whole table. The server supports various tile grid schemes, including the popular XYZ scheme used by web mapping libraries like Leaflet and OpenLayers. Tiles can be generated at any zoom level, with the level of detail automatically adjusted based on the zoom level.

One notable limitation of pg_tileserv is that it only supports PostGIS databases, limiting its interoperability with other GIS data formats and databases. However, given the widespread use and powerful capabilities of PostGIS, this is a reasonable trade-off for many applications.

3.6 Existing Visualization Tools

There exist several tools for geospatial visualization that have been widely adopted for various applications. However, their use in the context of visualizing moving object data presents certain limitations. This section discusses a few of these tools and their inherent constraints when dealing with moving object data.

Google Maps Platform Google Maps Platform [33] is a suite of APIs and SDKs that provide developers with a set of mapping and location features to integrate into their applications. Although it has robust capabilities for static mapping and location services, it lacks specialized functionalities for animating and visualizing moving objects. The extent of animation is limited to simple marker movements, which can be insufficient for complex scenarios that involve the animation of numerous objects.

OpenLayers OpenLayers [34] is a high-performance, feature-packed library for creating interactive maps in web applications. While it offers advanced functionalities such as support for vector tiles and integration with various map sources, it doesn't provide built-in support for animating moving objects. Developers need to implement custom solutions, which can be challenging and time-consuming, particularly for large datasets. Another student, Soufian El Bakkali Tamara, is currently exploring a solution for moving object visualization using OpenLayers, and his work can be found on GitHub [35].

Mapbox GL JS Mapbox GL JS [36] is a powerful library for integrating interactive maps into web applications. Its support for WebGL makes it capable of rendering large datasets efficiently. However, similar to other tools, it lacks explicit support for visualizing moving objects. While it is possible to animate markers, this approach might not scale well for scenarios that involve numerous moving objects.

CesiumJS CesiumJS [37] is a geospatial 3D mapping platform for creating virtual globes. Its primary strength lies in the visualization of global-scale geospatial data. It does support the animation of moving objects, but primarily for 3D data. For 2D map-based visualization of moving objects, other solutions might be more suitable.

ArcGIS API for JavaScript The ArcGIS API for JavaScript [38] is part of Esri's suite of GIS tools. It provides robust capabilities for creating 2D and 3D interactive maps. While it supports animation, its primary focus is on static geospatial data visualization. For real-time or highly interactive moving object visualization, custom solutions might need to be developed.

Deck.gl Deck.gl [39] is a WebGL-powered framework designed for large-scale data visualization. It excels in rendering large datasets and provides built-in support for visualizing trajectories and other forms of movement data. However, the complexity of Deck.gl and its focus on 3D visualizations might make it overkill for applications that only require simple 2D map-based visualizations of moving objects. Fabrício Ferreira da Silva was the first to propose moving object visualization using MobilityDB, focusing particularly on the MVT implementation and its comparison with the GeoJSON implementation in his master's thesis [23]. His work serves as a pioneering foundation for this project and Soufian El Bakkali Tamara's work, despite facing challenges with larger datasets.

These limitations underline the challenges that need to be addressed for efficient and effective visualization of moving object data. This thesis aims to explore solutions to some of these challenges by implementing and evaluating methods to support MVT in Leaflet.

3.7 Cache Systems: Overview and Comparison

Caching is a critical aspect of any data-intensive system for performance optimization. By storing frequently used data in a readily accessible location, cache systems can significantly reduce data retrieval time and server load. Several cache strategies are commonly employed in various applications, each with its strengths and weaknesses.

Least Recently Used The Least Recently Used (LRU) caching strategy removes the least recently used items first when the cache is full. This strategy assumes that data items accessed recently will likely be accessed again in the near future. It is relatively simple and efficient in implementation. However, LRU does not take the frequency of access into consideration, which may lead to higher cache misses if certain items are accessed sporadically but repeatedly.

First In, First Out The First In, First Out (FIFO) strategy is another simple caching algorithm, where the oldest item (i.e., the first one to be stored) is removed when the cache is full. This strategy is easy to understand and implement, but it does not consider the usage pattern of data items, leading to potential inefficiency. An item that is infrequently accessed but happened to be loaded first will remain in the cache, possibly at the expense of more frequently accessed data.

Least Frequently Used The Least Frequently Used (LFU) caching strategy is a more sophisticated caching strategy that removes the least frequently used items first. LFU excels in situations where the frequency of access is a more important consideration than recency. In other words, if certain data items are accessed sporadically but frequently overall, LFU will prioritize keeping them in the cache. LFU is particularly relevant in our context, as the visualization of moving objects often requires repeated access to a subset of map tiles, making frequency a more critical consideration than recency.

However, LFU has its drawbacks, such as the potential for cache pollution, where infrequently accessed items can remain in the cache if they were frequently accessed in the past. Moreover, the overhead of maintaining frequency information could also be a concern. Nevertheless, for the purpose of this project, LFU emerged as the optimal strategy due to its emphasis on access frequency, which aligns closely with the usage pattern of a map visualization tool.

In conclusion, the choice of caching strategy depends heavily on the specific requirements and usage patterns of the application. LFU, with its focus on access frequency, has shown to be the most appropriate for a map visualization tool dealing with moving objects.

3.8 Well-Known Binary and Well-Known Text

Well-Known Binary (WKB) is an integral part of the Open Geospatial Consortium (OGC) Simple Feature Access specification [40]. WKB is a standard binary format used for representing geometric objects as sequences of bytes. It offers a compact representation of geometry, being an excellent choice for scenarios where space efficiency and transfer speed are paramount. The size of data in WKB format is significantly smaller compared to other formats like GeoJSON, resulting in faster transmission and reduced storage costs. However, the downside of WKB is that it is not directly human-readable and does not have direct support from certain libraries and tools. Consequently, it needs conversion into a different format, often Well-Known Text (WKT), for use in unsupported environments.

Well-Known Text (WKT) is another part of the OGC Simple Feature Access specification [40]. WKT is a text markup language used for representing vector geometry objects, spatial reference systems of spatial objects, and transformations between spatial reference systems. Unlike WKB, WKT is a human-readable format, making it a common choice for data interchange between systems that do not natively support binary formats. WKT's readability allows developers to debug and understand the data more easily, but at the cost of larger size compared to WKB.

Format	**Encoding**
WKB	0101000000000000000003e400000000000002440
WKT	POINT(30 10)
GeoJSON	{"type":"Point","coordinates":[30,10]}

Table 3.1: Comparison of WKB, WKT, and GeoJSON Encodings for a Point

MobilityDB takes advantage of both WKB and WKT formats [10]. It uses WKB for internal storage due to its compactness, and can output data in either WKB or WKT format, depending on the needs of the client application.

However, the temporal features of MobilityDB add a layer of complexity to these formats. Traditional geometric objects become "moving objects" with added time dimensions, such as the validity period of geometric shapes or interpolation behavior for moving trajectories. This means that WKB and WKT representations

from MobilityDB include temporal information beyond what would be found in static geometry.

This added complexity can pose a challenge when trying to parse and utilize these formats in environments not designed for them. For example, Leaflet, one of the commonly used libraries for interactive maps, does not natively support WKB or WKT. This necessitates additional steps of deserialization and parsing before the data can be effectively used, adding overhead and computational complexity to the system.

3.9 FastAPI

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints [41]. It is extremely easy to use and follows an asynchronous model for handling requests, which makes it particularly efficient for I/O bound applications.

FastAPI is built on the Starlette framework for the web parts and Pydantic for the data parts. This enables it to deliver high performance, easy data validation and serialization, and asynchronous request handling. Its key features include automatic interactive API documentation, OAuth2 authentication with JWT tokens, and template rendering.

While FastAPI has many advantages, it also has its limitations. Since it is relatively new, it may not have the wide range of third-party extensions available to some more established frameworks. Furthermore, while its asynchronous model is powerful, it requires careful handling of database sessions and connections to prevent blocking operations, which can add some complexity to the development process.

Chapter 4

Project Description

4.1 Project Overview

This project aims to provide a solution to a common issue in geospatial analysis: efficiently visualizing moving objects on a map. The central component of this endeavor is a visualization tool designed to handle substantial amounts of spatiotemporal data, particularly moving objects, and render them on a map.

The project employs several open-source technologies, including MobilityDB for handling moving object data, pg_tileserv for serving dynamic map tiles, Leaflet for rendering maps and adding interactive features, and React for building the user interface. The choice of these technologies is based on their widespread use, active development communities, and robust capabilities, especially in the context of geospatial applications.

Moreover, the project tackles the performance bottleneck that often accompanies data-intensive applications, namely the latency in data retrieval and rendering. To this end, we introduce an innovative solution: a cache system based on the Least Frequently Used (LFU) policy. The LFU cache system prioritizes storing map tiles that are more frequently accessed, therefore reducing the time to retrieve and render these tiles.

The project is implemented and evaluated using two different datasets, each with different characteristics. This approach ensures the tool's versatility and adaptability to various use-cases.

By implementing this visualization tool, we aim to make a significant contribution to the field of spatiotemporal data visualization and offer a robust, open-source solution for applications involving moving objects.

4.2 Technologies Used

4.2.1 MobilityDB

Developed at the Université Libre de Bruxelles, it provides comprehensive features for efficiently managing, querying, and analyzing moving object data. This makes MobilityDB a natural choice for applications that deal with trajectories of moving objects, such as vehicles, wildlife, or natural phenomena.

The use of MobilityDB in this project was both a requirement due to its development at our institution, and a technical decision justified by its capabilities. It serves as the primary data storage system for the moving objects data. Its efficient handling of spatiotemporal data, offering advanced data types and functions specifically designed for managing moving objects, made it an ideal choice for our needs.

Furthermore, MobilityDB's compatibility with the robust and mature technologies PostgreSQL and PostGIS was a significant factor in our selection. This compatibility allowed us to leverage the capabilities of these technologies while also benefiting from the spatiotemporal extensions provided by MobilityDB. The open-source nature of MobilityDB was another important aspect, aligning with the overall goal of using open-source technologies for this project.

Despite MobilityDB's multitude of advantages, it has some complexities regarding its setup and usage due to the advanced nature of its functionality. However, its benefits in managing and analyzing spatiotemporal data outweigh these challenges, making it a valuable tool in our project. The documentation being well written and the availability of the different developers made this challenge less difficult than it appears.

4.2.2 Leaflet

The project employs Leaflet to render interactive maps in the frontend, and to overlay the spatiotemporal data served by pg_tileserv. The selection of Leaflet was driven by its ease of use, flexibility, and performance. Moreover, Leaflet's strong community and extensive plugins were also decisive factors.

It is noteworthy that Leaflet does not have built-in support for vector tiles (MVT). To overcome this challenge, we decided to use the Leaflet.Vectorgrid plugin, which we modified to handle the spatiotemporal data types.

The open-source nature of Leaflet aligns well with our project's goal of leveraging open-source technologies. Despite the need for additional efforts to handle MVT, Leaflet's versatility and extensive functionality make it an optimal choice for the map visualization component of the project.

4.2.3 React

React is a popular open-source JavaScript library for building user interfaces, particularly single-page applications. It is developed and maintained by Meta and a community of individual developers and companies [42]. In fact, according to the 2022 JAMstack survey, React was the most used JavaScript framework [43].

In our project, React serves as the backbone of the frontend, handling the user interface and user interactions. The use of React was motivated by its virtual DOM feature, which optimizes re-rendering and thus improves the application's performance when displaying dynamic, real-time data.

React's robust ecosystem, coupled with comprehensive documentation and an active community, were additional factors in our choice. However, React has a steep learning curve, especially for developers unfamiliar with its concepts and paradigms. Despite this, its benefits and widespread adoption in the industry made it an excellent choice for our project.

4.2.4 LFU Caching System

As said before, the algorithm keeps track of the frequency of access for each data item. When the cache reaches its maximum size and a new item needs to be inserted, the item with the lowest frequency count is evicted.

The LFU caching system is particularly suited to scenarios where the frequency of access is more significant than the recency of access. This property made it a fitting choice for our project, where we needed to cache the most frequently used map tiles for efficient visualization.

4.2.5 Pg_tileserv

As said in Section 3.5, pg_tileserv uses the vector tile format for efficient transfer and quick rendering of geographic data. The server does not need any pre-seeding or caching and can serve directly from the database, making it a dynamic and flexible solution for serving geographic data.

In this project, pg_tileserv is used as a vital component for serving the spatiotemporal data stored in the MobilityDB to the frontend for visualization. This use of pg_tileserv was driven by its ability to directly serve from the database, which ensured that the most recent data is always available for rendering. Moreover, its ability to provide data in the vector tile format was crucial to support the MVT-based visualization implementation.

Pg_tileserv also provide a built-in caching system which is really easy to use. With the previously presented cache system that we implemented, we have now a two layer cache-system (backend and frontend) for the MVT-implementation.

The open-source nature of pg_tileserv was another key aspect that made it align well with our project's goal of leveraging open-source technologies. However, while it provides dynamic and efficient data serving capabilities, it necessitates a careful configuration and optimization for high-performance use cases, such as large numbers of moving objects in our project.

4.2.6 FastAPI

The selection of FastAPI for this project was primarily influenced by its proven reliability and straightforward setup and usage in previous smaller projects. FastAPI provides a streamlined approach to designing and developing robust APIs that can effortlessly scale to support larger applications. Its uncomplicated installation process and ease of use position it as a preferred tool among Python developers striving for efficient web application development.

The operation of the FastAPI server in this project was facilitated by the ASGI server, Uvicorn. Uvicorn, a highly efficient ASGI server implementation that employs uvloop and httptools, brings the advantages of asynchronous programming to the fore. It enables FastAPI to function as a web server, equipped to manage a large number of concurrent connections.

4.3 Architecture and Design

4.3.1 System Overview

Our visualization tool is designed as a web-based application built on a client-server model. The system is primarily divided into three significant components: the frontend, the backend, and the database.

- **Frontend:** The frontend of our application is designed using React, a popular JavaScript library for building user interfaces. The frontend interacts with the user, accepts user requests, and presents data in an easily understandable format. For mapping and geospatial capabilities, the Leaflet library is utilized. A caching layer is implemented in the frontend, storing frequently accessed tile data in memory using a LFU (Least Frequently Used) caching system.
- **Backend:** The backend of our system is divided into two distinct services:
  - **FastAPI Server:** The FastAPI server, implemented in Python, is responsible for certain database requests.
  - **Pg_tileserv Server:** This server is used for the MVT implementation.
- **Database:** The MobilityDB extension of PostgreSQL serves as our database. It is designed to manage and analyze object trajectories over time. The database stores the spatial data, and when a request is received, it fetches and returns the requested data to the backend.

This system architecture allows for efficient interaction between the user and the spatial-temporal data. The user inputs are processed and visualized in real-time, offering an interactive tool to track and analyze moving objects.

4.3.2 Design Principles

Our visualization tool was built adhering to a number of fundamental design principles. These principles were chosen to ensure a high degree of user-friendliness, performance, and maintainability.

- **Performance Efficiency:** With a two-layer caching system, one at the frontend using LFU, and one at the backend (Pg_tilserv) using its built-in cache system, the tool achieves a high degree of performance efficiency, especially for frequently accessed data. This results in faster response times and better user experience.
- **Maintainability:** By segregating the system into the frontend, backend, and the database, each with its own distinct functionality, the design allows for easier maintenance and updates. Each part can be updated or modified independently, ensuring the overall system remains robust and stable.

