Master's Thesis

Visualization of Mobility Data on Openlayers

Soufian EL BAKKALI TAMARA

Under the supervision of
M. Esteban ZIMANYI

Academic year
2022 - 2023

I hereby declare on my honor that this thesis is the fruit of my personal and original work. I also declare that I have respected the rules of academic integrity by correctly citing all the sources used in this work.

Brussels, August 11, 2023

Soufian El Bakkali

Abstract

The gathering and analysis of mobility data has taken on considerable importance in many areas, such as urban planning, transport and logistics. However, the effective visualization of this massive data still represents a major challenge. This thesis proposes an innovative approach to visualizing mobility data using the OpenLayers library. It begins with a detailed description of OpenLayers, which offers powerful features for interactive web based mapping. It then presents an overview of existing techniques for visualizing mobility data, highlighting current limitations and shortcomings. The methodology adopted for the comparative visualization of mobility data is based on the integration of MobilityDB with OpenLayers, taking into account the GeoJSON and vector tiles formats. Mobility data is efficiently stored and managed in MobilityDB while OpenLayers is used for data visualization. A case study is carried out to assess the efficiency of the proposed approach. Mobility data from various sources, such as Danish AIS and Brooklyn public transportation, is preprocessed and integrated into OpenLayers. In this study, we compare the use of pg_tileserv, a PostgreSQL tiling service, and Express, a web development framework, for the efficient distribution of mobility data in the form of vector tiles. We evaluate the performance, flexibility and ease of use of pg_tileserv and Express in the context of mobility data visualization. Both approaches are implemented and tested in terms of loading time, scalability and ease of deployment. The results obtained provide valuable information for choosing the visualization method best suited to the specific needs of mobility data. Future prospects are also discussed, opening the way to further developments in the field of comparative visualization of mobility data.

Acknowledgments

I would like to express my deepest gratitude and most sincere thanks to the following people and institutions who contributed in such a way to the completion of this thesis:

My promoter, Esteban Zimanyi, for his guidance throughout this research journey. His valuable ideas, constructive comments and encouragement have played a decisive role in the direction and quality of this work.

My family and friends, for their unconditional love, understanding and constant support. Their confidence in my abilities and their willingness to listen to me during difficult times have been my sources of strength.

The research participants who generously shared their time, knowledge and experience. Their invaluable contributions enriched this study and made it a reality.

Finally, I would like to express my gratitude to all those who, directly or indirectly, contributed to the completion of this thesis. Your support, encouragement and inspiration have been essential every step of the way.

Contents

List of Figures

List of Listings

List of Tables

List of Abbreviations

AIS Automatic Identification System

API Application Programming Interface

COG Course over Ground

CPU Central Processing Unit

CRS Coordinate Reference System

DMA Danish Maritime Authority

DRAM Dynamic Random Access Memory

EPSG European Petroleum Survey Group

ETA Estimated Time of Arrival

EWKB Extended Well-Known Binary

FPS Frames Per Second

GIS Geographic Information Systems

GML Geography Markup Language

GPS Global Positioning System

GPU Graphics Processing Unit

GTFS General Transit Feed Specification

IMO International Maritime Organization

JS JavaScript

JSON JavaScript Object Notation

JSX JavaScript Syntax Extension

MFJSON Moving Features JavaScript Object Notation

MMSI Maritime Mobile Service Identity

MTA Metropolitan Transportation Authority

MVT Mapbox Vector Tile

NPM Node Package Manager

OGC Open Geospatial Consortium

OS Operating System

OSM OpenStreetMap

RAM Random Access Memory

ROT Rate of Turn

SOG Speed over Ground

SPA Single Page Application

SQL Structured Query Language

SRID Spatial reference system

WKB Well-Known Binary

Chapter 1

Introduction

With the constant growth of technologies, particularly mobile and tracking technologies, today's society is generating and using spatiotemporal data more and more. This data, which combines spatial information with temporal information, offers the potential for in-depth analysis and understanding of the world around us.

The development of navigation technologies, such as the Global Positioning System (GPS), has played a major role in the growth of this data. Mobile devices, connected vehicles, intelligent objects and more have become massive sources of spatiotemporal data, providing information on trips, etc.

This data offers a lot of different use cases for businesses, particulars and even governments. For example, air traffic control entities need information about the current position and direction of a plane, car rental companies have to keep track of their rented cars or even someone who orders food has access to the live location of the delivery person. Governments can also use this data to monitor and manage public transport, plan urban development and make safety decisions.

Faced with this abundance of data, it brings us to the question of **how to efficiently visualize this data**. Spatial data visualization libraries and frameworks have developed quickly to meet this growing demand. Popular tools include deck.gl, Mapbox GL JS, Leaflet and OpenLayers, which offer powerful features for displaying and interacting with geospatial data on the web.

In addition to visualization libraries, it is also essential to consider the methods used to represent spatial data. There are three commonly used formats: vector tiles, GeoJSON and Well-Known Binary (WKB). Vector tiles are a popular format for spatial data representation, dividing the data into ready-to-use tiles. This approach improves rendering performance and enables fluid interaction with data at different scales. On the other hand, GeoJSON offers a simple and portable representation of geospatial data based on JSON. Lastly, the Well-Known Binary format provides a binary representation of geometric data, designed for more compact storage and faster data processing.

To gain a better understanding of these different visualization and storage methods, an in-depth benchmark was carried out using OpenStreetMap (OSM) data obtained through Geofabrik's download server. The aim was to evaluate the performance and loading times of the different libraries and methods, focusing on specific regions of the OSM dataset, such as cities, towns, villages and hamlets in

different European countries. In addition, a separate benchmark was carried out to evaluate the performance of mobility data, focusing on the efficiency of rendering moving objects.

This thesis focuses on the use of OpenLayers to visualize AIS mobility data of ships navigating in waters close to Denmark, in particular in the Skagerrak, Kattegat and Baltic Sea. The aim is to demonstrate how OpenLayers can efficiently manage and present spatiotemporal information providing valuable insights into ship movements and maritime activites.

In addition, the study explores the integration of New York's General Transit Feed Specification (GTFS) data into the visualization, using pg_tileserv and Express architectures for vector tile management. This integration enables us to enrich our visualization with another data source.

Furthermore, this research explores a comparative analysis, examining in detail two distinct but essential visualization approaches. The first methodology involves the use of vector tiles, offering a sophisticated means of representing complex spatiotemporal data. On the other hand, the second approach employs the versatile GeoJSON format, adding a further level of depth to the visualization process.

Moreover, this study will also include a thorough exploration of two renowned frameworks: the dynamic deck.gl and the multipurpose OpenLayers. Through an evaluation of these frameworks, this research aims to reveal the interaction between software libraries and the visualization of complex mobility data. By highlighting the strengths and weaknesses of each tool, particularly when confronted with complex spatiotemporal datasets, this analysis seeks to provide valuable information for both researchers and developers in the field.

Throughout these chapters, we will explore in detail the tools and technologies used to visualize and analyze mobility data, focusing on performance and different storage approaches. The results and conclusions of this work will provide a better understanding of the benefits and limitations of each method, offering valuable guidance for the implementation of mobility data visualization solutions in a variety of application contexts.

1.1 Objectives

First of all, the main objective is to find an effective solution for displaying mobility data on OpenLayers, by exploiting its geospatial visualization functionalities. Furthermore, this thesis also aims to evaluate the performance of this visualization. This implies other objectives to be achieved:

- One goal is to evaluate and compare the performance of different visualization libraries, in particular deck.gl, Mapbox GL JS, Leaflet and OpenLayers, in the context of displaying mobility data from various sources. By carrying out an in-depth benchmark, we will be able to determine the strengths and weaknesses of each library, in terms of loading time and data rendering.