- **Open Source:** The tool is built using open-source technologies which enhances its accessibility and promotes interoperability. The choice of technologies like React, Leaflet, MobilityDB, FastAPI, and Pg_tilserv, all being open-source, aligns with this principle.
- **Scalability:** The design of the tool allows for scalability. The system can handle an increasing number of moving objects and can accommodate more complex and larger datasets over time, thanks to the robust database system (MobilityDB) and efficient data handling at the frontend and backend. As we will see in the results chapter, in order to achieve this scalability, the PostgreSQL server must be relatively powerful.

4.3.3 Component Design

The system is composed of several distinct components, each fulfilling specific roles within the overall structure. This component-based design fosters modularity, scalability, and maintainability. Here, we present an overview of the design of the major components of the system:

- **Frontend:** The frontend, designed with the React library, provides a dynamic user interface. The Leaflet library is used for geospatial visualization.
- **FastAPI Server:** The FastAPI server handles specific requests for simple data, such as fetching the minimum and maximum timestamp values, and returning the GeoJSON file of the data.
- **Pg_tileserv Server:** The pg_tileserv server caters to the MVT implementation and utilizes its built-in caching mechanism to cache responses for a certain TTL.
- **Database:** The MobilityDB extension of PostgreSQL ensures efficient storage and handling of spatio-temporal data.

4.3.4 Data Flow

Understanding the flow of data through the system is integral to appreciating how the system functions as a whole. The process begins with user interaction at the frontend, and involves several stages before the data is visualized on the map interface (See Figure 4.1).

- **User Interaction:** The user starts by interacting with the interface, where they can specify a limit to the number of moving objects displayed from the selected dataset. This action can be performed at any time and will initiate the subsequent steps anew.
- **FastAPI Server Request:** Depending on the user's action, the React frontend initiates a request to the FastAPI server. This will request the maximum and the minimum values of the timestamps in the database.
- **Database Query:** The FastAPI server translates this request into an appropriate SQL query and sends it to the MobilityDB database.

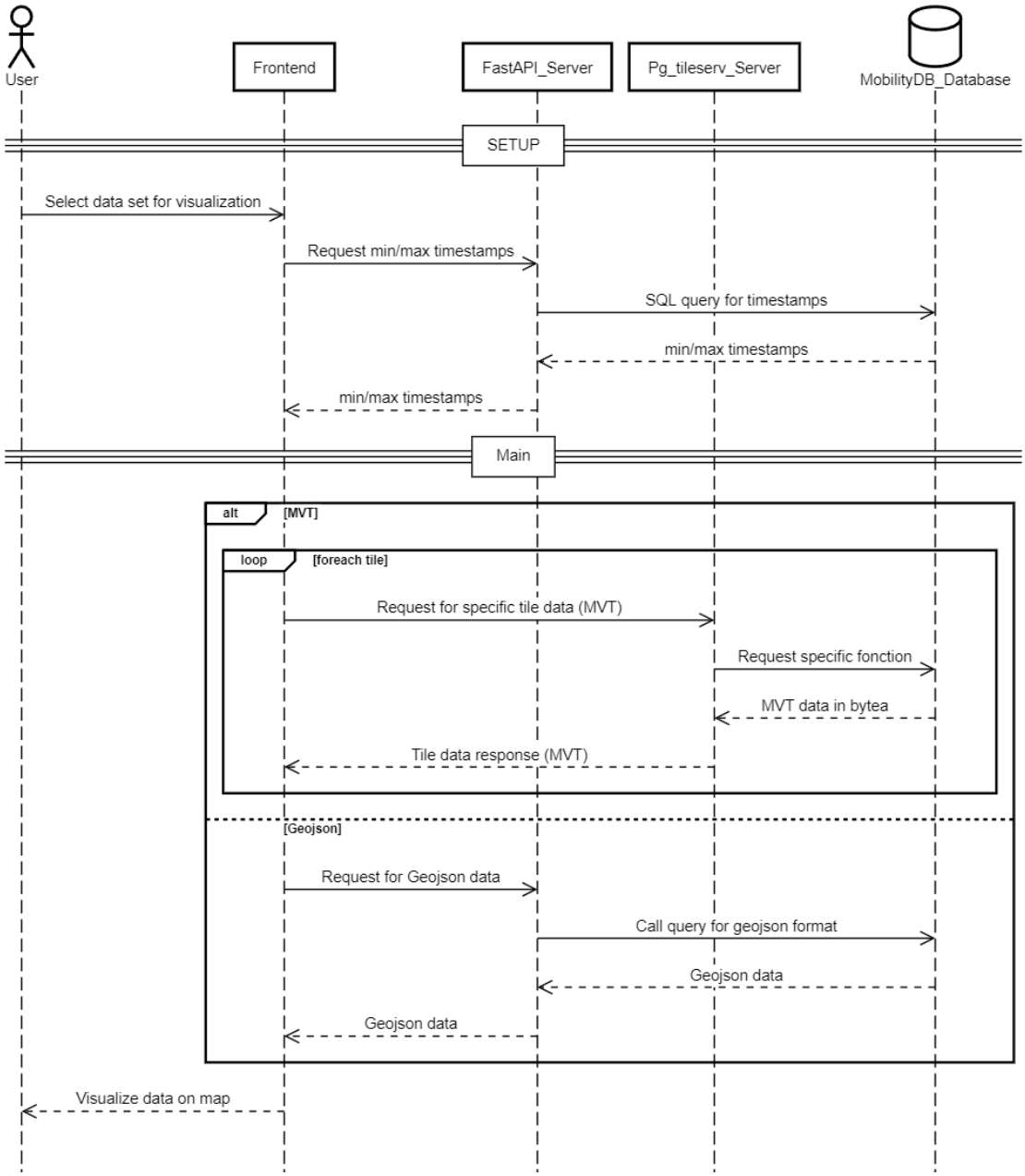


Figure 4.1: Basic sequence of data

- **Data Retrieval:** The database processes the SQL query, retrieves the requested data, and sends it back to the FastAPI server. The FastAPI server then sends the data to the frontend.

MVT specific

- **pg_tileserv Request:** For MVT implementation, if the data are not cached in the frontend, requests for specific tile data are made to the pg_tileserv server. The server utilizes its built-in caching system to speed up responses if the data have been recently used (not exceeding the TTL). Otherwise, it will ask for the data to the Database

- **Data Response:** The servers send back the retrieved data as a response to the frontend. The data is specific tile data in MVT format.

GeoJSON specific

- **GeoJSON data:** For the GeoJSON implementation, a simple request is made to the FastAPI server the will send a specficic query to the MobilityDB Database in order to retrieve the data in the GeoJSON format

Both implementation

- **Data Visualization:** The frontend processes the received data and visualizes it on the map interface using the Leaflet library. In the case of MVT, an LFU caching mechanism on the frontend ensures efficient handling of frequently accessed tile data.

Note that in the specific case of MVT, each time the User move the map, we need to restart to the **MVT specific** step. Which is why a caching system is implemented in order to speed up this process on already loaded tiles. This specific flow can be seen on the Figure 4.2

4.4 User Interface Design

The interface consists of several components, shared across both MVT and GeoJSON implementations, along with some components specific to the MVT implementation (See Figure 4.6).

The shared components include:

- **Time Control:** The `Time Control` component, crucial for manipulating the simulation's time frame, is strategically located at the bottom of the map display. Drawing inspiration from standard media player designs, we deemed this placement intuitive as users are accustomed to finding time control options in this area. The component allows users to start, stop, or reset the simulation's timestamp, providing a dynamic interaction with the displayed data.
  - **Start/Stop Button:** This button is used to start or stop the motion of the moving objects.
  - **Reset Time Button:** This button sets the current timestamp to the minimum value of the timestamp as provided by the FastAPI backend server.
  - **Time Slider:** This slider ranges from the minimum to the maximum timestamp, increasing as time progresses. It can be used to select a specific time when the motion of the objects is active. (See Figure 4.3)

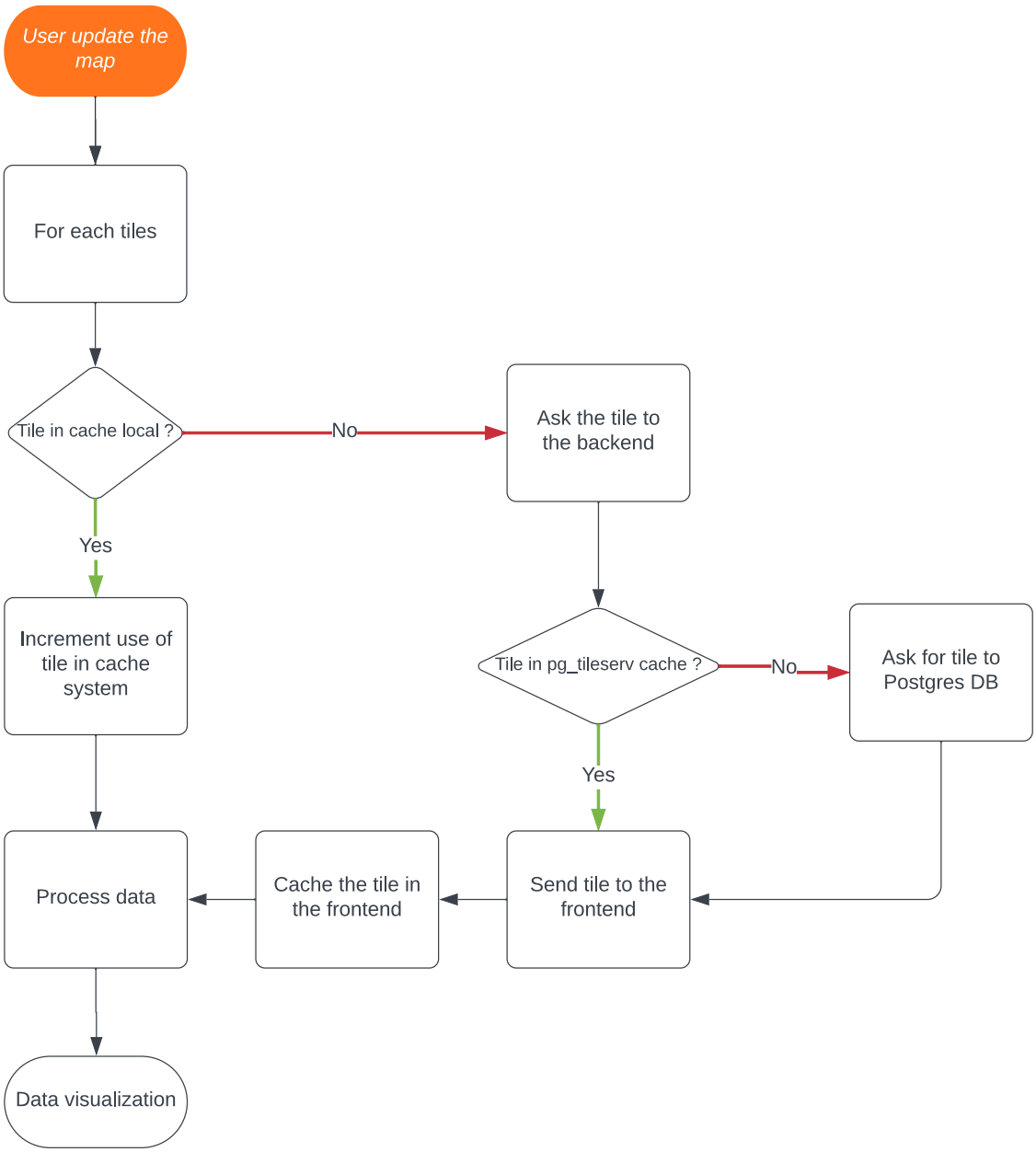


Figure 4.2: Cache system

Figure 4.3: Time slider

- **Parameters:** Adjacent to the `Time Control` component on the right, we positioned the 'Parameters' component (See Figure. This placement ensures that all user controls are conveniently located at the bottom of the screen. This component allows users to define specific parameters of the simulation, including the limit of moving objects to be shown. Having these controls close

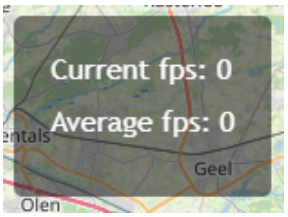


Figure 4.4: Updates information

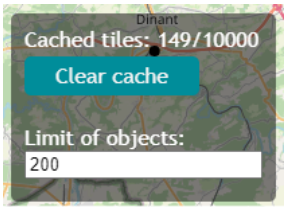


Figure 4.5: Settings for the simulation

to the time controls offers a streamlined interface, enabling users to quickly and efficiently adjust their settings without needing to navigate away.

  - **Limit Input:** This input field is used to define the limit of moving objects shown on the map.

- **Performance Info:** The `Performance Info` component (See Figure 4.4) is placed at the top right corner of the screen. This placement complements the zoom-in/zoom-out buttons typically found at the top left of mapping interfaces. The component provides important performance metrics such as the current and average frames per second, crucial for testing and optimizing the web application. However, to maintain a clean, uncluttered user interface in the production version, this component could be removed.

  - **Current and Average FPS:** These indicators display the current and average frames per second, providing an insight into the performance of the implementation. The average FPS is reset every time the motion of the objects is restarted

The MVT-specific components include:

- **Parameters (MVT)**

  - **Cached Tiles:** This indicator displays the number of tiles cached and the maximum number of tiles that can be cached. The maximum value is hardcoded into the frontend and cannot be changed by the user.
  - **Clear Cache Button:** This button clears the cache of all tiles. After pressing the button, only the current tiles on the screen will be cached.

This interface design aims to provide users with the necessary control and information to effectively use the visualization tool, while maintaining a clean and uncluttered presentation of the moving objects' data. It is noteworthy that any changes in the interface are reflected in real-time for the user. This includes changes

to the limit of moving objects, alterations in the time settings, as well as the initiation and pause of movement, among others.

The component-based architecture of our application provides considerable benefits, particularly in terms of flexibility and modularity. A key advantage lies in the convenience with which we can display two distinct implementations simultaneously, as illustrated in Figure 4.7. This capability proves especially beneficial when working with multiple databases or when it is necessary to exhibit two subsets of the same dataset. For instance, one might want to separately visualize the positions of buses and trams on distinct maps.

By isolating distinct functionalities into separate components, we ensure that each aspect of the application can function independently. This structure not only allows for the simultaneous display of multiple database connections, but also promotes ease of maintenance, as changes to one component have minimal impact on others.

Additionally, the component-based design promotes code reusability, which can significantly speed up the development process and reduce the risk of errors. This makes it much more convenient to experiment with different implementations and visualize their effects in real-time, directly within the user interface.

In essence, the component-based architecture provides a versatile and robust framework, enhancing the user experience by providing diverse and concurrent views of data, and facilitating ongoing development and maintenance efforts.

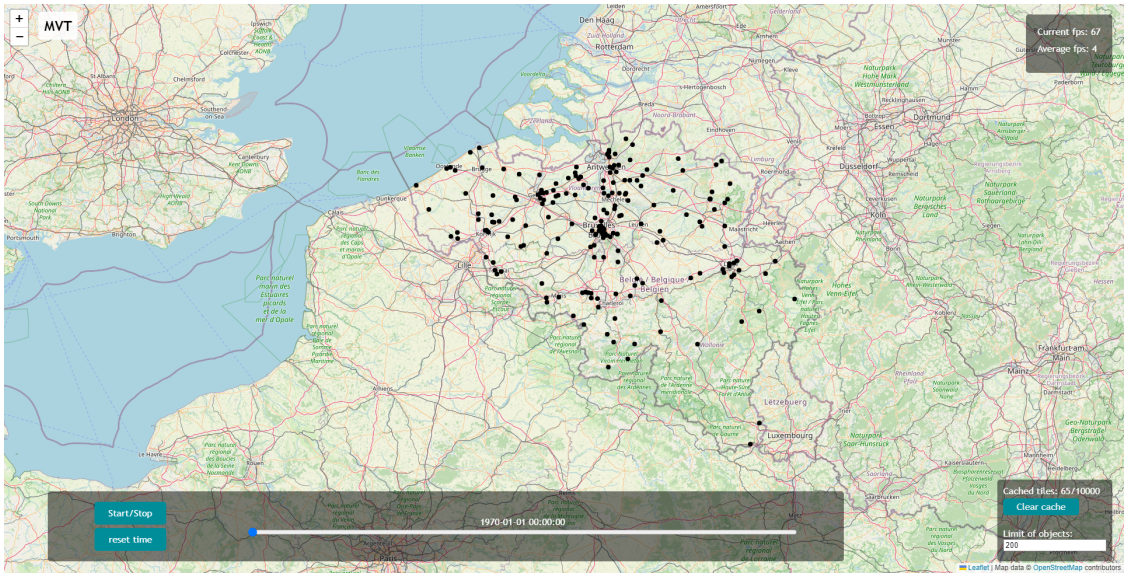


Figure 4.6: Simple version of the controls

4.5 Data Used

In order to test the functionality and performance of the visualization tool, two distinct datasets were utilized. Both datasets consist of tgeompoint data, which serve as the basis for generating the tile or GeoJSON data for our visualizations.

- **Persona Dataset:** The first dataset is a synthetic dataset referred to as the `Persona` dataset. It was designed to simulate the daily travels of a large

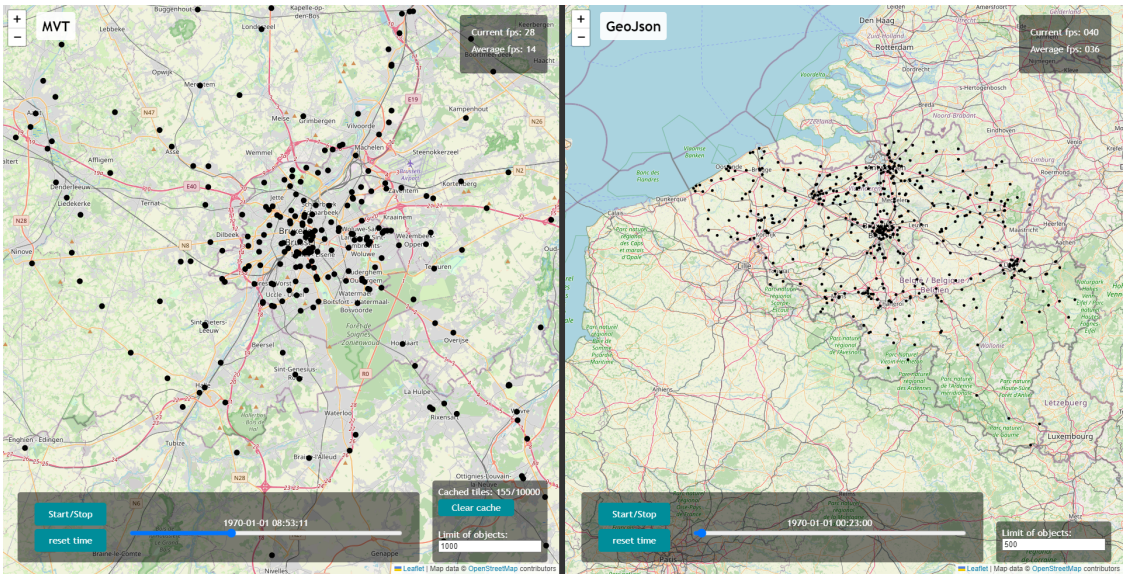


Figure 4.7: MVT and GeoJSON implementation side by side

number of individuals within Belgium. The data replicates typical daily movements such as commuting to work and returning home. The dataset was graciously provided by Syty.io.[1] The dataset is easy to import into our system, being distributed as a .sql file containing all necessary commands for data importation. This dataset, on average, includes 421 instants per day.

- **AIS Data:** The second dataset used in this project is the Automatic Identification System (AIS) dataset. This dataset consists of daily positional data for various maritime vessels. On average, the AIS data contains approximately 1446 instants for each day. The data allows users to choose the duration of import, from a single day up to a full week, or even the entire dataset, which dates back to 2006. However, due to resource constraints on the local machine running all components of the project, we have chosen to limit the imported data to one week. The AIS data is provided in a CSV format, simplifying the import process. Additionally, Prof. Zimányi, the project supervisor, has created an SQL script that automates the database creation and data import processes. The AIS data can be accessed via the AIS website.[2]

These two datasets provide a comprehensive and diverse set of data points for our system, allowing for robust testing of the visualization tool under various conditions and data scenarios.

[1] `https://www.syty.io/`
[2] `https://web.ais.dk/aisdata/`

4.6 Project Workflow

4.6.1 Design

The project's design and architecture drew significantly from the work of Fabrício Ferreira da Silva [23]. Fabrício's approach provided a solid groundwork and inspired the choice of the database, the usage of pg_tileserv, and the adoption of React for frontend development.

However, the current project diverges from Fabrício's work in several key areas. A primary distinction lies in the choice of the library for data visualization. As per the project requirements, Leaflet was chosen for this purpose due to its extensive capabilities, ease of use, and active community support.

A significant enhancement in the current project is the implementation of a local caching system. During the development phase, it was observed that the Vectorgrid extension used with Leaflet was underperforming in certain scenarios. As a response, a local Least Frequently Used (LFU) caching system was designed and implemented on the client side. This enhancement significantly improved performance by reducing redundant data fetch operations.

Another difference comes from the need for a backend server for retrieving simple data from the database. While in Silva's work, these data were hard-coded, this approach was found to be insufficient for the needs of this project. To accommodate a more dynamic data retrieval mechanism, a FastAPI server was developed. This backend server provided a flexible and efficient means to query necessary information from the database, offering a more adaptable solution to data handling.

4.6.2 Evaluation

This project anticipates the use of two datasets: Persona and AIS. These will be evaluated over different numbers of visible moving objects and over different spans of time (one day for the Persona dataset, and one, two, or seven days for the AIS dataset). The metrics for evaluation will include:

- Average updates per second (without map panning or zooming)
- Average updates per second (with map panning and zooming)
- Memory usage of the webpage

The MVT implementation will be evaluated based on the average loading time of a tile, while the GeoJSON implementation will be evaluated based on the average loading time for the entire GeoJSON file.

This brief overview provides an understanding of how the project's performance will be assessed. A more detailed evaluation process, discussion of the testing results, and comparison of the performance between the different implementations will be provided in the "Testing and Evaluation" chapter.

4.7 Unique Features and Highlights

This project contains several distinctive features and innovations which set it apart from existing solutions for visualizing spatiotemporal data. The significant highlights are as follows:

- **Extended Vectorgrid Protobuf:** In this project, the class `vectorgrid.protobuf` from the Leaflet Vectorgrid library has been extended to handle moving objects. This modification enables the library to support spatiotemporal data types, which is not a built-in feature in the original library. This decision was an innovation made necessary by the project's use of Leaflet, a requirement that introduced a unique challenge compared to projects using the DeckGL library, which has built-in support for moving objects. It is this creative solution that allows for the display and manipulation of spatiotemporal data in a Leaflet-based project.

- **LFU Cache System:** A unique feature of this implementation is the custom solution for data caching, utilizing a Least Frequently Used (LFU) cache system. This system is implemented in the frontend and stores data in memory, providing a performance advantage by keeping the most frequently used tiles readily available. This feature is an original contribution to the project and distinguishes it from other similar applications.

- **Optimized pg_tileserv Caching:** Unlike the implementation of Fabricio, this project leverages the built-in caching system of pg_tileserv. The caching parameters have been configured to yield faster response times for tile requests. This optimization significantly improves the performance and the user experience, making for a more efficient and responsive tool.

- **User-Centric Interface Design:** Distinguished from other similar projects such as those based on Deck.GL, which often rely on pre-existing components with hard-coded values, our project emphasizes on user interactivity and control over the visualization process through a custom-built interface. It offers users extensive capabilities to manage various interactions including data selection, zoom levels, playback speed, and time windows, amongst others. This user-centric design approach, avoiding hard-coded values, enhances the user experience by providing a more dynamic, flexible, and insightful visualization of moving object data.

These unique features and innovations represent the project's contribution to the field of spatiotemporal data visualization. The extension of `vectorgrid.protobuf`, the implementation of a LFU cache system, and the optimization of the pg_tileserv caching configuration all underscore the project's commitment to efficient data handling and performance optimization.

Chapter 5

Implementation

5.1 System Setup

The project's development was carried out on a Windows 10 operating system. The core database system and its extensions, on the other hand, were installed and executed within the Ubuntu subsystem for Windows. The integration of this subsystem was deliberate; it provided the ease of use and installation of the Linux environment while remaining within the familiar and flexible Windows ecosystem.

The project heavily relies on Git. This project was publicly hosted on my GitHub,[1] enabling transparency, accessibility, and potentially fostering collaboration with other developers.

5.1.1 Hardware Specifications

- CPU: AMD Ryzen 7 3700x (8 physical cores-16 virtual cores-3.6GHz)
- Memory: 32 GB DDR4 2400 MHz
- GPU: RTX 3060
- Hard Drive: M2 SSD

5.1.2 Software Specifications

- PostgreSQL version: 13
- PostGIS version: 3.2.1
- Python version (for FastAPI server): Python 3.9
- Node version: v16.17.1
- Leaflet: 1.9.3
- leaflet.vectorgrid: 1.3.0
- pg_tileserv: 1.0.9

[1] https://github.com/flobaudry/mobilitydb-leaflet

- MobilityDB version: develop branch compiled using MEOS
- Operating system: Windows 10 with Ubuntu subsystem

5.1.3 Setup Process

The entire setup process is freely available on my GitHub page,[2] but here are the key insights:

- **Frontend:**
  - Run `npm install`
  - Start the frontend with `npm start`
- **FastAPI server:**
  - Install necessary python packages with `pip install -r requirements.txt`
  - Copy `.env.dist` to `.env` and modify the values to match your database information
  - Start the server using `uvicorn wsgi:app`
- **Pg_tileserv:**
  - Specify the database URL in the `.bashrc` file or modify the config file to specify your database (`DbConnection = "user=you host=localhost dbname=yourdb"`). More information on the official documentation page.[3]
  - To enable the cache in pg_tileserv, uncomment the `CacheTTL` line in the config file and set the value that you want for the TTL of your cached data.
- **MobilityDB:**
  - Follow the installation instructions on their GitHub page.[4]
  - To generate the MEOS library, pass the additional parameter `-DMEOS=on` to `cmake`. This information is specified in the MobilityDB's wiki page.[5]

5.1.4 Justifications for Version Choices

- **PostgreSQL 13:** This was the version already installed on the Ubuntu subsystem. This installation was done during initial work with MobilityDB before the commencement of this thesis project.
- **Python 3.9:** This version was chosen since it was the latest version installed on the development machine. The new features in the more recent Python versions were not essential for this project, so updating the version was deemed unnecessary.

[2] `https://github.com/flobaudry/mobilitydb-leaflet`
[3] `https://access.crunchydata.com/documentation/pg_tileserv/1.0.9/installation/`
[4] `https://github.com/MobilityDB/MobilityDB`
[5] `https://github.com/MobilityDB/MobilityDB/wiki/Building-MobilityDB-and-MEOS`

- **Node (version v16.17.1):** This version was chosen due to previous development experience with Node for React Native. It was the version installed during that development, and it was used for this project as well.
- **PostGIS 3.2.1 and other libraries:** The latest versions of these libraries were chosen, as they provided the most up-to-date functionalities and support.

5.2 Codebase Overview

The project is organized in a hierarchy of directories, each serving a specific purpose. Below is the structure and a brief description of each directory and file:

- `backend/` contains all the code to run the FastAPI backend.
  * `requirements.txt` contains all the libraries needed to run the FastAPI server.
  * `wsgi.py` contains the FastAPI app that will be used with uvicorn.
- `frontend/` contains all the code to run the React frontend.
  - `public/` This directory is part of the Create React App's structure. It contains static assets such as HTML file, images, and any files that can be referenced in the HTML.
  - `src/` contains all the source of the React project. It is mostly generated by `create-react-app` but the `App.js` is modified to run with our custom components.
    - `components/` contains the custom React components for this project.
      * `GeojsonLayer.jsx` contains the component for the GeoJSON implementation.
      * `MVTLayer.jsx` contains the component for the MVT implementation.
      * `Slider.jsx` contains the component used for the time slider. It is used by both GeojsonLayer and MVTLayer.
  * `package.json` A manifest file that includes metadata about the project. It specifies the packages required for the project.
  * `package-lock.json` An automatically generated file that is meant to lock the versions of the dependencies to ensure consistency across installations.
* `.env` File containing the information of the database for the FastAPI server (In the GitHub repository, only a `.env.dist` file is available to serve as a template).
* `.gitignore` specifies the files and directories that should be ignored by git.

* `functions.sql` contains some queries needed for the project to run in the database.

* `LICENSE` contains the license information about this project.

* `README.md` README file of the project.

The project follows standard conventions for naming files and directories, enhancing readability and under- standing. It is organized in a way that separates backend and frontend, with clear distinctions and connections between them, creating a streamlined workflow.

5.3 Initial Implementation and Challenges

In the early stages of the project, the first phase of the development was focused on using the original `vectorgrid.protobuf` class, without any modifications. The initial goal was to facilitate visualization of the data represented by the tiles. However, the original structure of `vectorgrid.protobuf` did not natively support spatiotemporal data handling, thus requiring a different approach to accommodate this requirement.

To handle the challenge of spatiotemporal data, the burden was transferred to the PostgreSQL server. A new parameter, `timestamp`, was introduced to the PostgreSQL function to enable it to work in conjunction with the `attimestamp` feature from MobilityDB. This `attimestamp` feature was critical as it fetched the current geometry corresponding to the data, allowing it to be processed further.

During the rendering of tile layers, a request was sent to the pg_tileserv server. The purpose of this request was to fetch the next set of tiles correlated with the current timestamp. The `setUrl` function of the vectorgrid.protobuf class was used for this operation. It is important to note that this request was made synchronously, i.e., after the complete loading of the current tile layer.

The first implementation of this project served as a functional prototype that helped to validate various features. It enabled the testing of time incrementation, start and stop button operations, time slider. The hands-on testing of these features was instrumental in understanding the system's dynamics and identifying the areas for enhancement.