- Then, a second objective is to look at different methods of serving spatiotemporal data, with a focus on vector tiles and the GeoJSON format. By assessing the pros and cons each method, we will be able to recommend the best approach according to the specific needs of each project.
- Another goal is to observe whether the React framework impacts on the performance of visualization.
- Finally, we aim to integrate New York GTFS data and Danish AIS data into our visualization, using the pg_tileserv and Express architectures for vector tile management, in order to demonstrate their usefulness in a real-world scenario.

By achieving these goals, we will contribute to improving the understanding and use of mobility data visualization tools, offering valuable insights for decision-makers, developers and researchers working in this ever-evolving field.

1.2 Organization

Chapter 2 provides an introduction to the study, highlighting the tools and technologies used in data visualization and offering an overview of the current state of the field.

Chapter 3 provides an in-depth benchmark based on static data, evaluating and comparing the different visualization solutions available on the market. Performance, usability and functionality criteria are used to measure the capabilities of the selected tools.

Chapter 4 describes different implementations for visualizing mobility data, exploring the approaches and techniques used to efficiently process and display data in motion, such as real-time public transport data or ship tracking data.

Chapter 5 presents the results of measuring the performance of the different implementations presented in the previous chapter, analyzing metrics such as loading times, FPS and the handling of large amounts of data.

Finally, Chapter 6 concludes the study by summarizing the main results, practical implications and future prospects, highlighting the recommended choices for data visualization on OpenLayers adapted to the specific needs of mobility data applications.

1.3 Contribution

With this thesis, we have made significant contributions by implementing an innovative solution to improve the visualization of mobility data and conducting several evaluations to verify its performance.

Firstly, our major contribution lies in the design and implementation of a mobility data visualization solution on OpenLayers. This solution leverages OpenLayers' capabilities to effectively and dynamically represent data from mobile devices. As part of our thesis we also identified key challenges related to mobility data visualization and proposed solutions to overcome them.

This work highlights the importance of mobility data visualization in various domains such as ship management or public transportation. The findings and contributions can serve as a solid foundation for future research and for the development of more advanced mobility data visualization solutions.

This thesis constitutes a significant contribution to the scientific and professional community by presenting an innovative solution for mobility data visualization on OpenLayers and providing evaluations to validate its performance. This research opens up exciting new perspectives for improving spatiotemporal data visualization, offering opportunities for pratical and innovative applications.

Chapter 2

State of the Art

2.1 MobilityDB

MobilityDB [35] is a powerful extension based on PostgreSQL and PostGIS that allows to manage geospatial trajectories and their temporal properties. Geospatial data plays an important role in various fields such as public transportation, environmental monitoring, location based services and more. However, traditional databases cannot effectively manage the temporal dimension associated with spatial data. MobilityDB overcomes this limitation by introducing new data types and functions for handling temporal information, opening up new possibilities for spatiotemporal analysis.

Figure 2.1: MobilityDB logo

One of the key features of MobilityDB is the implementation of new data types designed to manage spatial and temporal information. These data types include `tgeompoint` and `tgeogpoint`. A `tgeompoint`, for example, can represent the position of a moving object over time, making it ideal for tracking trajectories.

In addition to geospatial data, MobilityDB can include the temporal dimension to more traditional data types resulting in `tbool`, `tint`, `tfloat` and `ttext`. This feature makes it possible to store and analyze time varying attributes. For example, using `tint`, it is possible to store the speed of a vehicle at a given moment as an integer value associated with a `tgeompoint` corresponding to the vehicle's location.

MobilityDB offers a wide range of functions specifically designed to help with the handling of spatiotemporal information. These functions enable users to perform complex operations on geospatial trajectories with temporal attributes. Users can calculate the distance traveled by a moving object, determine the duration of a trajectory, identify the points of intersection of two trajectories and much more.

To efficiently analyze large spatiotemporal datasets, MobilityDB offers temporal

aggregates. These aggregates allow its users to calculate statistics over time intervals, providing information on trends or patterns present in the data. For example, users can calculate the average speed of a vehicle over a given period or determine the locations most frequently visited during a specific period.

2.2 Geographic Formats

Geospatial representation of data is essential in many areas, such as navigation, public transportation, data visualization and more. Hence, several formats such as **vector tiles**, **GeoJSON** or **Well-Known Binary** have been developed to simplify and contribute to geographic representation. Each format has its advantages and disadvantages.

2.2.1 GeoJSON

GeoJSON [4] is a popular data format used to represent geospatial features (like points and areas) and their non-spatial properties such as the name of a city. It is lightweight, human-readable and easy to parse because the format is based on JavaScript Object Notation (JSON). GeoJSON is widely used in applications ranging from web maps to Geographic Information Systems (GIS). It offers a standardized method for encoding geographic data, making it easy to share and exchange spatial information between different platforms and tools.

At the core of GeoJSON are the "features". A feature represents a specific spatial entity and is composed of 2 parts:

1. **Geometry**: This describes the geographical shape of a feature. It can be a point, a line, a polygon, a set of points or lines or even a set of different polygons. Geometry is generally defined by latitude and longitude.
2. **Properties**: These are the attributes associated with the feature. They provide additional information about the entity. Properties can represent information such as name, category, language, ...

Here is an example of a GeoJSON point indicating the location of London:

```
{
    "type": "Feature",
    "geometry": {
        "type": "Point",
        "coordinates":  [-0.1278, 51.5074]
    },
    "properties": {
        "name": "London"
    }
}
```

In this example, the feature represent the city of London with coordinates of longitude -0.1278 and latitude 51.5074. The associated attribute is "name" which provides the name of the city.

GeoJSON Types

GeoJSON supports different types of geometry to represent various geographic features. Here are the main data types supported: (only the types that will be used in this thesis are illustrated)

1. `Point`: represents a geographic point defined by its latitude and longitude coordinates. For example, a city. (An example of the representation of a point has already been given)
2. `MultiPoint`: represents a collection of points. This allows to group several points together in a single entity.

```
{
    "type": "Feature",
    "geometry": {
        "type": "MultiPoint",
        "coordinates": [
            [0.0, 0.0]
            [1.0, 0.0]
            [2.0, 1.0]
        ]
    }
    "properties": {
     "name": "Cities in Europe"
     "number": 3
    }
}
```

3. `LineString`: represents a sequence of connected line segments defined by an array of coordinates. For example, a road or a river.

```
{
    "type": "Feature",
    "geometry": {
        "type": "LineString",
        "coordinates": [
            [0.0, 0.0]
            [1.0, 0.0]
            [2.0, 1.0]
            [4.0, -3.0]
        ]
    }
    "properties": {
```

```
         "name": "Railroad"
            "length": 20
        }
    }
```

4. `MultiLineString`: represents a collection of lines. This allows to group multiple lines together in a single entity.

```
{
    "type": "Feature",
    "geometry": {
        "type": "MultiLineString",
        "coordinates": [
            [
                [0.0, 0.0]
                [1.0, 0.0]
                [2.0, 1.0]
            ],
            [
                [3.0, 0.0]
                [3.0, 2.0]
                [2.0, 4.0]
            ]
        ]
    }
    "properties": {
         "name": "Main roads"
         "number": 2
    }
}
```

5. `Polygon`: represents a closed shaped defined by a ring with the ring being characterized by an array of coordinates. Polygons can represent objects such as lakes, islands or even countries.

6. `MultiPolygon`: represents a collection of polygons. This allows numerous polygons to be grouped together in a single geographic entity.

The main strength of GeoJSON lies in its simplicity and compatibility with a wide range of programming languages. This is largely attributed to its fundamental structure built on the JSON (JavaScript Object Notation) format which is popular worldwide. The elegance of GeoJSON's simple design not only makes it easy to understand, but also allows it to be easily integrated with a wide range of programming languages, reinforcing its accessibility and applicability in various software development contexts around the world.