However, despite the successes of the initial phase, it also brought forth significant challenges. As the implementation was tested with larger and complex datasets, the PostgreSQL server's response time emerged as a significant bottleneck. The server struggled with timely responses, especially when handling extensive data.

Upon completion of the rendering process for the tiles associated with a particular timestamp, several requests were sent to the server. This high demand on the server became a challenge when dealing with large datasets.

This issue led to the development of the caching system as a solution. The caching system allowed for local storage of the spatiotemporal data, reducing the load on the server and the need for constant server requests. Additionally, the data was also stored in pg_tileserv for quicker access.

This change meant that the geometric transformation could be performed locally using JavaScript. This decreased the need for additional server requests and

significantly improved system performance, particularly when dealing with large datasets, thereby effectively addressing the limitations of the initial implementation.

5.4 Important Algorithms/Methods

5.4.1 Interaction Between FastAPI and PostgreSQL

The FastAPI server communicates with the PostgreSQL database using the psycopg2 library. To establish this connection, it uses the database access variables stored in the .env file. This approach ensures that these critical access details are not exposed, and also allows easy integration with online hosting solutions that support environment variables.

The project establishes two separate connections to handle the AIS and Persona databases. Here is a simplified snippet of the connection code:

```
con_ais = psycopg2.connect(
host=os.getenv("DB_HOST"),
database='[db_name (ais or persona)]',
user=os.getenv("DB_USER"),
password=os.getenv("DB_PASSWORD"),
port=os.getenv("DB_PORT")
)
```

It is important to note that this project is designed as a performance testing tool for moving objects using MobilityDB and Leaflet, and not for production use. Thus, it does not implement extensive error handling or security measures. Specifically, it is currently vulnerable to SQL injection attacks. While this is acceptable given the context and purpose of this project, any future iterations of this project intended for production use should include comprehensive error handling and appropriate security measures to protect against such attacks.

5.4.2 Usage of Pg_tileserv

The pg_tileserv is configured using specific settings corresponding to our local database. Key configurations are as follows:

- **DbPoolMaxConns:** Set to 50, under the assumption that we won't need more than 50 tiles at the same time.
- **DbTimeout:** Set to 30. This can be modified if some tiles take more time to load.
- **CacheTTL:** Set to 600. This enables a cache duration of 10 minutes. In a real case, I would set the cache to last for a day to remove some of the tiles if they are not used daily.
- **MaxFeaturesPerTile:** Set to -1 to ensure no information is lost. If we want to filter the number of features, it can be directly done on the SQL function called by pg_tileserv.

An SQL function is created on the PostgreSQL server to enable MVT generation. This function returns a `bytea` object containing the MVT information (See code A.1):

This function takes in four parameters: the tile coordinates `z`, `x`, `y`, and an optional `maxpoints` parameter with a default value of 10. The function returns a `bytea` object, which represents the tile data.

The function works in the following way:

- **Calculate the Tile Envelope:** It calculates the bounding box for the given tile coordinates using the `ST_TileEnvelope(z, x, y)::stbox` function.
- **Select Appropriate Trajectories:** It then selects the `id` and trajectory data of each point within the calculated bounding box from the 'persona_small_4000' table. The `asMVTGeom(fullday_trajectory, bounds.geom)` function is used to convert the `fullday_trajectory` column (a tgeompoint column) into MVT-compatible geometry. The `ORDER BY` clause sorts the data by `id`, and the `LIMIT maxpoints` clause limits the data to a maximum of `maxpoints` rows.
- **Construct MVT Geometry:** The `mvtgeom` CTE (Common Table Expression) then extracts the `tripid`, `geom` and `times` from the selected data.
- **Construct the MVT:** Finally, the `st_asmvt(mvtgeom, 'reduced')` function constructs the MVT from the `mvtgeom` data.

5.4.3 Data Fetching in React

GeoJSON

In the context of our React-based frontend, data fetching is initiated by a client-side request to the FastAPI backend server, targeting the `/geojson` endpoint. The parameters employed in the request include `limit`, which defines the maximum number of objects intended for display on the map, and `db_name`, which signifies the source database for the query, be it AIS or Persona.

Upon receiving the request from the frontend, the backend server responds by composing a corresponding SQL query targeted at the PostgreSQL server. The structure of this SQL command is as follows:

```
1 WITH trips AS (
2 SELECT {column_name} AS col
3 FROM {table_name}
4 {order_by} LIMIT %s
5 )
6 SELECT asMFJSON(col::json)
7 FROM trips LIMIT %s
```

In this command, `column_name` represents the column containing the `tgeompoint` type data, `table_name` stands for the specific table housing the data of interest, and `order_by` specifies the optional clause for data sorting. The `LIMIT` instruction in the SQL command is directly controlled by the `limit` parameter in the frontend request, providing a flexible and dynamic way to manipulate the amount of data retrieved and displayed.

The SQL command utilizes the `asMFJSON` function from MobilityDB, applied to the `tgeompoint` column, effectively transforming it into a GeoJSON. This operation embeds the coordinates along with their respective timestamps, thus enabling the spatial-temporal visualization capabilities of our frontend map. The combination of GeoJSON format and the frontend map rendering allows us to efficiently manage and display complex spatiotemporal data.

Mapbox Vector Tiles

The data fetching process for MVT introduces a slight increase in complexity compared to GeoJSON. In this case, the request initiation is tied to the loading of a new tile on the frontend map, facilitated by our `CustomVectorGrid` class. Each time a new tile is required for display, a request is sent to the `pg_tileserv` server, targeting the data specific to the tile in question.

Recognizing the frequent requirement for simultaneous loading of multiple tiles during interactive map navigation operations such as zooming or panning, our implementation utilizes a strategy of concurrent data fetching. This mechanism is handled by the parent VectorGrid class, which orchestrates parallel processing of the tile data requests. This approach not only significantly optimizes the tile loading operation but also ensures a smooth and efficient user experience by maintaining responsiveness of the map interface even during intensive navigation operations.

This MVT data fetching process, alongside our custom caching mechanism detailed in the preceding sections, forms the backbone of our implementation's capability to handle, manipulate, and visualize spatiotemporal datasets on the frontend map interface. This harmonized integration between frontend and backend components is a fundamental factor contributing to the project's successful management of complex geospatial data and user interaction scenarios.

5.4.4 Caching Mechanism

The frontend of the application employs a caching mechanism to expedite the loading of tiles. Given the large volume of data that needs to be displayed on the map, this caching strategy is integral to the application's performance. The caching process operates as follows:

Whenever the frontend requires a tile, it first checks the cache. If the requested tile is found in the cache, the tile data is immediately used, and the usage counter for that tile is incremented within the cache. If the tile is not present in the cache, the frontend sends a request to the `pg_tileserv` server for the tile data.

Upon receipt of the tile data from the `pg_tileserv` server, which is initially in Protobuf format, the frontend converts this data into JSON format. This converted JSON data is then primed for addition to the cache.

However, before the tile data is added to the cache, the frontend checks for sufficient space in the cache. If the cache is full, some space needs to be cleared. This is accomplished by deleting a tile from the cache, following a specific algorithm:

1. The frontend first checks if there are any tiles in the cache that are not at the current zoom level. If there are, it targets these tiles for possible deletion. If no such tiles are found, it considers tiles at the current zoom level for deletion.

2. From the identified subset of tiles (either not at the current zoom level or at the current zoom level), the tile with the least uses is selected for deletion.

Importantly, when a tile is deleted from the cache, only its JSON data is removed. The record of uses for each specific tile is preserved. This strategy ensures that a tile that has been frequently requested but, for some reason, was deleted, will still have a high score.

After making room in the cache, the new tile data is added, and its use counter is incremented.

5.4.5 Object Movement Handling

The handling of object movement varies depending on whether the data is represented as MVT or as GeoJSON.

MVT

In the case of MVT, we have extended the `vectorgrid.protobuf` class to overwrite some of the basic tile functions. The newly created class includes some attributes necessary to facilitate object movement, such as `_timestamp` (representing the current timestamp of the frontend) and `_jsons` (which stores the cached JSONs along with their usage counters).

Whenever the timestamp is updated using the `setTimestamp(timestamp)` function, the map is redrawn based on the following algorithm:

1. Retrieve the JSON associated with the tile, either from `pg_tileserv` or from the cache.

2. Extract all the features of the map at the current timestamp using the `extract_geom` function. Given that each point is linked to a specific timestamp, if the current timestamp does not match a timestamp for a particular point, we interpolate the x and y positions between the two closest timestamps.

3. Return the JSON with the updated features for each object. The default behaviour of `vectorgrid.protobuf` takes care of the rest.

GeoJSON

In the case of GeoJSON, the data are loaded only once, as it contains all the necessary information for the entire simulation. The data are represented in a single layer that includes all points.

Once the layer has finished loading, the next one is initiated based on the current timestamp of the simulation (which operates independently of the layer loading system). Just like the MVT implementation, the GeoJSON implementation uses interpolation to simulate object movement when the current timestamp doesn't directly correspond to a recorded timestamp.

5.5 Data Visualization

This section explores the methods used for data visualization in the project. Two distinct approaches were employed, namely MVT and GeoJson.

5.5.1 MVT

For MVT, the styling was achieved through the use of `vectorTileLayerStyles` property of VectorGrid. An example of this is:

```
vectorTileLayerStyles: {
  reduced: function (properties, zoom) {
    return {
      radius: properties.radius,
      weight: 2,
      fill: true,
      fillColor: "black",
      fillOpacity: 1,
      color: "black",
    };
  },
}
```

One of the major advantages of this method is the capability to employ a variety of styles, even across different tile layer names. It also allows for the addition of properties to the tile from the PostgreSQL database. This can be particularly useful for modifying the size or the color of the points based on the properties. For instance, if the requirement was to have points with the property `vehicle` equals to `car` in red color and the rest in black, it could easily be accomplished as shown:

```
fillColor: property.vehicle === "car" ? "red" : "black"
```

5.5.2 GeoJson

With GeoJson, the default LayerGroup from Leaflet is employed for data visualization. The features are displayed using the `l.Circle` class from Leaflet and added to the LayerGroup. Whenever a rerendering of the layer is required, the current layer is cleared and new circles are added for the specific timestamp.

The `l.Circle` class from Leaflet enables easy styling through its options, as demonstrated below:

```
L.circle([coordinates[1], coordinates[0]], {
  color: "black",
  fillColor: "black",
  fillOpacity: 1,
  radius: 10,
}).addTo(geojsonlayerRef.current);
```

This provides great flexibility and control over the visualization, making it possible to tailor the styling to specific requirements.

Chapter 6

Testing and Evaluation

This chapter is dedicated to providing a detailed account of the rigorous testing and evaluation undertaken to ensure the proper functioning, reliability, and efficiency of the developed system, with a focus on the two primary implementations—MVT and GeoJSON. Performance evaluation forms the cornerstone of software development, offering vital insights into the system's functionality under different circumstances, and is particularly crucial in the context of this project, where the goal extends beyond merely developing a functional system to discerning the more efficient and effective implementation.

An array of performance metrics have been used to compare the two implementations, including query execution time, average updates per second under standard conditions, as well as during interactive operations such as zooming and panning, and memory usage. These metrics give us a comprehensive picture of each implementation's performance, which further aids in identifying the best-suited solution for different use-cases.

The testing process has been conducted on two different datasets, with varying time windows for the Automatic Identification System (AIS) dataset (1, 2, and 7 days). This use of different datasets not only tests the system's robustness and scalability but also provides a broader understanding of its performance in varying data contexts. Through this chapter, we aim to offer an in-depth understanding of the system's performance, offering key insights that could shape its future iterations and refinements.

6.1 Methodology

6.1.1 Test Environment

The testing for this project was carried out in an environment consistent with the one described in the Implementation chapter, ensuring a realistic representation of system performance during development. The hardware and software configurations are identical, providing a consistent baseline for understanding the behavior and the performance of the application. Both the backend and the frontend applications were executed simultaneously on the same machine, thereby reflecting a real-world scenario where server and client processes would potentially share system resources.

The testing environment is identical to the previously outlined development environment. The consistency between the development and testing environments ensures a more accurate representation of the system's performance characteristics and potential bottlenecks, thereby providing a more robust and reliable evaluation.

6.1.2 Datasets

The two datasets used for testing and evaluation in this project are the `Persona` dataset and the `AIS` dataset.

Persona Dataset

The comprehensive Persona dataset, notable for its large volume of entries, constitutes a significant resource in this project. Nonetheless, to facilitate a more manageable testing and evaluation process, a scaled-down version of this data, aptly named the "Persona Small", was employed. The "Persona Small" subset contains a carefully selected 100,000 entries from the original dataset. This selection presents a diverse yet manageable amount of data, allowing for a nuanced and efficient performance analysis.

The Persona dataset is provided in the form of a SQL script that seamlessly combines the creation of the requisite database structure and the insertion of all pertinent records. This integrated approach simplifies the preparation process for testing. The Persona dataset spans a full day's worth of intricate travel data, ensuring a broad representation of realistic user travel patterns and behaviors.

However, given the substantial volume of the data involved, further optimizations were pursued to augment the performance of requests. Specifically, a materialized view was constructed based on a subset of 4000 Persona records. This strategic decision, aimed at enhancing the efficiency of data retrieval, resulted in a significant increase in the speed of requests. In a similar vein, recognizing the high frequency of access to the "fullday_trajectory" column in our test scenarios, an index was created on this column. This indexing strategy further expedited data retrieval, resulting in overall improved system performance.

AIS Dataset

The AIS dataset comprises records of numerous ship movements. This data is freely available online, and for this project, a subset of this data was filtered based on geographical position. Only the data for ships within the box defined by -16.1 $<$ longitude $<$ 32.88 and 40.18 $<$ latitude $<$ 84.17 was included. After applying this filter, a total of 14,166 ship records were available for testing.

For each day, a separate CSV file was generated, containing all positions of the different ships, each uniquely identified by their Maritime Mobile Service Identity (MMSI). Seven consecutive days of data were chosen for this project, specifically from the 23rd to the 29th of May 2023. For the one-day tests, data from the first day (23rd May) were used, and for the two-day tests, data from the first two days (23rd and 24th May) were used.

The quantity of ships represented within the database is not a static number, but instead fluctuates based on the temporal scope of the data under consideration.

More specifically, this quantity alters with the varying duration in days for which the data has been accumulated. To illustrate this concept, here are the quantities of ships recorded for different durations:

- **1 Day:** 4865 ships.
- **2 Days:** 6501 ships
- **7 Days:** 14166 ships

It is due to this variation in ship counts based on the temporal scope of the dataset, that we are unable to present results for 10000 objects in the case of the 1-day and 2-day sub-datasets. In these instances, the total number of ships recorded is less than the desired count of 10000. For the 1-day sub-dataset in particular, where the ship count totals 4865, the results derived from the complete dataset are represented in the 5000 objects data column.

Datasets Comparison

The Persona and AIS datasets serve as substantial sources for testing the visualization tool, each offering unique characteristics and challenges. A comparison of the main features of these datasets is illustrated in Table 6.1.

The Persona dataset, sourced from the Syty.io Project, provides a larger volume of synthetic data entries. Its SQL script delivery format simplifies integration, and a materialized view and index were employed for optimization. Conversely, the AIS dataset contains detailed ship movement records, available freely online, with a more complex CSV per day delivery format. The data's manageability and relevance were ensured by filtering based on geographical position.

	Persona Dataset	**AIS Dataset**
Data Type	Synthetic data (trips)	Ship movement records
Source	Syty.io Project	Online, freely available
Size	100,000 entries	14,166 ship records
Delivery Format	SQL script	CSV per day
Performance Optimization	Materialized view and index	Filtered based on geographical position

Table 6.1: Comparison of Persona and AIS datasets

6.1.3 Test Implementation

Testing was carried out using a variety of measures designed to accurately gauge the performance of the two implementations under different usage scenarios. Each of the following subsections detail the procedure undertaken to test each performance metric:

Loading Times

The loading times were assessed differently for the MVT and GeoJSON implementations, considering the way the data is fetched in each case.

For the MVT implementation, a single tile (with a zoom level of 8, x=139, and y=81) was queried repeatedly. On the other hand, the GeoJSON implementation entailed the retrieval of the entire GeoJSON. To get a comprehensive understanding of the time complexity of the solutions, the tests were performed at different data volume levels: 100, 250, 500, 1000, 2000, 5000, and 10000 objects. An object represents any moving entity. In our case, these could be cars or boats

The testing procedure involved the following steps:

- Run the query five times without recording the time to allow for PostgreSQL caching, which could enhance the query performance.
- Execute the same query 25 more times, this time recording the time taken for each execution.
- Compute the average of the recorded times to arrive at the final average loading time.

This methodology helped in determining the loading times for each implementation and provided insight into how the size of the data impacts this metric.

To streamline this process, two specific functions, namely `average_exec_time_mvt` (See Appendix A.3) and `average_exec_time_json`(See Appendix A.2), were developed. These functions take the total number of runs as an argument. Additionally, for the `average_exec_time_mvt` function, we can specify the z, x, and y coordinates of the tile we wish to test. This allows for the efficient execution of tests, saving time and effort while ensuring accuracy.

Average Updates Per Second

To carry out this test, the simulation was run for a set duration of five minutes. This duration was chosen to provide a significant enough time frame to observe the performance trends and ensure the stabilization of the average update rate.

It is important to note that the choice of the five-minute window is not arbitrary. Preliminary runs of the simulation indicated that the average update rate tends to stabilize after approximately 30 seconds. Despite this early stabilization, a longer duration was still selected for the test. This is to account for any potential fluctuations that might occur in the performance over time, ensuring a more comprehensive evaluation of each implementation.

The test involves visualizing the entire viewport and then starting the simulation. The viewport is filled with points to ensure that all tiles (in the case of MVT) have something to load, pushing the implementations to their maximum capacity.

This rigorous testing approach serves to stress-test the implementations and provides a measure of their robustness and reliability in a real-world scenario of continuously updating data.

Average Updates Per Second While Moving

The movements during this test were deliberately made arbitrary, encompassing a variety of actions such as zooming and panning. This design choice was made to best mimic the diverse range of actions a typical user might perform. Moreover, this method provides a robust measure of how well the implementations handle the complexity introduced by the simultaneous management of data updates and user interactions.

The test was performed by initiating the simulation, and then introducing arbitrary movements on the map. Throughout this testing phase, which lasted five minutes, both zooming and panning actions were performed randomly, intending to closely emulate a user's random interaction with the application. By incorporating these arbitrary movements in the testing methodology, we are able to gain a more holistic understanding of the performance and reliability of the MVT and GeoJSON implementations in a realistic, interactive context.

Upon further investigation, we determined that panning and zooming does not have any impact on the GeoJSON implementation. This is because the implementation does not require loading additional data while the webpage is active. Unlike the MVT implementation that loads new tiles every time there is a movement or zoom action. Based on this finding, we made the decision to exclude this specific test from the GeoJSON implementation.

Memory Usage

The tests were performed under the "moving" scenario. This scenario was chosen as it presents a more realistic and challenging test for the implementations. By continuously updating data while handling user interactions, the implementations are stressed, thereby providing a more comprehensive view of their memory efficiency under typical usage conditions.

The Google Chrome browser's development tools were utilized to accurately measure the memory consumption of the webpage. The measurements were taken after a period of five minutes of continuous use, which allows any potential memory leakages or inefficiencies to become more pronounced.

It is important to note that these measurements reflect not only the memory efficiency of the data delivery method (MVT or GeoJSON) but also the efficiency of the frontend code in processing and rendering this data. Therefore, this metric provides valuable insight into the overall performance and resource management of the implementations as a whole.

With these detailed testing procedures, it was possible to obtain a comprehensive performance evaluation of the MVT and GeoJSON implementations under various scenarios, providing valuable insights into their strengths and potential areas for optimization.

6.1.4 Evaluation Metrics

A set of comprehensive evaluation metrics were utilized to scrutinize and contrast the performance of the MVT and GeoJSON implementations. These metrics were

deliberately selected due to their relevance to user experience and system efficiency in actual application scenarios.

Loading Times This metric is a critical aspect of user experience as they directly influence the perceived speed and responsiveness of the application. In interactive applications that involve map visualization, rapid loading times are essential for a smooth user experience. The speed with which the system can fetch and render data can significantly affect the user's ability to interact with the map, and delays can be disruptive. Hence, the loading times of the MVT and GeoJSON implementations were measured under various data volume levels to provide a comprehensive assessment of their performance in different scenarios.

Average Updates Per Second This metric gives an indication of the system's capability to handle a continuous stream of data updates. In applications that deal with real-time or dynamic data, it is important for the system to be able to process and reflect changes in the data at a high rate. As such, the average updates per second metric provides an insight into how well the MVT and GeoJSON implementations can manage a constant flow of updates, and thus, their suitability for real-time applications.

Average Updates Per Second While Moving This metric takes into account user interaction with the application in the form of arbitrary movements on the map, providing a more realistic measure of system performance. In real-world scenarios, users would be expected to interact with the map by panning and zooming while the data updates occur. Therefore, this metric gauges how well the MVT and GeoJSON implementations manage the complexity introduced by the simultaneous occurrence of data updates and user interactions.

Memory Usage Efficient usage of system memory is crucial for overall system performance, especially in scenarios where resources may be limited. Excessive memory consumption can lead to slow response times and can even cause the application to crash in extreme cases. By monitoring the memory consumption of the MVT and GeoJSON implementations during continuous use, potential memory leaks or inefficiencies can be identified, and the overall memory efficiency of the implementations can be evaluated.

Taken together, these metrics provide a comprehensive and multifaceted evaluation of system performance under various conditions, enabling a thorough comparative analysis of the MVT and GeoJSON implementations in the context of map visualization applications.

6.1.5 Test Scenarios

In order to derive a well-rounded and precise evaluation of the performance attributes for both the MVT and GeoJSON implementations, a series of varied test scenarios were thoughtfully designed. The principal aim behind these was to simulate a wide array of realistic user interactions and data volumes, which would

correspondingly yield an authentic and relevant performance assessment. An elaborate discussion on these devised scenarios is as follows:

Low Data Volume This scenario primarily aimed to gauge the system's efficiency with a smaller volume of data, specifically encompassing 100 and 250 objects. The tests conducted under this scenario endeavored to investigate the performance of both implementations under circumstances that could be characterized as more routine or 'lighter' usage conditions. Representing a typical situation for numerous users, this test scenario underscores that not all instances of usage necessitate heavy data interactions, thus making it a crucial aspect of this evaluation.

Medium Data Volume This scenario was crafted with the intention of scrutinizing the system's performance under circumstances involving a moderate volume of data. To achieve this, the tests were meticulously carried out with 500 and 1000 objects. This testing paradigm embodies a more demanding usage situation, one that might be encountered by users involved in intricate tasks with the application or dealing with a broader set of data points.

High Data Volume In the quest to evaluate the performance of the system under significant stress, the high data volume scenario was established. For this, the implementations were tested with as many as 2000 objects. This rigorous test scenario was designed to emulate a 'heavy' usage condition, one which would necessitate a robust and efficient mechanism for data handling and processing from the system, thereby testing its true capabilities.

Extreme Data Volume With the intention of pushing the system to its maximum capacity, the "Extreme Data Volume" scenario was included in the testing scheme. This scenario involved measuring the loading times of sizable datasets, specifically comprising of 5000 and 10000 objects. Initially considered to be too daunting to be conducted under 'real conditions', due to the anticipated high loading times, this scenario was ultimately carried out in a real environment. The substantial volume of data tested in this scenario presented a rigorous evaluation of the system's ability to handle extreme loads. Additionally, it allowed us to assess the scalability of the system by observing the correlation between increased data volumes and the corresponding loading time. This exercise provided us with invaluable insights into the performance limits of both MVT and GeoJSON implementations, guiding the discussion on the trade-off between data volume and system efficiency.

Each of these scenarios was meticulously designed with the main objective of conducting a comprehensive evaluation of the two implementations. The aim was to gain a deep understanding of their performance characteristics and behavior under a diverse range of loads and usage conditions, thereby providing valuable insights into their applicability and suitability in various real-world situations.

6.1.6 Limitations of the Methodology

While the methodology employed in this study provides a comprehensive understanding of the performance of both MVT and GeoJSON implementations, it is important to consider certain limitations. These limitations primarily arise from constraints tied to the testing environment, the nature of the datasets used, and the specificity of the scope of the test.

Dataset Representation The datasets used in this study, namely Persona and AIS, were carefully chosen to be representative of a broad range of real-world scenarios. The AIS dataset was further divided into three subsets of varying lengths (1 day, 2 days, and 7 days), providing an insight into performance across different data volumes. However, it is important to note that these datasets may not fully encapsulate the vast variety of real-world data. Certain edge cases, particularly those involving more complex or significantly larger data structures, may yield different results. The performance metrics obtained in this study should therefore be generalized to other datasets with due consideration of the differences in data structure and volume.

Testing Environment The testing environment utilized in this study is significantly simplified compared to a typical production environment. In the tests, the database, backend, and frontend were all running on the same machine. This arrangement eliminates factors such as network latency and inter-process communication, which could potentially impact performance in a distributed setup. Additionally, the computing power of the machine used for testing may not match up to the high-performance servers used in production environments. Therefore, it is reasonable to expect that the observed performance could vary, potentially significantly, in a real-world production environment.

Performance Metrics The performance metrics selected for this study aim to offer a detailed evaluation of both static and dynamic aspects of performance, considering both loading times and interactive usage scenarios. However, these metrics may not fully capture all aspects of real-world performance. In particular, they do not account for how performance may evolve over extended periods of operation, nor how it may be affected by sudden, large-scale changes in the volume of data updates. Future studies may need to consider additional performance metrics or testing scenarios to cover these aspects.

Despite these limitations, the study provides valuable insights into the performance of MVT and GeoJSON implementations. These limitations should be considered as areas for future refinement, particularly as the system transitions towards a real-world, production environment.

6.2 Results

6.2.1 Loading Time Testing

In the following subsection, we present the results of the loading time testing performed on different subsets of data. For the AIS (1 day) dataset, we included the results of 4865 objects in the 5000 objects column due to the limitation in the number of objects available for the test.

Mapbox Vector Tiles

The Table 6.2 presents the results for each dataset using MVT.

Objects	Persona_small (1 day)	AIS (1 day)	AIS (2 days)	AIS (7 days)
100	46	250	374	383
250	108	339	582	555
500	216	430	770	1373
1000	437	551	1030	1872
2000	880	1222	2121	2806
5000	1757	3796	5952	7822
10000	1746			19120

Table 6.2: Tile loading time per dataset using MVT (ms)

GeoJSON

The Table 6.3 presents the results for each dataset using GeoJSON.

Objects	Persona_small (1 day)	AIS (1 day)	AIS (2 days)	AIS (7 days)
100	25	169	256	213
250	57	376	497	689
500	112	766	1001	1504
1000	352	1453	2208	2876
2000	862	1961	3757	6106
5000	2191	3325	5681	13772
10000	4358	3325		18367

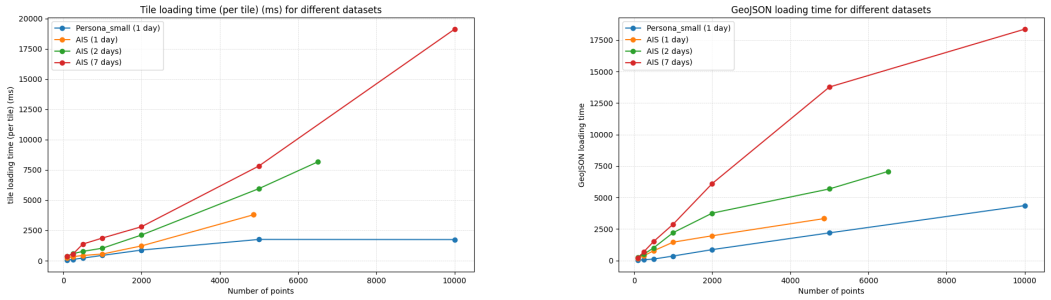
Table 6.3: GeoJSON loading time per dataset (ms)

Comparison

Persona Dataset The Table 6.4 compares the loading times using MVT and GeoJSON for the Persona dataset.

AIS (7 days) Dataset The Table 6.5 presents the results for each dataset using GeoJSON.

Upon comparing the MVT and GeoJSON methods across different datasets(see Figure 6.1), we can infer that each approach possesses its distinct set of advantages



(a) MVT Implementation (b) GeoJSON Implementation

Figure 6.1: Comparison of MVT and GeoJSON Implementations

Objects	MVT (ms)	GeoJSON (ms)
100	46	25
250	108	57
500	216	112
1000	437	352
2000	880	862
5000	1757	2191
10000	1746	4358

Table 6.4: Comparison of MVT and GeoJSON for Persona Dataset

and trade-offs. The choice between the two is largely governed by the specifics of the use-case scenario and the volume of data involved.

The MVT approach elucidates that the loading times, representing query execution times, increase linearly with the number of objects. In this setup, the server carries the majority of the processing load, converting the data into a format that the client can parse swiftly. This method exhibits efficiency when dealing with a relatively smaller number of objects (i.e., in the range of 100 to 2000 objects). However, when the number of objects escalates (5000 and more), the loading time correspondingly surges, rendering MVT less optimal for larger datasets.

Contrastingly, the GeoJSON approach, despite the longer loading times for smaller datasets (100 to 1000 objects), demonstrates superior performance with larger datasets (2000 objects and more). This could be attributed to GeoJSON transferring the entire data set to the client in one go, leading to longer initial load times, but fewer overall requests. This makes it more suitable for scenarios where the complete dataset needs to be loaded and manipulated on the client-side.

6.2.2 Updates per Second Testing

This section provides a detailed analysis of the update per second (ups) metrics, contrasting the MVT and GeoJSON implementations. This performance indicator is essential as it quantifies the system's responsiveness to user interaction or data alterations.

Objects	MVT (ms)	GeoJSON (ms)
100	383	213
250	555	689
500	1373	1504
1000	1872	2876
2000	2806	6106
5000	7822	13772
10000	19120	18367

Table 6.5: Comparison of MVT and GeoJSON for AIS (7 days) Dataset

MVT

The Table 6.6 presents the results for each dataset using the MVT approach.