2.2.2 MFJSON

In addition to GeoJSON, which is designed to represent static geographic features, there is MFJSON which stands for Moving Features JavaScript Object Notation. MFJSON is a data format specified by the OGC [14, 17] for representing mobile geospatial features. In MFJSON, moving entities are represented using basic GeoJSON but with the addition of temporal information in the form of timestamps. Each MFJSON feature can associate a set of timestamps with coordinates.

Here is an example of a MFJSON feature representing moving object that was in the coordinates `[0.0, 0.0]` at midnight the first of June 2023 and was at the coordinates `[1.0, 0.0]` at 12h22 the same day:

```
{
 "type": "MovingPoint",
 "coordinates": [
     [0.0, 0.0]
     [1.0, 0.0]
 ],
 "datetimes": [
    "2023-06-01T00:00:00+01",
    "2023-06-01T12:22:34+01"
 ]
}
```

Although GeoJSON and MFJSON share similarities as they are both based on JSON, they are designed to fulfill different needs. GeoJSON is ideal for representing static spatial information and does not support temporal data. It is mainly used to display country borders, roads, cities and more. MFJSON on the other hand supports temporal information and is used to represent moving entities. It is useful for tracking moving objects such as vehicles, ships or even animals. This allows to analyze the evolution of moving objects over time. However, MFJSON is not as popular as GeoJSON.

2.2.3 Vector Tiles

In order to understand the impact of vector tiles on map representation, we first need to understand what raster tiles are. Raster tiles have long been the standard for online geographic representation. They consist of images (mainly in PNG or JPG) cut into small square tiled, each tile representing a part of the map at a specific scale.

As can be seen in Figure 2.2, at the lowest zoom level (level 0) we have only one tile but as soon as we zoom in a little more, the tile is divided into 4 different tiles which each correspond to an image.

Raster tiles have been widely used because of their simplicity and ease of imple-

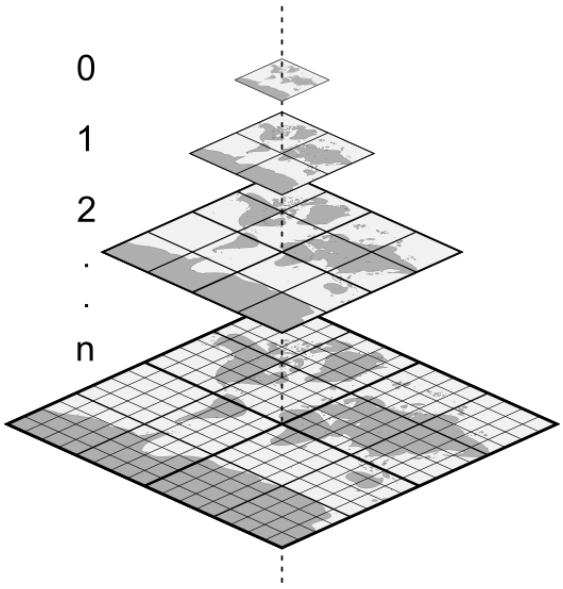


Figure 2.2: Tile pyramid [24]

mentation. They have provided static maps to millions of users around the world. However, raster tiles have some important shortcomings.

Firstly, raster tiles take up a lot of storage space. Since each tile is a still image, thousands of tiles must be generated to cover different zoom levels and map styles. This requires considerable storage space and can lead to longer loading times when a large amount of map data is used. Furthermore, raster tiles do not allow for customization or advanced interactivity. Map styles and attribute information are fixed in the images, which limits the flexibility of the end user to customize the appearance of the maps. In addition, interactions, such as dynamic zooming or searching for specific features, are limited due to the static nature of raster tiles. This is where vector tiles come in. Unlike raster tiles, vector tiles store geographic data as vectors rather than still images. Vectors represent geographic objects such as points, lines and polygons, and their associated attributes as seen in Figure 2.3. This approach allows for a more dynamic and customizable map representation.

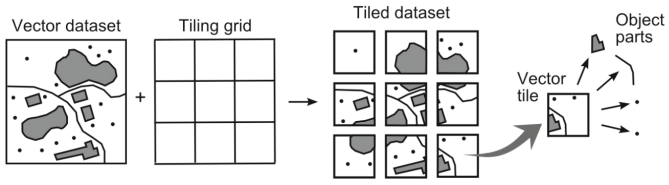


Figure 2.3: How vector tiles work [9]

Vector tiles are lighter in terms of file size than raster tiles, making it easier to transfer them quickly over the Internet and reducing the amount of storage space required. In addition, vector tiles offer advanced customization, as styles and

attributes can be adjusted in real time, providing a more flexible and interactive user experience. Netek et al. [21] have conducted an experiment to observe the performance between vector tiles and raster tiles. The results showed that in most cases vector tiles had better loading times.

Mapbox, a company specialized in web-mapping, was one of the first players to develop a specification for vector tiles: the **Mapbox Vector Tile** file format or **MVT** for short. They introduced MVTs, which are typically encoded in .pbf (Google protobufs) format, as a solution for storing and transferring vector tiles efficiently.

2.2.4 Well-Known Binary

Well-Known Binary (WKB) is a representation format used to store and exchange spatial geometry in geographic information systems and spatial databases. It is a widely used standard for representing geometric objects such as points, lines and polygons in binary format. The WKB format was introduced by the OGC [5] to provide a standardized method of storing and transmitting geometric data. It enables geometric features to be represented efficiently using less storage space than traditional text-based formats. The use of hexadecimal codes in the WKB format enables geospatial data to be represented efficiently using a compact binary representation. This makes it easier to store, transfer and process spatial geometry, while preserving coordinate accuracy.

The format is generally divided into several parts:

1. Endianness: the first byte of the WKB is used to determine the byte order. There are two possible types of endianness: big endian and little endian. In big endian, the most significant bytes are stored first, while in little endian, the least significant bytes are stored first. This endianness specification ensures interoperability between different platforms and architectures.

2. Geometry type: the WKB uses a numerical code to represent the type of geometry. For example, code 1 represents a point, code 2 represents a line, code 3 represents a polygon, and so on. This specification of the type of geometry enables the coordinates of the geometry to be interpreted correctly. Figure 2.4 shows the most common geometry types and their code

3. Coordinates: this part of the WKB contains the coordinates which define the spatial geometry. The structure of the coordinates can vary depending on the type of geometry. For example, for a point, the coordinates can simply be represented by a pair of `x` and `y` values. For a line or polygon, the coordinates can be represented by a sequence of `x` and `y` pairs. The coordinates can be stored as real numbers (floats) or integers (ints), depending on the precision requirements of the application.

Figure 2.5 comes from the document written by the OGC [5] which describes the WKB standard. It illustrates the WKB representation of a polygon containing a

Type	Code
Geometry	0
Point	1
LineString	2
Polygon	3
MultiPoint	4
MultiLineString	5
MultiPolygon	6
GeometryCollection	7
CircularString	8
CompoundCurve	9
CurvePolygon	10
MultiCurve	11
MultiSurface	12
Curve	13
Surface	14
PolyhedralSurface	15
TIN	16

Figure 2.4: WKB geometry codes

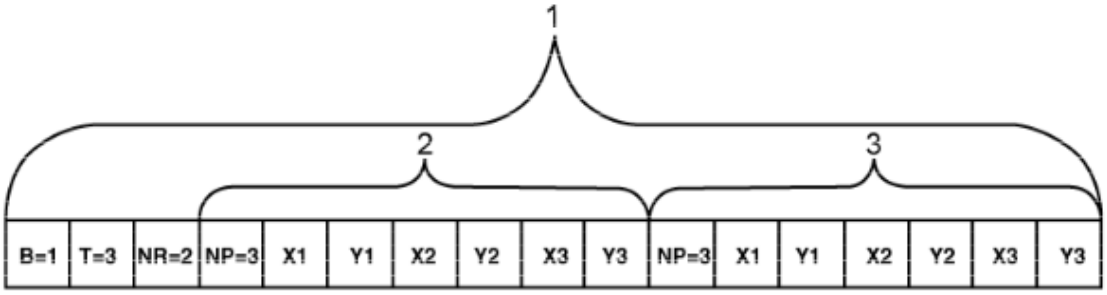


Figure 2.5: Well-Known Binary representation of a polygon

ring on the inside and another on the outside.