Objects	**Persona (1 day)**	**AIS (1 day)**	**AIS (2 days)**	**AIS (7 days)**
100	96 ups	100 ups	102 ups	104 ups
250	68 ups	81 ups	72 ups	71 ups
500	44 ups	61 ups	54 ups	45 ups
1000	27 ups	44 ups	38 ups	29 ups
2000	16 ups	27 ups	23 ups	18 ups
5000	9 ups	12 ups	10 ups	8 ups
10000	5 ups	12 ups	-	4 ups

Table 6.6: Updates per second for MVT

GeoJSON

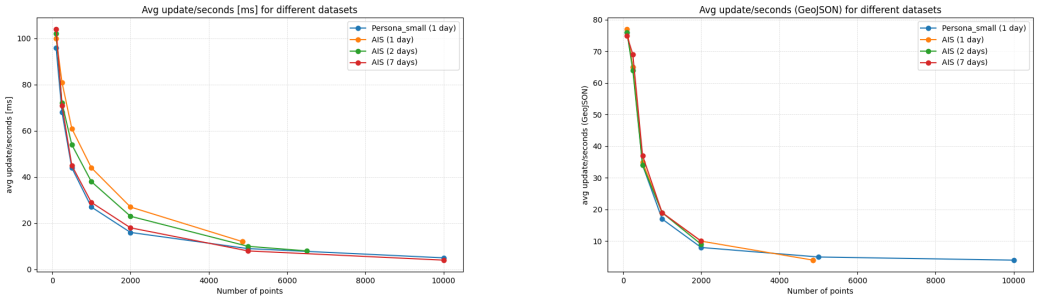
The Table 6.7 presents the results for each dataset using the GeoJSON approach.

Objects	**Persona (1 day)**	**AIS (1 day)**	**AIS (2 days)**	**AIS (7 days)**
100	76 ups	77 ups	76 ups	75 ups
250	65 ups	65 ups	64 ups	69 ups
500	34 ups	35 ups	34 ups	37 ups
1000	17 ups	19 ups	19 ups	19 ups
2000	8 ups	10 ups	9 ups	10 ups
5000	5 ups	4 ups	-	-
10000	4 ups	-	-	-

Table 6.7: Updates per second for GeoJSON

Comparison

Persona Dataset The Table 6.8 presents a side-by-side comparison of the MVT and GeoJSON implementations using the Persona dataset.



(a) MVT Implementation (b) GeoJSON Implementation

Figure 6.2: Comparison of MVT and GeoJSON Implementations

Objects	MVT	GeoJSON
100	96 ups	76 ups
250	68 ups	65 ups
500	44 ups	34 ups
1000	27 ups	17 ups
2000	16 ups	8 ups
5000	9 ups	5 ups
10000	5 ups	4 ups

Table 6.8: Comparison of update times for Persona dataset

AIS (7 days) Dataset The Table 6.9 presents a side-by-side comparison of the MVT and GeoJSON implementations using the AIS (7 days) dataset.

Comparing the MVT and GeoJSON approaches for updating objects (see Figure 6.2) shows that both have their strengths and weaknesses. For smaller datasets (e.g., Persona and AIS (1 day)), MVT tends to provide faster updates per second. As the number of objects increases, however, the MVT update speed starts to decrease, making it less suitable for larger datasets. In contrast, GeoJSON shows a consistent performance across different datasets, regardless of their size.

For larger datasets (e.g., AIS (7 days)), GeoJSON outperforms MVT, maintaining a relatively stable update speed. Although GeoJSON takes more time for initial data loading (as observed in the previous section), it handles data updates more efficiently for larger datasets.

One major limitation of the GeoJSON approach, however, is its inability to process extremely large datasets (as observed in the 5000 and 10000 objects tests). This is a critical point to consider, particularly for applications dealing with substantial amounts of geospatial data.

To summarize, the MVT approach seems to be more suitable for applications dealing with smaller datasets and requiring faster updates, while GeoJSON could be a better choice for scenarios involving larger datasets, providing that the data size does not exceed its processing capacity. In this specific case, the MVT approach could be used even if the number of updates per second is really slow.

Objects	MVT	GeoJSON
100	104 ups	75 ups
250	71 ups	69 ups
500	45 ups	37 ups
1000	29 ups	19 ups
2000	18 ups	10 ups
5000	8 ups	-
10000	4 ups	-

Table 6.9: Comparison of update times for AIS (7 days) dataset

6.2.3 Updates per Second while Playing Testing

In this section, we compare the performance of MVT and GeoJSON methods in terms of the average time taken to update objects while playing the data on a map.

MVT

For the MVT method, as seen in the Table 6.10, the average update per second while playing varies with different datasets and number of objects.

Objects	Persona (1 day)	AIS (1 day)	AIS (2 days)	AIS (7 days)
100	96 ups	100 ups	102 ups	104 ups
250	68 ups	81 ups	72 ups	71 ups
500	44 ups	61 ups	54 ups	45 ups
1000	27 ups	44 ups	38 ups	29 ups
2000	16 ups	27 ups	23 ups	18 ups
5000	9 ups	12 ups	10 ups	8 ups
10000	5 ups	12 ups	-	4 ups

Table 6.10: Average updates per second while playing for MVT

GeoJSON

For the GeoJSON method, no further requests are made after the initial data load. As a result, moving on the map does not require any additional requests to the server, and thus, the average update per second while playing remains the same as that when not playing. Consequently, there are no new data to present in a table for this approach in this particular case.

Comparison

Due to the differences in the operation of MVT and GeoJSON methods, a direct comparison of their performances while playing is not feasible. However, it is important to note that MVT might perform better in scenarios that require real-time updates and more interactivity, while GeoJSON would be better suited for scenarios where the entire data needs to be loaded at once, and the data size is not a concern.

Comparing the results of MVT while playing and not playing shows that(see Figure 6.3), as expected, the average update time per second is slightly higher when the map is moving. This can be explained by the fact that when the map is moving, new tiles may come into view, and thus additional data must be fetched and rendered.

The Table 6.11 illustrates the comparison of average update times per second while playing versus not playing for the MVT approach:

Objects	MVT Playing	MVT Not Playing
100	100 ups	74 ups
250	81 ups	60 ups
500	61 ups	46 ups
1000	44 ups	34 ups
2000	27 ups	24 ups
5000	12 ups	10 ups
10000	12 ups	10 ups

Table 6.11: Comparison of average update times per second for MVT while playing versus not playing

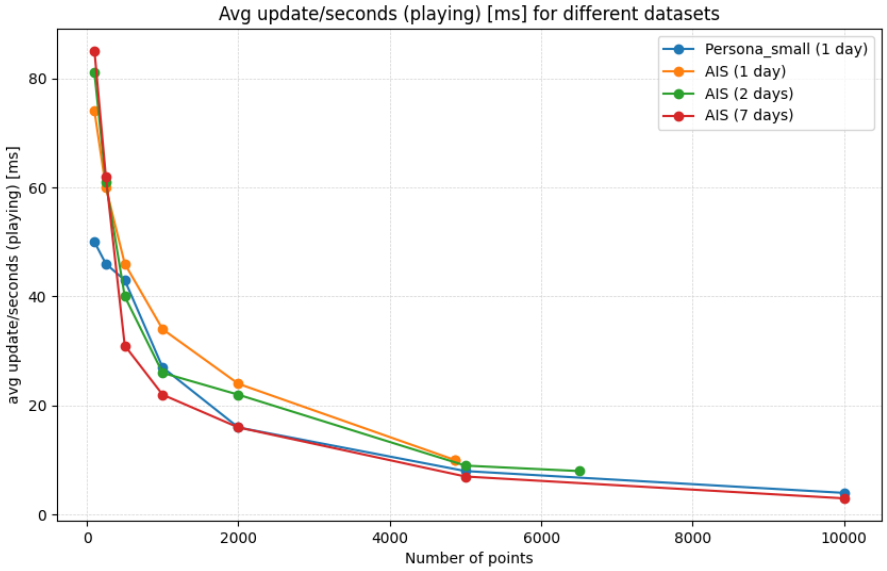


Figure 6.3: Average updates/second (Playing)

These results(see Figure 6.3) provide additional insights into the performance characteristics of the MVT approach. In scenarios where real-time updates and interactivity are important, the slight increase in update time when the map is moving could potentially impact user experience. However, given that the differences are generally small, this impact might be negligible for most use cases.

6.2.4 Memory Usage Testing

This section provides the results for the memory usage of each approach during the data visualization process.

MVT

The Table 6.12 presents the results for each datasets using the MVT approach.

Objects	Persona_small	AIS (7 days)
100	15	20
250	15	30
500	15	80
1000	15	135
2000	50	525
5000	100	-
10000	200	-

Table 6.12: GeoJSON memory usage per dataset (MB)

GeoJSON

The Table 6.13 presents the results for each dataset using the GeoJSON approach.

Objects	Persona_small (1 day)	AIS (1 day)	AIS (2 days)	AIS (7 days)
100	20	47	55	40
250	25	64	60	63
500	30	66	91	185
1000	40	91	135	276
2000	60	286	377	667
5000	134	695	-	-
10000	257	-	-	-

Table 6.13: GeoJSON memory usage per dataset (MB)

Comparison

Persona Dataset: The Table 6.14 presents a comparison of the memory usage results for the MVT and GeoJSON implementations using the Persona dataset.

For the Persona dataset, it is evident that the MVT approach uses less memory compared to the GeoJSON approach across all numbers of objects. This suggests that MVT could be a more efficient choice for this dataset, especially for larger numbers of objects where the difference in memory usage becomes more pronounced.

Objects	MVT	GeoJSON
100	15	20
250	15	25
500	15	30
1000	15	40
2000	50	60
5000	100	134
10000	200	257

Table 6.14: Comparison of MVT and GeoJSON memory usage for Persona dataset (MB)

Objects	MVT	GeoJSON
100	20	40
250	30	63
500	80	185
1000	135	276
2000	525	667

Table 6.15: Comparison of MVT and GeoJSON memory usage for AIS (7 days) dataset (MB)

AIS (7 days) Dataset The Table 6.15 presents a comparison of the memory usage results for the MVT and GeoJSON implementations using the AIS (7 days) dataset.

As for the AIS (7 days) dataset, both MVT and GeoJSON implementations show a similar trend in terms of memory usage with an increase in the number of objects. However, the GeoJSON approach takes more memory than MVT, particularly as the number of objects increases, which may limit its usability for larger datasets.

In the case of memory usage, the GeoJSON approach tends to have higher usage than MVT. This is expected given that GeoJSON sends all the data to the client-side in one go, thus taking up more memory. In contrast, MVT, which processes data server-side and only sends necessary data to the client, shows significantly less memory usage especially with smaller datasets. However, as the number of objects increase, the memory usage of MVT also increases, albeit at a slower rate compared to GeoJSON.

6.3 Comparitative Analysis

Loading Time For smaller datasets (100 and 2000 objects), MVT demonstrates superior performance in loading time. However, with larger datasets (5000 and 10000 objects), GeoJSON outperforms MVT. This suggests a dichotomy in performance where MVT is more efficient for smaller data sizes while GeoJSON proves to be more proficient for larger datasets, indicating that the size of the dataset is an essential factor in choosing the appropriate format.

Updates per Second - Not Moving Considering the number of updates per second when the map is static, MVT performs better across all tested datasets. This demonstrates the capability of MVT to provide a more up-to-date visualization in scenarios where the map is not undergoing changes in position or scale, thus ensuring a constantly refreshed user view even when static.

Updates per Second - Moving In the case of updates per second while the map is moving, MVT shows superior performance with smaller datasets. However, GeoJSON performs better when handling larger datasets. This underlines a clear division in performance based on data size and map movement, showing the strengths of MVT with smaller datasets and GeoJSON with larger ones during map interactions.

Memory Usage Throughout the testing, MVT consistently used less memory compared to GeoJSON, regardless of the dataset size. This implies that MVT could potentially be a more resource-efficient choice when memory capacity is a significant consideration, a crucial aspect for devices with limited memory resources.

Evaluation of MVT and GeoJSON using Different Datasets Both MVT and GeoJSON were effective in managing various datasets, including the Persona dataset with less points per trajectory and the AIS dataset with significantly more points per trajectory. This demonstrates the versatility and wide-ranging applicability of both MVT and GeoJSON, capable of handling datasets of different complexities without requiring specific adjustments.

Influence of Map Size on MVT Performance An important observation is the significant influence of map size on MVT's performance. A larger map size means more tiles, leading to more requests to the backend server and PostgreSQL database, possibly resulting in longer loading times. GeoJSON, which loads all data at once, does not exhibit this issue, suggesting that the choice between MVT and GeoJSON could also depend on the anticipated map size.

Limitations One of the identified limitations is the non-optimization of the Leaflet mapping library for animations or moving objects, affecting both MVT and GeoJSON's performance when handling a large number of objects (5000 and above). This limitation may influence the choice of mapping library depending on the anticipated dynamics of the data.

Concluding Remarks The comparative analysis provides insights into MVT and GeoJSON's performance characteristics, highlighting their suitability for different dataset sizes and conditions. Moreover, it underlines the impact of map size on MVT's performance and identifies limitations due to non-optimization of the Leaflet mapping library. Thus, the choice between MVT and GeoJSON would depend on the specific use-case, available resources, and dataset characteristics, further explored in the following "Discussion" chapter.

Chapter 7

Discussion and Future Work

7.1 Performance Analysis

In this section, we discuss the performance results obtained from the evaluation of our tool. The main focus is on the comparison between the two data representation formats, MVT and GeoJSON, and the impact of the LFU caching strategy on the MVT implementation.

7.1.1 Efficiency of MVT and GeoJSON Implementations

The experimental results confirmed the anticipated benefits and trade-offs of the two data representation formats. On the one hand, the MVT implementation, due to its compact nature and the use of binary encoding, exhibited superior performance in terms of data transmission and rendering time, especially for large datasets. MVT's efficiency becomes particularly evident when visualizing a high number of moving objects, as it maintains a stable loading time up to a certain threshold. However, the performance decreases significantly when visualizing extremely large datasets (e.g., 10,000 moving objects), highlighting the limitations of the MVT implementation and Leaflet's capabilities.

On the other hand, the GeoJSON implementation proved to be more accessible and straightforward to use, given its simplicity and compatibility with various backends, including FastAPI in Python. However, GeoJSON's efficiency dropped rapidly as the size of the dataset increased, thus making it less suitable for large scale moving object visualization. The inherent nature of GeoJSON encoding, which results in larger file sizes compared to MVT, was a limiting factor for its performance, particularly concerning data transfer and rendering times.

7.1.2 Impact of LFU Caching

The LFU caching system played a critical role in the MVT implementation, significantly improving the database's performance by minimizing the number of requests to the database. The original MVT implementation involved making a database request for each timestamp to retrieve the corresponding data, which put a considerable strain on the database as the volume of data increased.

The LFU caching strategy effectively mitigated this issue. After the initial DB request for a specific tile, subsequent requests for the same tile could be served

from the cache, eliminating the need for additional DB requests unless the tile was removed from the cache. This strategy not only reduced the load on the database but also resulted in faster loading times and more efficient data management, enhancing the overall performance of the MVT implementation.

7.1.3 Performance Trends and Observations

The performance trends observed during the evaluation reaffirmed the distinct characteristics and suitability of the MVT and GeoJSON formats for specific use-cases. While MVT emerged as a more efficient format for large-scale moving object visualization, its performance benefits come at the cost of complexity and the necessity of a tile server. In contrast, GeoJSON, while less efficient for larger datasets, offers greater simplicity and flexibility, which could make it a preferred choice for smaller-scale applications or in situations where setting up a tile server is not feasible.

In conclusion, the performance analysis highlights the importance of considering the specific requirements and constraints of a given application when choosing the data representation format and the caching strategy. Future work can further explore this aspect, potentially investigating other data formats or caching strategies to improve the visualization tool's scalability and performance.

7.2 Comparison with Deck.gl Solution

In this section, we draw a performance comparison between our visualization tool and the implementation by Fabrício Ferreira da Silva. Both implementations leverage MVT and GeoJSON formats for data representation. Fabrício's approach utilizes synthetic trajectories data derived from the BerlinMod data generator, which creates trajectories based on residential, working, and shopping regions, simulating typical human movements on workdays and weekends. The resulting dataset generally includes up to 2000 vehicles spanning 28 days, with an average of 15560 points per vehicle at a scale factor of one.

Fabrício's tests comprised datasets at different scale factors: 0.005, 0.05, 0.2, and 1.0. For the sake of comparison, we will primarily discuss the 0.2 scale factor dataset, which provides data for 894 vehicles over 13 days, with an average of 11477 points per vehicle.

7.2.1 Comparison of Frame Rates

The Table 7.1 presents the frame rates (fps) achieved by Fabrício's MVT and GeoJSON implementations across different scale factors:

Scale Factor	MVT fps	GeoJSON fps
0.005	28.8	30
0.05	21.6	31.2
0.2	9	29.4

Table 7.1: Average number of FPS for Fabrício Ferreira da Silva's Solution

Compared to our tool, Fabrício's MVT implementation appears to be less efficient with smaller datasets, yielding lower fps rates. Similar trends are observed with the GeoJSON implementation. However, his GeoJSON solution manages to maintain a relatively constant fps across different scale factors, unlike our tool. This discrepancy could be due to the lack of built-in support for moving objects in Leaflet, resulting in performance variations with custom implementations.

7.2.2 Comparison of Loading Times

The loading times for Fabrício's solution were notably longer across all scale factors. The Table 7.2 details the loading times for different scale factors and zoom levels:

Scale Factor	Zoom Level	MVT Loading Time (seconds)
0.005	11	7.64
0.005	15	5.91
0.05	11	44.76
0.05	15	32.21
0.2	11	528
0.2	15	192

Table 7.2: Average loading time for Fabrício Ferreira da Silva's Solution

Even with our largest dataset, the loading times for both MVT and GeoJSON implementations did not exceed 20 seconds. Fabrício's loading times, however, were considerably higher, especially for the 0.2 scale factor dataset. This can be attributed to the denser nature of the datasets used by Fabrício, which contain a greater number of points per vehicle.

Interestingly, our tool outperformed Fabrício's even with comparable or greater vehicle density, particularly for the 0.005 and 0.05 scale factors. This performance edge indicates that our tool is capable of efficiently handling a larger number of vehicles.

It is important to note that these comparisons were made under different hardware conditions. Fabrício's experimental evaluation was conducted on a 'MacBook Pro 2020, equipped with a 2 GHz Quad-Core i5 processor, 16 GB of RAM, and a 1.5GB GPU. The MobilityDB was executed in a Docker container allocated with 4 CPUs and a maximum of 8.0 GB memory' [23]. In contrast, our system was deployed on more powerful hardware, which could contribute to the observed performance advantage.

In conclusion, although both implementations have their own strengths and weaknesses, our visualization tool shows encouraging results in terms of both frame rates and loading times. The tool is particularly adept at handling larger and denser datasets, demonstrating its robustness and scalability. It also suggests that the efficiency of trajectory visualization can be significantly enhanced by deploying the system on more capable hardware.

7.3 Reflection on Objectives

Implement a Solution to Visualize Large Volumes of Data This objective was partially met. The implementation does enable visualization of large volumes of AIS and Persona data, but performance issues arise when dealing with more than 10,000 vehicles simultaneously. This limitation appears to stem from the inherent restrictions of the Leaflet library, which is not initially designed for moving objects. Despite these challenges, the system demonstrates satisfactory performance for smaller scales, indicating that the design and implementation are sound, with room for improvements in scalability.

Caching Strategy Evaluation The implementation and evaluation of the Least Frequently Used (LFU) caching strategy were successful. The caching system dramatically improved the efficiency of the MVT implementation by significantly reducing the database requests. Without the caching system, the MVT implementation struggled due to the high frequency of database requests. On the other hand, when a high number of tiles are cached, the performance is greatly improved. In the 'Future Work' section, we will discuss a proposal for a queuing system for the backend tile server to further optimize the solution.

Performance Evaluation A comprehensive performance evaluation was carried out as planned. The evaluation included testing the performance of both the MVT and GeoJSON implementations under various conditions, like different numbers of moving objects and different datasets. The results of these tests have provided valuable insights into the strengths and weaknesses of both implementations and have guided the discussion of improvements and future work.

User Experience Assessment Although no formal tests with actual users were performed, it is believed that the timeline UI component is quite self-explanatory and user-friendly. The other parameters (number of objects and reset tiles cached) might need some initial exploration by the users to fully understand their implications, but with some trial and error, the functionality can be learned. Future work could involve more extensive user testing and feedback collection to further improve the user experience and refine these components.

7.4 Future Work

7.4.1 Use of Well-Known Binary Instead of GeoJSON

The current implementations utilize GeoJSON to transmit spatial-temporal data to the frontend, a format readily interpreted by various mapping software. However, as an improvement, Well-Known Binary (WKB) could be used, which might decrease the data transmission size significantly due to its binary nature. It will also require less computational resources on the server side to prepare the data. However, this approach would come with its own challenges, most notably the requirement of a system that can deserialize WKB on the frontend while preserving the essential timestamp information for each coordinate. Currently, there is no

known extension available for leaflet to handle WKB for spatiotemporal objects, which leaves room for development.

As mentioned, transitioning from GeoJSON to WKB has potential benefits but also introduces new challenges. Two alternative paths for future work have been identified that address these challenges: decoding WKB on the frontend and backend respectively. Each approach has its own implications for data processing and system performance, and their development could present exciting opportunities for further research.

These propositions are detailed in the following subsections, providing a roadmap for potential improvements and advancements in this field of study.

Decoding Well-Known Binary on the Frontend

A promising direction for future work could be to decode the WKB data directly on the frontend. To accommodate this, a suitable class structure can be designed to handle the decoded spatiotemporal data. Two such classes could be:

- **TPoint:** This class could have two attributes: `coordinates`, an array of floats, and `timestamp`, an integer. Several operations could be defined on this class. For the context of this project, interpolation of two TPoints would be particularly interesting and useful.
- **Trip:** This class would contain multiple TPoint objects. A function could be designed within this class to extract the specific position of the trip given a timestamp.

In this setup, most of the computation is transferred to the frontend. It is important to note that, with react, it is possible to force some functions to be executed on the server side to reduce the computational load on the client.

Decoding Well-Known Binary on the Backend

Alternatively, the WKB could be decoded directly on the backend. This could be achieved using the PyMeos project,[1] which is linked to MobilityDB. By decoding the WKB using PyMeos on the backend server, the computational load on the PostgreSQL server can be significantly reduced. The backend server can then perform the heavy lifting in terms of computations required by the frontend. Implementing a memoization strategy on the backend server could further enhance the overall performance of the architecture.

It would be an interesting future work to evaluate the performance trade-offs between these two methods and possibly find a hybrid approach that utilizes the best of both strategies.

7.4.2 Shared Time Bar for Multiple Maps

The current implementation treats each map as an individual unit with its independent time bar. This structure presents difficulties when attempting to conduct

[1] `https://github.com/MobilityDB/PyMEOS`

comparative analysis across multiple maps. Consider the task of comparing the movements of various types of public transport, such as buses and trams, over an identical time period. The implementation of a unified time bar for multiple maps could significantly enhance such comparative investigations.

Current Implementation To comprehend the proposed solution, we must first understand the current structure. Both of our implementations (MVT and GeoJSON) possess an independent `timestamp` state. These states are updated in a `useEffect` as shown below:

```
useEffect(() => {
  const UPDATE_PER_SECOND = 100;
  if (startSimulation) {
    const interval = setInterval(() => {
      updateTimez();
    }, 1000 / UPDATE_PER_SECOND);
    if (intervalTime) {
      clearInterval(intervalTime);
    }
    setIntervalTime(interval);
  } else {
    clearInterval(intervalTime)
  }
}, [startSimulation]);
```

Proposed Implementation The objective is to migrate the aforementioned code from each component (`MVTLayer` and `GeojsonLayer`) to the parent component (`App`). Consequently, the `timestamp` state should be passed as `props` to each implementation. The revised `App` component would look as follows:

```
function App() {
  const db_name = "persona";
  let [timestamp, setTimestamp] = useState(0);
  return (
    <div className={'App notransition'}>
      <MVTLayer db_name={db_name} timestamp={timestamp}/>
      <GeojsonLayer db_name={db_name} timestamp={timestamp}/>
    </div>
  );
}
```

Following this, each component should incorporate the new props as shown below:

```
export default function MVTLayer({db_name, timestamp})
```

It is worth noting that the `useEffect` currently relies on `startSimulation` state, which is not present in the `App` component. This state also needs to be moved to the `App` component.

Moreover, all visual rendering related to the slider and the start/stop button needs to be relocated to the `App` component. The revised render code appears as:

```
<div className={'control-panel'}>
  <div style={{display: "flex", flexDirection: "column"}}>
    <div className={`button `+ (startSimulation ? 'start' : 'stop
   ')} onClick={() => setStartSimulation(!startSimulation)}>
```

```
4         Start/Stop
5       </div>
6       <div className={'button'} onClick={() => {
7         new Date(minMaxTimestamp[0] * 1000).toISOString().substr(0,
      19).replace('T', ' ');
8         setTimestamp(minMaxTimestamp[0]);
9       }}>
10        reset time
11      </div>
12    </div>
13    <div style={{display: "flex", flexDirection: "column", minWidth:
        "70%", justifyContent: "center", justifyItems: "center",
      alignItems: "center"}}>
14      <div>{timez}</div>
15      <Slider min={minMaxTimestamp[0]} max={minMaxTimestamp[1]}
      value={timestamp} onChange={(e) => {
16        setTimestamp(parseInt(e.target.value));
17        setTimez(new Date(e.target.value * 1000).toISOString().slice
      (0, 19).replace("T", " "));
18      }}/>
19    </div>
20  </div>
```

Finally, to enhance modularity, this render code could be moved to a separate component named `TimeController`. While the shared states should continue to exist in the `App` component, the logic can be moved to the `TimeController` component. This division of responsibilities will offer superior control and aid in reducing bugs.

7.4.3 Expanding the Techniques and Methods

The rapidly advancing field of spatiotemporal data visualization offers a plethora of exciting directions for future research and development. While some opportunities for improvement and innovation have been discussed in detail in previous sections, there are several other potential avenues that merit consideration. These possibilities, each of which could fill a unique niche in the spatiotemporal visualization landscape, are briefly introduced in the following paragraphs. While these ideas represent only a small sample of the potential future developments in this area, they serve to underscore the vast array of possibilities that await exploration.

Integration of Leaflet.MarkerCluster Plugin: The Leaflet.MarkerCluster plugin presents a promising enhancement to our current project [**MarkerCluster**]. This plugin aggregates points into clusters to simplify visual analysis, especially in dense areas. The potential application is significant. For instance, it can provide insights on spatial-temporal density patterns throughout different times of the day.

One notable aspect of this plugin, as described in the [**leafletArticle**], is its impressive performance: 'It is very fast, so for example clustering 50,000 points isn't a problem. Also, all the heavy calculation happens on initial page load, and after this the map works smoothly'. Despite this promising claim, there is an initial

loading time that requires attention. As part of future work, it would be worthwhile to investigate how this loading time scales with the volume of data, and whether it is manageable in a real-world scenario.

Enabling or disabling this plugin's functionality on the fly could allow users to toggle between different levels of detail according to their analysis needs. This added flexibility could significantly enhance the user experience and utility of the application.

Enhancing Temporal Data Transmission Currently, the system primarily focuses on the transmission of geographical locations tied to specific timestamps, limiting the data's depth. Future improvements could incorporate the transmission of a wider array of temporal data, such as changes in the transportation mode for a given trajectory (e.g., transitions from walking to taking a bus, then switching to a car). This enhancement would not only allow for richer, more detailed visualizations but also pave the way for complex analyses, like congestion prediction or route optimization, further expanding the use cases of the system.

Improvement of Backend Tile Server Queuing System The current backend tile server uses a First-In, First-Out (FIFO) queuing system for handling requests, which while logical, may not be the most efficient approach for this specific use-case. For instance, when a user performs multiple operations such as panning and zooming, they would prefer the most recent requests to be fulfilled first. A Last-In, First-Out (LIFO) system or a priority-based queue could offer a better user experience by prioritizing recent requests. This could make a good project for future work, perhaps even in collaboration with the pg_tileserv development team.

Incorporation of Real-Time Data The current system is primarily designed to handle historical AIS data. As an area of future work, it could be intriguing to incorporate real-time data, enabling immediate tactical decisions or predictions. A number of APIs offer real-time data, such as the API from "La STIB",[2] "NextBus",[3] and "Transport for London Unified API."[4] These APIs provide real-time public transit data, and could be used as a starting point for implementing real-time data streaming. However, this would introduce new challenges, including the need to manage data streaming, update the frontend in real-time, and handle potentially increased server loads.

Extending to Other Domains While this study primarily centers around terrestrial and maritime data, the methodologies and techniques implemented hold potential for extension to a broader range of domains, including but not limited to aerial transportation, urban planning, or environmental studies. The extension to these new domains would necessitate modifications to accommodate the respective types of data and to develop suitable visualizations that accurately represent the inherent characteristics of these datasets. However, the fundamental principles

[2] `https://stibmivb.opendatasoft.com/pages/home/`

[3] `https://www.nextbus.com/xmlFeedDocs/NextBusXMLFeed.pdf`

[4] `https://api.tfl.gov.uk/`

of managing large-scale spatiotemporal data, as laid out in this study, would remain relevant across different applications. Therefore, future work could consider expanding the application's reach into these different domains, creating a more versatile tool that can serve a wider range of needs. This future exploration, although challenging, promises an exciting opportunity to further test and refine the methods and learnings put forth in this thesis.

Exploring Diverse Visualization Techniques The current study primarily focused on visualizing spatiotemporal data through animations of moving objects over time. However, MobilityDB, the underpinning database system for managing and analyzing moving object data, can support a much broader range of visualizations. Future work could involve exploring and implementing these diverse visualization techniques to better interpret and communicate complex spatiotemporal patterns. This could include static visualizations, such as heat maps or dot density maps, or more dynamic representations, such as interactive 3D models or augmented reality overlays. Each of these techniques brings its own unique perspective to the data and can help to reveal different aspects of the phenomena under study.