There are 3 parts in this illustration:

1. The polygon as a whole, which contains all the information described above. We can distinguish that this is in little endian because the first byte is equal to 1. The following bytes correspond respectively to the geometry type code, which is 3 (polygon), and to the number of rings that make up the polygon (in this case 2).

2. The inner ring. The first byte describes the number of points that make up the ring and the rest of the values correspond to the `x` and `y` pairs.
3. The outer ring. Like the inner ring, the first byte contains the number of points that make up the ring and the rest of the values correspond to the `x` and `y` pairs.

Extended Well-Known Binary

The Extended Well-Known Binary (EWKB) format is an extension of the WKB format. EWKB is a binary representation of geometric data that allows additional information to be stored compared with the traditional WKB format. As explained above, the WKB format is a standardized binary format used to represent geometric objects such as points, lines and polygons in a specified coordinate system. However, the WKB format has certain limitations in terms of representing data such as 3D geometries, geometries with additional measurements, and more.

This is where the EWKB comes in by extending the WKB format to allow the representation of more complex geometric data. The EWKB adds additional functionalities to the WKB format, allowing information such as `z` coordinates (adding the third dimension), additional measurements (e.g. temperature), specific geographic reference systems (SRID) and other customized data to be stored. The EWKB format uses an additional flag in the binary representation to indicate the extended functionality used. For example, there may be specific bits to indicate the presence of extra dimensions or additional measurements.

Well-Known Binary in MobilityDB

The Well-Known Binary format as well as the Extended Well-Known Binary are used in MobilityDB through the `asBinary` and `asEWKB` functions. The authors of MobilityDB have made a presentation [34] where they explain how WKB operates in a database that uses MobilityDB. Let's take the query in Listing 2.1 as an example.

```
1  SELECT asEWKB(tgeompoint 'Point(1 1)@2000-01-01');
```

Listing 2.1: Returns the trips in EWKB

It allows to extract in EWKB a point located at coordinates (1, 1) in a given spatial coordinate system with the timestamp associated with this point is 2000-01-01. Figure 2.6 shows the result of the query.

```
\x0181000000000000f03f000000000000f03f005c6c29ffffffff
Endian Flags 1 1 2000-01-01
```

Figure 2.6: Query result

- Byte `01` represents endianness.

- The following byte `81` indicates the value of the flags. As explained above, flags are used to specify additional information about the geometry.
- The following segments `000000000000f03f` are identical and are used to represent `x` and `y` coordinates respectively.
- Finally, the remaining bytes `005c6c29ffffffff` represent the time information associated with the point.

2.3 Data Source

2.3.1 Automatic Identification System

The Automatic Identification System (AIS) is deployed on ships to provide short-range coastal tracking. This technology provides information such as the position, speed and direction of ships at sea, playing a fundamental role in areas such as maritime safety, ship traffic management and the monitoring of activities at sea. By facilitating the real-time exchange of data between ships and coastal stations, AIS helps prevent collisions, improve vessel traffic management and increase maritime surveillance for safer and more efficient operations.

To illustrate the use of AIS data, we will take the example of Denmark, a country known for its thriving maritime industry and its strategic position as a shipping hub in Northern Europe. Denmark has developed an AIS network that collects and processes data from ships navigating in its territorial waters and neighbouring international waters. This data is collected by AIS shore stations, satellites or even ships equipped with AIS transceivers. The information collected is then transmitted to maritime control centres, port authorities and other entities responsible for managing maritime traffic. In Denmark, the Danish Maritime Authority (DMA) is responsible for processing AIS data.

Danish AIS data is useful for a variety of applications. First and foremost, it is used for maritime safety. By monitoring ships' movements, the authorities can detect potentially dangerous situations, such as risks of collision or irresponsible behaviour at sea. This means that preventive measures can be taken quickly to avoid accidents.

AIS data is also essential for maritime traffic control. It helps optimize the use of shipping routes by providing real-time traffic information. This enables port authorities to plan the docking of ships, reducing waiting times and improving the efficiency of port operations. For example, on 1 July 2020 the International Maritime Organization (IMO) in collaboration with the Danish Maritime Authority and the Swedish Maritime Administration established new shipping route regulations in the Skagerrak and Kattegat along the Danish and Swedish coasts. The goal is to create more predictable traffic patterns to improve the safety of navigation in the region and reduce the high number of ships in the existing route by establishing new recommended routes as can be seen in Figure 2.7.

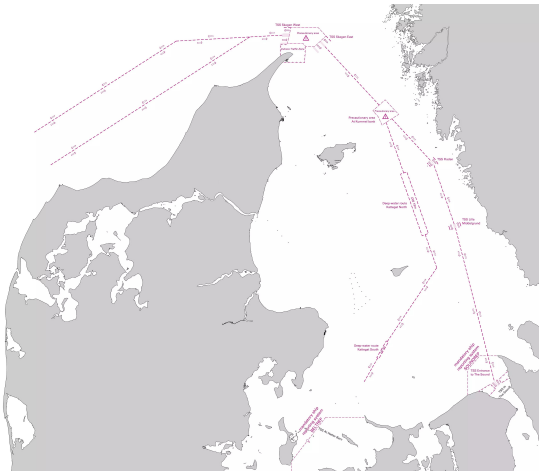


Figure 2.7: New recommended routes

Danish AIS data is also used in maritime surveillance. It helps prevent pirate activity, illicit trafficking and other illegal activities at sea. By monitoring the movements of ships, the authorities can detect suspicious behaviour and take appropriate measures to ensure the safety of territorial waters. AIS data can also be used for research and analysis. Scientists and researchers can study ship traffic patterns, environmental trends and other aspects of shipping to better understand the challenges and opportunities facing the maritime sector.

De Vreede has made a thesis on the management of historical AIS data [7]. The aim of the research is to assess the benefits of database structures for storing and querying large historical mobility datasets. One of the open questions in this thesis is: *What is AIS data, what are its features ?* This question allowed to have a better understanding of this data source and how to use it better.

Danish AIS data contains the following information:

- **Timestamp**
- **Type of mobile**
- **MMSI**
- **Latitude**
- **Longitude**
- **Navigational status**
- **ROT**
- **SOG**
- **COG**
- **Heading**
- **IMO**
- **Callsign**
- **Name**
- **Ship type**
- **Cargo type**
- **Width**
- **Length**
- **Type of position fixing device**
- **Draught**
- **Destination**
- **ETA**
- **Data source type**
- **Size A**
- **Size B**
- **Size C**
- **Size D**

2.3.2 General Transit Feed Specification

General Transit Feed Specification (GTFS) is an open standard used to represent public transport timetables and information. This format was developed by Google to facilitate the integration of transport data into online applications and services. GTFS data provides a set of files that describe stops, routes, schedules, timetables and other essential information about public transport services. This data allows developers to create applications and tools that help users plan their journeys, find out bus, metro and train timetables, and optimize their trips.

GTFS consists of two main parts: GTFS Static and GTFS Realtime.

- GTFS Static contains information on public transport routes, timetables, fares and geographical details. It is presented in simple text files. This straightforward format makes it easy to create and maintain without the need for complex or proprietary software.
- GTFS Realtime contains real-time information about current trips, current vehicle positions and service alerts. This format uses Protocol Buffers for its structure and transmission.

One of the most iconic transport agencies using GTFS data is the Metropolitan Transportation Authority (MTA) in New York, which manages the city's public transport network, including Brooklyn's public transport. The MTA's GTFS data is publicly available, encouraging the creation of a multitude of applications and services for users. Developers can use this data to create user-friendly mobile applications, interactive websites and intelligent tools that enhance the public transport experience for New York travellers and visitors to the city.

2.4 WebGL

WebGL (Web Graphics Library) technology represents a revolutionary advance in the creation and visualization of 3D graphics within web browsers. Developed from the OpenGL specification, a graphics programming interface widely used in the industry, WebGL has made it possible to transfer the power of real-time graphics rendering to the world of the web. This technology has opened up a multitude of possibilities for visualization, online gaming, interactive 3D modelling and much more.

The essence of how WebGL works lies in its ability to use the graphics processing capabilities of a user's graphics card to generate 3D images in a web browser, without the need for external plugins. Modern browsers that support WebGL provide a 3D rendering context, allowing developers to use JavaScript to manipulate geometric elements, textures, shaders and lights to create interactive three-dimensional scenes. WebGL has a wide range of applications. One of the most notable uses is in online gaming. Developers can create 3D games directly in browsers, eliminating the need for complex downloads or installations. This opens the door to instant

gaming experiences that are accessible to a large audience. In addition to entertainment, WebGL also has applications in education and training. Interactive 3D models can be used to simulate scientific experiments, architectural demonstrations, human anatomy courses and many other subjects. Design and modelling professionals are also taking advantage of WebGL to create product visualizations, architectural models and even online art exhibitions.

WebGL has also brought about a significant transformation in the field of cartographic visualization. By combining the power of WebGL technology with geospatial data, it is now possible to create interactive maps directly in web browsers. This approach is transforming the way we explore and understand geographic information. Developers can integrate three-dimensional elements into maps, allowing users to explore virtual landscapes, analyze topographic relief with precision, and visualize complex geographic data in an intuitive way. Whether for urban planning, environmental modeling or navigation, WebGL offers a new range of possibilities for bringing maps to life and offer interactive map visualization experiences.

2.5 OpenLayers

2.5.1 Definition

The definition that developers provide us is the following:

> *« **OpenLayers** is a modular, high-performance, feature-packed library for displaying and interacting with maps and geospatial data »* [1]

In simple terms, OpenLayers is a JavaScript library that handles geospatial data. It allows to easily display maps on the web. In order to achieve that, OpenLayers makes use of 4 basic concepts:

- **Map**: The main component of OpenLayers is the map. It takes a target container and renders the map inside of it (For example a `<div>` element).
- **View**: While the map sets a target element to render the data, the role of the view is to describe the properties of the map such as the center, the zoom and more.

  The view also handles **projections**. Map projections allow us to flatten the globe in 2 dimensions but there are many ways to represent a sphere on a 2-dimensional surface as seen in Figure 2.8.[2]

  In OpenLayers, the default projection is the Spherical Mercator represented by the code: **EPSG:3857**
- **Source**: The source class allows to get remote data for a layer. OpenLayers supports GeoJSON, MVT, GML, map data from any source using OGC standards and more.

[1] *https://openlayers.org/en/latest/doc/tutorials/background.html*
[2] *https://gisgeography.com/map-projections/*

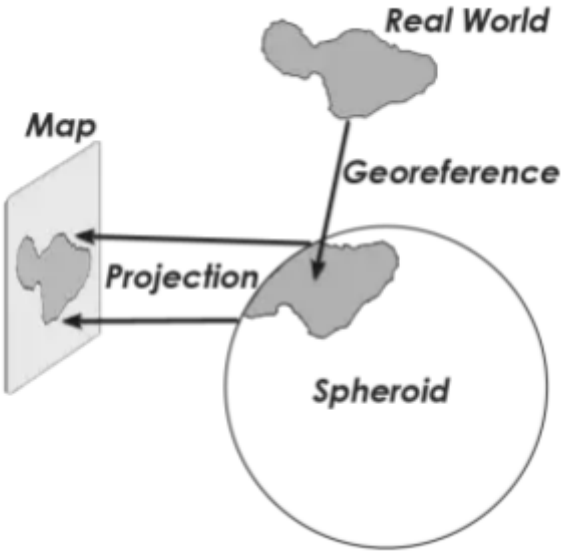


Figure 2.8: How projections work

- **Layer**: The layer is the visual representation of the data provided by the source. OpenLayers has 4 types of layers:
  - **Tile**: Renders sources that provides tiled images
  - **Image**: Renders sources that provide map images
  - **Vector**: Renders vector data
  - **VectorTile**: Renders data that is provided as vector tiles

2.6 Comparison with Other Tools

2.6.1 Leaflet

Leaflet is an open source JavaScript library (like OpenLayers) for interactive maps. It works efficiently on desktop and mobile platforms.[3] The library is small in size, but this is due to its modularity. Indeed, Leaflet does not have a lot of core features, but it is possible to extend the features with plugins. However, if a plugin for a certain feature does not exist, it will require some hard work to implement it. The power of Leaflet lies in its simplicity and lightness. It has an extensive community, is well documented and is open source, but it has been more than 2 years that Leaflet has not been updated.

[3] *https://leafletjs.com/*

2.6.2 Mapbox GL JS

Mapbox is a provider of online custom maps and Mapbox GL JS is a JavaScript library that allows to display Mapbox maps in a web browser and customize the map experience.[4] Mapbox is a pioneer in vector map technologies.
Mapbox GL JS is based on Leaflet and has recently switched to a non open source license. However, a group of developers created MapLibre which is a fork of Mapbox GL JS before the license switch.[5] Mapbox GL JS (and MapLibre) make use of WebGL [23], a technology that enables GPU computation which allows high precision computation.

2.6.3 deck.gl

deck.gl has been specifically designed to facilitate high-performance visualization of large datasets using WebGL, just like MapLibre.[6] It plays a central role in the Vis.gl suite of open source frameworks developed by Uber. This library can be seamlessly combined with other popular JavaScript libraries such as Mapbox GL JS or Google Maps JS, offering a wide range of possibilities for creating map visualizations and interactive experiences.

2.6.4 Summary

The main advantages of OpenLayers remains in its flexibility and the fact that it is open source. It also has an extensive documentation and all the required features are in the core library. However OpenLayers is not as popular as the other libraries and has a heavy codebase. If you want to customize maps, a significant amount of work is required.

While the main advantage of Leaflet is simplicity, it still has some disadvantages. Indeed, Leaflet is not actively developed and if we want to add a feature that does not exist, it will require a lot of work.

Then, Mapbox, a provider of base maps has developed its own JavaScript library (inspired from Leaflet) to display maps named Mapbox GL JS. Mapbox is a pioneer in vector map technologies and has a big community. However the main issue of Mapbox is that it is not open source anymore.

Finally, deck.gl is a framework that allows to visualize large datasets. As it makes use of WebGL, deck.gl is highly performant. Another advantage is that it can be coupled with other popular libraries such as Google Maps JS. However, during my research, I did not find a lot of opinions of experts on this tool compared to the others.

[4] *https://docs.mapbox.com/mapbox-gl-js/*
[5] *https://maplibre.org/*
[6] *https://deck.gl/*

2.7 React

2.7.1 Definition

React.js or simply **React** is a JavaScript library first deployed in 2013 by Facebook and is now maintained by the same company that renamed itself Meta. It is one of the most used front end libraries in the world to build websites. React is open source and is used to build interactive user interfaces. It introduces the concept of components and encourages the reuse of code.

2.7.2 Functionalities

- **JSX**: JavaScript Syntax Extension is a JavaScript extension that allows you to write HTML (with JavaScript) and that makes it easier to insert HTML elements in React. For example, Listing 2.2 describes a code without JSX and Listing 2.3 describes the same code using JSX.

```
1  const myElement = React.createElement('h1', {}, 'no JSX!');
```

Listing 2.2: Code not using JSX

```
1  const myElement = <h1>I Love JSX!</h1>;
```

Listing 2.3: Code using JSX

- **Components**: With React, it is possible to write components that use JSX as a class or function. The components are independent of each other which makes it possible to create several components that we assemble to build our user interface. An analogy would be the construction of a Lego toy.
- **Reutilization**: Components can be reused. Indeed, developers can take a previously developed component and use it in a new application, thus reducing the development effort.
- **SPA**: A Single Page Application is a web application that loads a single document and dynamically updates the current page when a new one is requested by the client. The objective is to obtain an application with fluid and fast transitions compared to the traditional method which is to call the server and load each time the new page.