Chapter 8

Conclusion

This thesis embarked on an ambitious and in-depth exploratory journey, the destination of which was the development of a robust, leaflet-based web application. The main objectif of this thesis was the visualization of large-scale spatiotemporal data, a growing field of study and practice that intersects a multitude of scientific and social domains. As datasets grow larger and become more complex, with the advancement of data acquisition technologies and methods, the demand for efficient and effective tools for managing and visualizing such data becomes paramount.

In response to this need, the thesis delved into an in-depth exploration of two methodologies, namely MVT and GeoJSON. The choice of these methodologies was driven by a range of factors, but the central objective was to find an optimal balance between computational efficiency, ease of implementation, and robust performance. Both methodologies have their own sets of strengths and challenges, and through this thesis, we sought to put them to the test in a rigorous and practical manner.

At the heart of the project was Leaflet, a widely recognized open-source JavaScript library known for its mobile-friendly interactive maps. As a primary requirement and constraint for this project, Leaflet represented a particular technological landscape that shaped the direction of our research. The task at hand was to design an application that could effectively utilize this tool, thereby diving deep into the examination of its capabilities and limitations in handling large-scale spatiotemporal data.

In the process of verifying our methodologies, the application was subjected to rigorous testing using two distinct datasets: an extensive Automatic Identification System (AIS) dataset and the synthetic Persona dataset. The AIS dataset, comprising real data of various ships, primarily focused on maritime traffic in the North of Europe. This dataset presented a rich and complex network of maritime mobility patterns, providing an authentic representation of the complexities inherent in large-scale spatiotemporal data.

On the other hand, the Persona dataset, a synthetic representation of the mobility patterns of Belgians over a single day, added another layer of diversity to our test environment. Although the data were not sourced from real-world movements, they mirrored realistic terrestrial mobility patterns, thereby providing a complementary perspective to the maritime focus of the AIS dataset.

Together, these datasets presented a comprehensive and multi-faceted image of the challenges and opportunities within large-scale spatiotemporal data visualization. By reflecting both maritime and terrestrial mobility patterns, they allowed

for a well-rounded and versatile examination of the effectiveness and robustness of our methodologies in practically relevant contexts.

The MVT implementation emerged as a resilient and effective approach to managing and visualizing large volumes of data. One of the key discoveries in this process was the instrumental role of the Least Frequently Used (LFU) caching system. The caching system significantly enhanced the performance of the MVT approach, providing rapid data retrieval and efficient handling of vast datasets. This underscores the potential of MVT, when optimally paired with an efficient caching mechanism, in the management and visualization of large-scale spatiotemporal data within a leaflet-based application.

Contrasting the MVT approach was the GeoJSON implementation. Despite appearing less complex, this approach surfaced with its unique set of advantages, most notably simplicity and adaptability. The flexibility of GeoJSON to operate with various backends, eliminating the need for a tile server, is a quality that can resonate with developers and users alike. As the results suggested, this methodology, in spite of its simplicity, proved effective in delivering a smooth, user-friendly approach to large-scale spatiotemporal data visualization.

However, in the process of uncovering these findings, we also faced some limitations. Among them, the challenge of dealing with JavaScript's capacity to handle large volumes of data stood out, as did the constraints imposed by Leaflet in handling moving objects. But rather than hindrances, we saw these as opportunities for future research to delve into alternatives and improvements. For instance, the use of Well-Known Binary (WKB) instead of GeoJSON could be one such alternative, potentially reducing data transmission size and hence, the computational burden on the server. Similarly, designing a more efficient queuing system for the backend tile server could be another strategy to enhance the performance of our implementations.

Looking towards the horizon of future research, there are several possibilities that could further enrich the visualization capabilities of the system. Some of these possibilities include the integration of more detailed trajectory information, like the means of transportation used for a specific trajectory, or the introduction of a single time bar for multiple maps to enable comparative visualization of different parts of the same dataset.

In conclusion, this thesis provides a compelling case for the feasibility of creating an efficient, user-friendly, and interactive leaflet-based web application that can effectively handle large volumes of spatiotemporal data. The developed platform performed impressively not only in terms of computational efficiency but also in providing a rich and intuitive user interaction with the data. It is our hope that the findings and insights shared here will serve as a guiding light for future research, further expanding the boundaries of possibilities in the realm of spatiotemporal data visualization. The ultimate goal is to contribute to a more effective and meaningful use of these rich and valuable datasets, which can open up new vistas of understanding and knowledge.

Bibliography

[1] E. Zimányi, M. Sakr and A. Lesuisse. 'MobilityDB: A Mobility Database Based on PostgreSQL and PostGIS'. In: *ACM Transactions on Database Systems* 45, 2020.

[2] M. Friendly. 'Visions and Re-Visions of Charles Joseph Minard'. In: *Journal of Educational and Behavioral Statistics* 27, March 2002, pp. 31–51.

[3] E. R. Tufte. *The visual display of quantitative information*. Graphics Press, 1992.

[4] A. Tatem, D. Rogers and S. I. Hay. 'Global Transport Networks and Infectious Disease Spread'. In: *Advances in Parasitology* 62, 2006, pp. 293–343.

[5] G. Xavier and S. Dodge. 'An exploratory visualization tool for mapping the relationships between animal movement and the environment'. In: ACM, November 2014, pp. 36–42.

[6] G. Andrienko, N. Andrienko and S. Wrobel. 'Visual analytics tools for analysis of movement data'. In: *ACM SIGKDD Explorations Newsletter* 9, December 2007, pp. 38–46.

[7] X. Xiong, M. Mokbel and W. Aref. 'Spatio-temporal Database'. In: *Encyclopedia of GIS*. Ed. by S. Shekhar and H. Xiong. Springer US, 2008, pp. 1114–1115.

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

[9] *About PostGIS*. URL: `https://postgis.net/` (visited on 09/06/2023).

[10] *An open source geospatial trajectory data management analysis platform.* URL: `https://mobilitydb.com/` (visited on 09/06/2023).

[11] J. He et al. 'Diverse Visualization Techniques and Methods of Moving-Object-Trajectory Data: A Review'. In: *ISPRS International Journal of Geo-Information* 8, January 2019, p. 63.

[12] M. Harrower and C. A. Brewer. 'ColorBrewer.org: An Online Tool for Selecting Colour Schemes for Maps'. In: *The Cartographic Journal* 40, June 2003, pp. 27–37.

[13] S. Dodge, M. Toka and C. J. Bae. 'DynamoVis 1.0: an exploratory data visualization software for mapping movement in relation to internal and external factors'. In: *Movement Ecology* 9, 2021, p. 55.

[14] C. Tominski et al. 'Stacking-Based Visualization of Trajectory Attribute Data'. In: *IEEE Transactions on Visualization and Computer Graphics* 18, December 2012, pp. 2565–2574.

[15] R. Scheepens et al. 'Composite Density Maps for Multivariate Trajectories'. In: *IEEE Transactions on Visualization and Computer Graphics* 17, December 2011, pp. 2518–2527.

[16] S. Liu et al. 'VAIT: A Visual Analytics System for Metropolitan Transportation'. In: *IEEE Transactions on Intelligent Transportation Systems* 14, December 2013, pp. 1586–1596.

[17] G. Di Lorenzo et al. 'AllAboard: Visual Exploration of Cellphone Mobility Data to Optimise Public Transport'. In: *IEEE Transactions on Visualization and Computer Graphics* 22, February 2016, pp. 1036–1050.

[18] B. Tversky, J. Bauer Morrison and M. Betrancourt. 'Animation: can it facilitate?' In: *International Journal of Human-Computer Studies* 57, 2002, pp. 247–262.

[19] T. von Landesberger et al. 'MobilityGraphs: Visual Analysis of Mass Mobility Dynamics via Spatio-Temporal Graphs and Clustering'. In: *IEEE Transactions on Visualization and Computer Graphics* 22, January 2016, pp. 11–20.

[20] T. N. Dang, L. Wilkinson and A. Anand. 'Stacking Graphic Elements to Avoid Over-Plotting'. In: *IEEE Transactions on Visualization and Computer Graphics* 16, November 2010, pp. 1044–1052.

[21] A. Kölzsch et al. 'MoveApps: a serverless no-code analysis platform for animal tracking data'. In: *Movement Ecology* 10, 2022, p. 30.

[22] G. A. M. Gomes, E. M. dos Santos and C. A. Vidal. 'Interactive Visualization of Traffic Dynamics Based on Trajectory Data'. In: *30th SIBGRAPI Conference on Graphics, Patterns and Images*. IEEE Computer Society, 2017, pp. 111–118.

[23] F. Ferreira da Silva. 'Visual Analytics for Moving Objects Databases'. MA thesis. Technische Universität Berlin, 2021.

[24] L. Van Calck. 'Visualizing MobilityDB Data using QGIS'. MA thesis. Univsersité Libre de Bruxelles, 2021.

[25] M. Schoemans, M. A. Sakr and E. Zimányi. 'MOVE: Interactive Visual Exploration of Moving Objects'. In: *Proceedings of the Workshops of the EDBT/ICDT 2022 Joint Conference*. Vol. 3135. CEUR Workshop Proceedings. CEUR-WS.org, 2022.

[26] S. El Bakkali Tamara. 'Visualization of Mobility Data on Openlayers'. MA thesis. Université Libre de Bruxelles, 2023.

[27] *An open-source Javascript library for mobile-friendly interactive maps*. URL: `https://leafletjs.com/` (visited on 09/06/2023).

[28] *Vector Tiles Standards*. URL: `https://docs.mapbox.com/data/tilesets/guides/vector-tiles-standards/` (visited on 09/06/2023).

[29] *Our vision*. URL: `https://www.mapbox.com/about` (visited on 09/06/2023).

[30] *Vector Tiles*. URL: https://docs.mapbox.com/help/glossary/vector-tiles/ (visited on 09/06/2023).

[31] *A New Commitment to Developer Relations at Mapbox*. URL: https://www.mapbox.com/developers/ (visited on 09/06/2023).

[32] *About pg_tileserv*. URL: https://access.crunchydata.com/documentation/pg_tileserv/latest/introduction/ (visited on 09/06/2023).

[33] *Google Maps Platform*. URL: https://developers.google.com/maps (visited on 04/07/2023).

[34] *A high-performance, feature-packed library for all your mapping needs*. URL: https://openlayers.org/ (visited on 04/07/2023).

[35] S. El Bakkali Tamara. *MobilityDB-OpenLayers*. URL: https://github.com/SoufianBk/MobilityDB-OpenLayers (visited on 04/07/2023).

[36] *Mapbox GL JS*. URL: https://docs.mapbox.com/mapbox-gl-js/ (visited on 04/07/2023).

[37] *3D geospatial visualization for the web*. URL: https://cesium.com/cesiumjs (visited on 04/07/2023).

[38] *ArcGIS Maps SDK for JavaScript*. URL: https://developers.arcgis.com/javascript/ (visited on 04/07/2023).

[39] *Deck.gl*. URL: https://deck.gl/ (visited on 04/07/2023).

[40] O. G. Consortium. *OpenGIS Simple Feature Access*. Open Geospatial Consortium. 2006.

[41] *FastAPI*. URL: https://fastapi.tiangolo.com/ (visited on 04/08/2023).

[42] *The library for web and native user interfaces*. URL: https://reactjs.org/ (visited on 17/06/2023).

[43] *Findings from the Jamstack Community Survey 2022*. URL: https://jamstack.org/survey/2022/ (visited on 17/06/2023).

Appendix A

Code snippets

```
1  CREATE FUNCTION public.tripsfct(
2              z integer, x integer, y integer, maxpoints integer
       default 10)
3  RETURNS bytea
4  AS $$
5      WITH bounds AS (
6          SELECT ST_TileEnvelope(z, x, y)::stbox AS geom
7      ),
8      val AS (
9          SELECT id,  asMVTGeom(fullday_trajectory, bounds.geom) as
       geom_times
10         FROM persona_small_4000, bounds
11         ORDER BY id
12         LIMIT maxpoints
13
14     ),
15     mvtgeom AS(
16         SELECT id as tripid, (geom_times).geom, (geom_times).times
17         FROM val
18     ),
19     SELECT st_asmvt(mvtgeom, 'reduced') FROM mvtgeom
20 $$
21 LANGUAGE 'sql'
22 STABLE
23 PARALLEL SAFE;
```

Listing A.1: MVT function

```
1      CREATE OR REPLACE FUNCTION average_exec_time_json(
2      runs INT
3  ) RETURNS VOID AS $$
4  DECLARE
5      startTime TIMESTAMP;
6      endTime TIMESTAMP;
7      i INT;
8      p4 INT;
9      totalInterval INTERVAL;
10     avgTime FLOAT;
11     p4_values INT[] := ARRAY[100,250,500,1000,2000, 5000, 10000];
12 BEGIN
13     FOREACH p4 IN ARRAY p4_values LOOP
14         totalInterval := '0 second';
```

```
15          FOR i IN 1..runs LOOP
16              startTime := clock_timestamp();
17              -- Call the function
18              PERFORM  asmfjson(transform(trip, 4326))::json from
    ships limit p4;
19
20              endTime := clock_timestamp();
21
22              -- Exclude the first 5 runs
23              IF i > 5 THEN
24                  totalInterval := totalInterval + (endTime -
    startTime);
25              END IF;
26          END LOOP;
27
28          -- Calculate the average execution time in milliseconds
29          avgTime := EXTRACT(EPOCH FROM totalInterval) * 1000 / (
    runs - 5);
30
31          -- Print the p4 value and its average execution time
32          RAISE NOTICE 'Average execution time for p4 = %: % ms', p4
    , avgTime;
33      END LOOP;
34  END;
35  $$ LANGUAGE plpgsql;
```

Listing A.2: GeoJSON loading benchmark

```
1   CREATE OR REPLACE FUNCTION average_exec_time_mvt(
2       runs INT,
3       p1 INT,
4       p2 INT,
5       p3 INT
6   ) RETURNS VOID AS $$
7   DECLARE
8       startTime TIMESTAMP;
9       endTime TIMESTAMP;
10      i INT;
11      p4 INT;
12      totalInterval INTERVAL;
13      avgTime FLOAT;
14      p4_values INT[] := ARRAY[100,250,500,1000,2000, 5000, 10000];
15  BEGIN
16      FOREACH p4 IN ARRAY p4_values LOOP
17          totalInterval := '0 second';
18          FOR i IN 1..runs LOOP
19              startTime := clock_timestamp();
20
21              -- Call the function
22              PERFORM tripsfct(p1,p2,p3,p4);
23
24              endTime := clock_timestamp();
25
26              -- Exclude the first 5 runs
27              IF i > 5 THEN
28                  totalInterval := totalInterval + (endTime -
    startTime);
29              END IF;
```

```
30          END LOOP;
31
32          -- Calculate the average execution time in milliseconds
33          avgTime := EXTRACT(EPOCH FROM totalInterval) * 1000 / (
    runs - 5);
34
35          -- Print the p4 value and its average execution time
36          RAISE NOTICE 'Average execution time for p4 = %: % ms', p4
    , avgTime;
37      END LOOP;
38  END;
39  $$ LANGUAGE plpgsql;
```

Listing A.3: MVT tile loading benchmark