2.7.3 React and OpenLayers

This thesis focuses on the implementation of mobility data visualization via OpenLayers using React. To this end, a new React application was created exploiting Node.js to integrate OpenLayers as a package. The integration of OpenLayers with React offers a powerful synergy between advanced map visualization and the development flexibility of React. By using Node.js in parallel, we establish a solid foundation for data management and processing, while facilitating communication between application components. This work opens up exciting perspectives in the

visualization of mobility data, enabling researchers and urban planners to explore mobility patterns. By integrating OpenLayers as a package, we offer a ready-to-use solution for geospatial data integration to React developers.

2.8 Related Work

The visualization of mobility data has attracted considerable attention in recent years due to the increasing amount of location based services and tracking technologies. This has led to a generation of vast amounts of trajectory data from moving objects presenting unique challenges and opportunities for the visualization and analysis of this complex spatiotemporal information. Furthermore, the integration of MobilityDB, a powerful temporal database extension, with OpenLayers provides an enhanced platform for effectively handling and visualizing mobility data, enabling real-time tracking and improved analysis of moving objects' trajectories. This synergy empowers researchers and practitioners to gain deeper insights into the intricate patterns within the spatiotemporal data landscape. This chapter offers a comprehensive review of existing techniques and methods for visualizing mobility data especially on OpenLayers using MobilityDB.

2.8.1 Visualization Techniques for Mobility Data

Visualizing mobility data requires various techniques to effectively represent and interpret the trajectories of moving objects. Adrienko et al. [2] present an in-depth review of visual analysis techniques for mobility and transportation. They highlight the importance of interactive visualization tools for learning from large mobility datasets. They draw attention to the need for techniques that combine the dimensions of space, time and other attributes to be able to understand complex mobility patterns.

He et al. [15] have carried out an in-depth study of various visualization techniques and methods for trajectory data of moving objects. Their analysis contains visual representations such as heat maps and density graphs providing valuable information on the spatial distribution and movement patterns of objects over time. These techniques can be used to discover frequently used routes and identify ares of interest.

The study of Zeng et al. [33] focuses on the visualization of mobility in public transport systems. They describe specific approaches for representing real-time transportation data. These visualizations can be used to understand mobility patterns, facilitate transportation or to identify bottlenecks. The results of the research provide interesting perspectives for the development of dynamic visualizations of public transport.

Gomes et al. [10] research presents an approach for visualizing traffic oscillation patterns by visualizing the objects movement in space and time. By exploiting this approach, researchers can design applications offering real time visualization

of road traffic and provide essential information for urban traffic planing and management.

In the field of urban mobility analysis and transport planning, the integration of various data sets is essential to obtain relevant information. In particular, the incorporation of GTFS data into MobilityDB, a powerful temporal database extension, has become a significant research effort. Iliass El Achouchi's master's thesis [8] stands out as a notable contribution in this area, where he is currently developing a novel approach to transparently import GTFS data into the MobilityDB framework. El Achouchi's work addresses the complexities of time varying transport datasets by exploiting MobilityDB's temporal data modeling capabilities, facilitating the efficient storage and retrieval of transport related information.

2.8.2 Visualization with OpenLayers

In 2019, Yuqin He et al. [16] developed an application using OpenLayers for marine information monitoring. The study highlights the use of OpenLayers for visualization while retrieving data in JSON. However, the work does not focus on the temporal dimension. This integration of OpenLayers shows the utility and versatility of OpenLayers in different fields and in this case: marine monitoring.

The work of Sitanggang et al. [27] aims to use OpenLayers to analyze spatial information in the context of agriculutre. They developed an application to monitor and analyze crops. The main objective of the authors was to develop a SOLAP (Spatial Online Analytical Processing) and have used OpenLayers and GeoJSON for the visualization layer. They were able to graphically represent cultivated fields and track their evolution allowing farmers (or decision makers) to identify areas with growth problems resulting in a better use of agricultural resources and increased harvests.

Similarly, Pushkarev and Yakubailik [25] carried out a study focusing on the integration of spatial analysis with OpenLayers for agricultural monitoring. They developed a web application for visualizing, analyzing and processing spatiotemporal data. The application allows its users to explore large scale environmental data. They access their information through multiple APIs. For example, the users can observe variations due to climate change. The integration of OpenLayers facilitated access to the data and allowed researchers and decision makers to have a better understanding of environmental issues.

These two studies demonstrate how OpenLayers is beneficial as a visualization platform for spatial analysis. While the research of Pushkarev an Yakubailik [25] prove that it is possible to visualize spatiotemporal data using OpenLayers as a visualization platform, it can be enhanced with more advanced features.

2.8.3 Visualization with MobilityDB

In the field of visualization and data analysis of moving objects, significant contributions have been made to improve our understanding of complex spatiotemporal data. This section explores related work that has paved the way for advances in this field, with particular focus on the incorporation of MobilityDB. In particular, the following research offers valuable insights into the visualization and analysis of mobility data:

Fabricio da Silva's thesis [26] at the Berlin University of Technology (TU Berlin) focuses on the field of visual analysis applied to databases of moving objects. The main objective of this research is to find efficient ways to visualize and interact with data about moving objects in space and time using MobilityDB. The author has taken an innovative approach by integrating the deck.gl library and Kepler.gl to facilitate the visualization of this complex data. deck.gl is a powerful geospatial visualization library that creates interactive visual representations that uses the WebGL technology. On the other hand, Kepler.gl is another library that focuses specifically on geospatial data visualization and is designed to be user-friendly and configurable. Using these two libraries, the author has succeeded in creating dynamic and attractive visualizations. Mobility data, such as trajectories, were presented in a clear and informative way thanks to this approach. The use of Kepler.gl also played a major role in the creation of convincing visual representations, reinforcing the value of this thesis' contributions to the analysis and visualization of databases using MobilityDB.

The aim of Ludéric Van Calck's work [29] was to address the issue of automatic visualization of moving objects using the desktop application QGIS. QGIS is renowned for its ability to visualize, edit and analyze geospatial data. In this context, the aim of the project was to explore various methods for automatically visualizing these moving objects within QGIS. To this end, several experiments were carried out with the aim of determining the most effective method for achieving this objective. The results obtained through these experiments were studied in order to identify optimal solutions for the fluid and accurate visualization of MobilityDB data within the QGIS environment. This project is of particular importance in the field of geospatial visualization, opening up new possibilities for analyzing and understanding moving objects in a variety of contexts.

The thesis written by Florian Baudry [3] focuses on the development of an innovative approach for visualizing spatial and temporal data from a database using MobilityDB. This approach is based on the integration of Leaflet and React technologies to provide an interactive, high-performance platform for visualizing moving objects on a map. The combination of Leaflet and React provides a modern, responsive user interface for visualizing spatial and temporal data. Leaflet is a JavaScript library widely used for creating interactive maps in web applications. By integrating it with React, a popular library for building user interfaces, it creates a dynamic experience for users. The approach developed by Baudry offers significant advantages for the visualization of mobility data. By exploiting Mobili-

tyDB's spatiotemporal management capabilities and combining them with modern visualization technologies such as Leaflet and React, this approach promises to provide a powerful platform for exploring and analyzing spatiotemporal data. While the thesis is still a work in progress, the results obtained so far indicate a promising potential for the research and industrial community interested in the visualization and analysis of mobility data.

Chapter 3

Simple Performance Test

The OpenLayers, MapLibre, deck.gl and Leaflet JavaScript libraries are of significant importance in the field of spatial data visualization. In this dedicated section, the objective is to carry out an in-depth comparative evaluation of these different frameworks. To do this, we use a variety of datasets. The purpose of our approach is to identify the framework that stands out for its superior performance and ability to efficiently handle the visual representation of spatially related data. Table 3.1 contains the resources of the computer used for the performance test.

CPU	Intel Core i5-9400F @ 2.90 GHz
GPU	NVIDIA GeForce GTX 1660
RAM	8 GB DDR4
DRAM Frequency	1333 MHz
OS	Windows 10
Browser	Firefox 98.0.1 (64 bits)
Web app	Parcel

Table 3.1: Ressources used for the performance test

3.1 Dataset Management

In this section, a performance test has been executed on OSM data extracts that come from the Geofabrik download server. For each dataset, there is an image showing what OpenLayers has displayed and a graph comparing the execution/loading times (in ms) of each tool on 10 executions. The performance test has been executed on datasets (all of them are in the GeoJSON format) containing of different type of points and are defined by OSM as: [1]

- **Cities**: *The largest urban settlement or settlements within the territory*
- **Towns**: *Between a city and a village in size*
- **Villages**: *A smaller distinct settlement, smaller than a town with few facilities available with people traveling to nearby towns to access these*

[1] *https://wiki.openstreetmap.org/wiki/Main_Page*

- **Hamlets**: *A smaller rural community, typically with fewer than 100-200 inhabitants, and little infrastructure*

Firstly, the data extract was downloaded from the Geofabrik download server.[2] The idea was to get the European data (25 GB) and then to upload it on a PostgreSQL database that has the MobilityDB extension with the help of the `osm2pgsql` command.[3] This command allows to import OSM data into a PostgreSQL/PostGIS database.

```
1  osm2pgsql -c -H localhost -U souf -W -d MobilityDB -C 7000 belgium-
     latest.osm.bz2
```

Listing 3.1: Upload of belgium OSM data

Listing 3.1 describes the command used to upload the belgian dataset and here is a description of the various parameters used in this command:

- `-c`: indicates to `osm2pgsql` to perform a complete import by deleting the existing data in the database and reimporting everything from the OSM file.
- `-H localhost`: Specifies the host (in this case, `localhost`) where the PostgreSQL database is located.
- `-U souf`: Indicates the user name (in this case, `souf`) that will be used to connect to the PostgreSQL database.
- `-W`: This option asks `osm2pgsql` for the password of the PostgreSQL user. After entering the command, the user will be prompted to enter the password.
- `-d MobilityDB`: Specifies the name of the PostgreSQL database (in this case, `MobilityDB`) into which the OSM data will be imported.
- `-C 7000`: This defines the maximum number of megabytes of memory that `osm2pgsql` can use during import. In this case, it is set to 7000 MB (or 7 GB) because the computer used has a maximum 8 GB.
- `belgium-latest.osm.bz2`: is the name of the compressed OSM file in `Bzip2` format containing geographical data for Belgium. OSM data is generally available as compressed files to reduce the size of the download.

However, with the computer used for this experiment, the upload was taking a long time so we decided to upload only half of the data.
In order to do that the following countries have been selected:

- **Belgium**: 471 MB
- **Netherlands**: 1.1 GB
- **Spain**: 994 MB
- **Great Britain**: 1.4 GB

[2] *https://download.geofabrik.de*
[3] *https://osm2pgsql.org/*

- **Germany**: 3.6 GB
- **France**: 3.9 GB

With a total size of around 11.5 GB (which is almost half of Europe's data size).

When data from OpenStreetMap (OSM) is imported into a PostgreSQL database, several tables are created to store this geospatial information. One of these tables, called `planet_osm_point`, contains information about points of interest, such as cities, towns, and more. Views are created from this table so that data can be extracted through the command `ogr2ogr`. [4] Listing 3.2 and Listing 3.3 contains an example of a SQL query used to create the view of the belgian data and the `ogr2ogr` command used to etract the data as GeoJSON.

```
CREATE OR REPLACE VIEW cities AS
  SELECT way, place, name
  FROM planet_osm_point
  WHERE planet_osm_point.place LIKE 'city';
```

Listing 3.2: SQL query used to create a view

```
    ogr2ogr -f "GeoJSON" ./be_cities.geojson PG:"host=localhost user=
   souf password=souf dbname=MobilityDB" cities
```

Listing 3.3: Command used to extract GeoJSON data

- `-f "GeoJSON"`: This option specifies the desired output format, which in this case is GeoJSON. This means that the data will be converted to GeoJSON.
- `./be_cities.geojson`: This is the path of the output file where the converted data will be saved in GeoJSON format. In this case, the file will be named `./be_cities.geojson`.
- `PG:"host=localhost user=souf password=souf dbname=MobilityDB"`: This part specifies the source of the data to be converted, which is a PostgreSQL/-PostGIS database.
- `cities`: This is the last argument in the command. It is the name of the specific table in the PostgreSQL database from which the data will be extracted to be converted into GeoJSON. In this case, the data comes from the view named `cities` created earlier.

However there was still an issue in order to have a clean dataset. The extracted GeoJSON file was in the `EPSG:3857` coordinate system and not all the tools were able to handle this coordinate system implicitly. In order to resolve this problem it was required to convert from EPSG:3857 to the `EPSG:4326` with the help of an online converter.[5]

[4] *https://gdal.org/programs/ogr2ogr.html*
[5] *https://mapshaper.org/*

3.2 Performance Test

The implementation of the benchmark can be found in a github repository and can be easily reproduced by following the steps described in Section 3.1 indicated.[6] Before starting the benchmark, certain aspects need to be clarified to ensure that the work is properly understood:

- All the SQL queries and commands used are detailed in the appendix (`benchmark_views.sql` and `bench_commands.bash`)
- In this Chapter, loading time means the time from the instant where the webpage starts loading to the instant where everything is displayed.
- Leaflet has not been benchmarked with the villages dataset and the hamlets dataset because it was not able to display the data even after 15 minutes of loading.

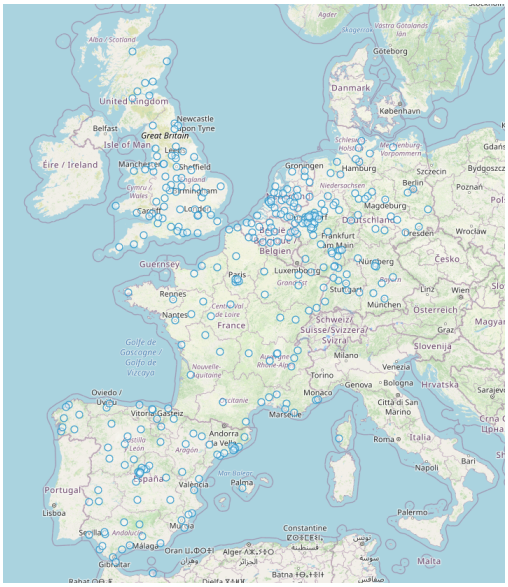
3.2.1 OpenLayers Display

The Table 3.2 presents the number of cities, towns, villages and hamlets per country, providing a clear perspective of the volume of data involved. In parallel, Figure 3.1 illustrates how these different data elements are represented on OpenLayers. These elements combine to provide an overview of the geospatial data landscape. It should be noted that the benchmark was carried out taking into account the total number of data points. The values presented were extracted via a query SQL when the data resided in the PostgreSQL database :

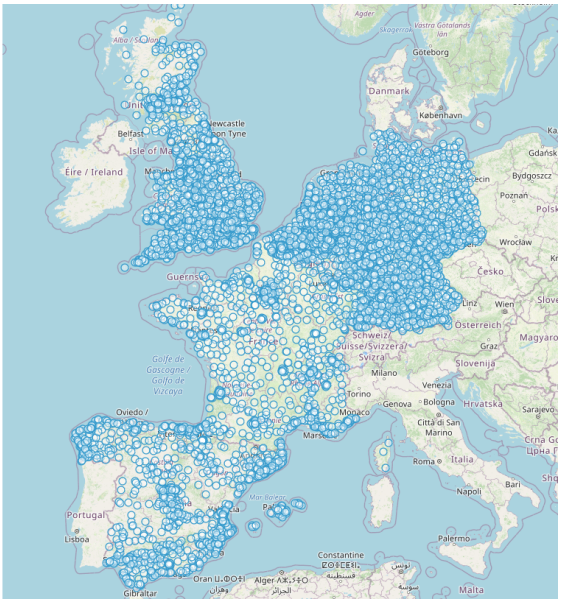
	Cities	**Towns**	**Villages**	**Hamlets**
Belgium	13	367	3 017	2 113
Netherlands	34	273	2 152	3 780
Spain	80	924	9 513	45 108
Great Britain	66	1 525	14 671	17 823
Germany	86	2 383	37 568	44 182
France	50	1 021	37 401	208 605
Total	**329**	**6 493**	**104 322**	**321 611**

Table 3.2: Number of points

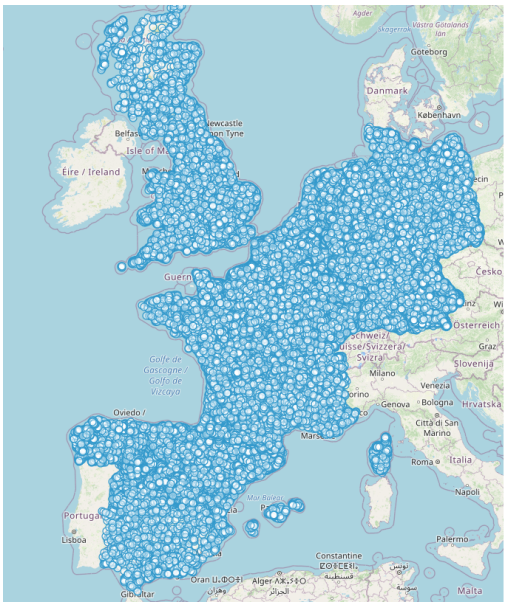
[6] *https://github.com/SoufianBk/perftest*



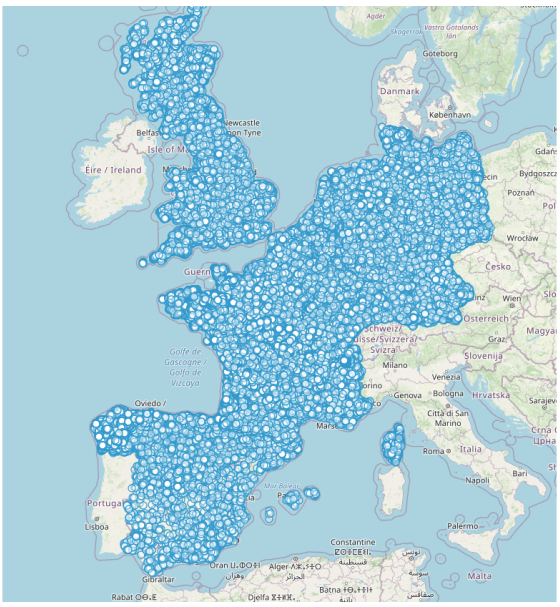
(a) Cities.json



(b) Towns.json



(c) Villages.json



(d) Hamlets.json

Figure 3.1: Display of various GeoJSON data in OpenLayers

3.2.2 Result

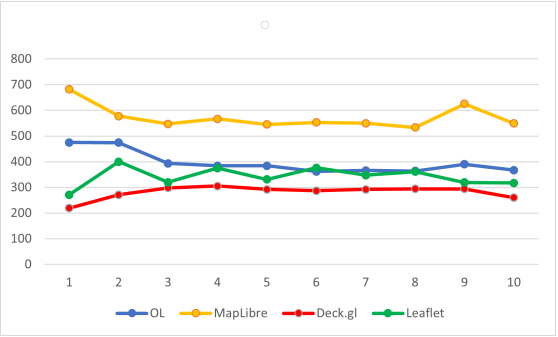
Figure 3.2 corresponds to graphs showing the loading time of the data on OpenLayers on 10 different executions but the last graph (Figure 3.2e) summarizes the loading times into an average. As we don't really see the difference between all the tools in the cities and towns in Figure 3.2e, Figure 3.2f contains the average loading times of the cities and towns.

The analysis of the execution times of OpenLayers, MapLibre, deck.gl and Leaflet allows us to compare it objectively with its capacities for loading GeoJSON data of various sizes: from the 300 cities to the 300,000 hamlets. The results provide essential information to help developers choose the tool best suited to their needs. When evaluating these geospatial libraries and frameworks, it is essential to consider not only their performance, but also their scalability, ease of use and compatibility with the desired functionality. Comparing loading times under different data loads gives developers an in-depth understanding of how each tool handles different levels of complexity and data volume.

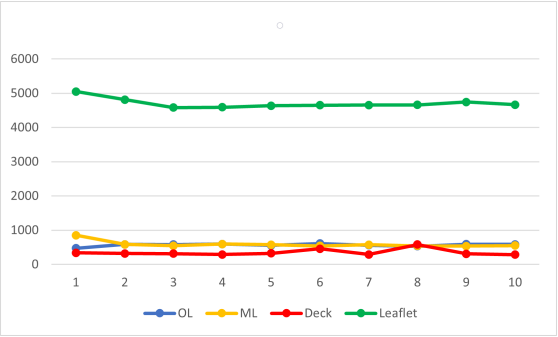
This analysis revealed that Leaflet, known for its simplicity and ease, was still competitive when handling 300 cities. However as the dataset grew to 6,500 towns, Leaflet's limitations in loading times started to show as seen in Figure 3.2f. Loading times slowed considerably and slowed even more when processing 100,000 villages where Leaflet took more than 15 minutes to display the data. (That is the reason why there is no benchmark of Leaflet on the villages and hamlets) This can be explained by the fact that Leaflet is a library made for mobile applications and might not be optimized for managing large datasets. These observations highlight the importance of taking into account the performance and limitations of each tool depending on the size of the data and the specific requirements of the project. For cases where the manipulation of large geospatial datasets is crucial, other options such as OpenLayers, MapLibre or deck.gl may offer more suitable performance thanks to their focus on the visualization of complex data and their optimization for web environments.

Comparing the results with OpenLayers, MapLibre and deck.gl, it was observed that MapLibre and deck.gl outperformed OpenLayers, and this difference widened with larger datasets. This disparity can be attributed to the fact that, unlike OpenLayers, MapLibre and deck.gl use WebGL. WebGL is a technology that exploits the potential of the GPU to accelerate the rendering of web pages, resulting in improved performance. The use of WebGL allows MapLibre and deck.gl to better take advantage of the parallel processing capabilities offered by the GPU, which is particularly beneficial when handling large amounts of geospatial data. WebGL's ability to delegate certain rendering tasks to the GPU reduces the load on the CPU, resulting in smoother and faster execution, especially when visualising large and complex data.

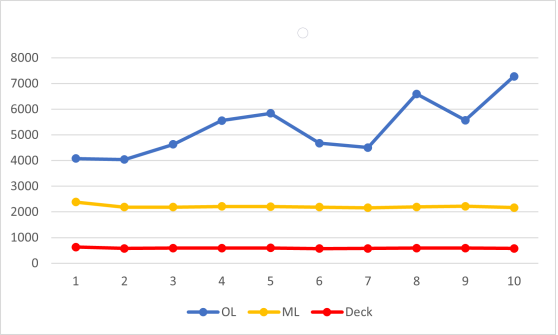
An interesting observation from this analysis is the consistent evolution of loading times as the amount of data increases for OpenLayers, MapLibre and deck.gl as



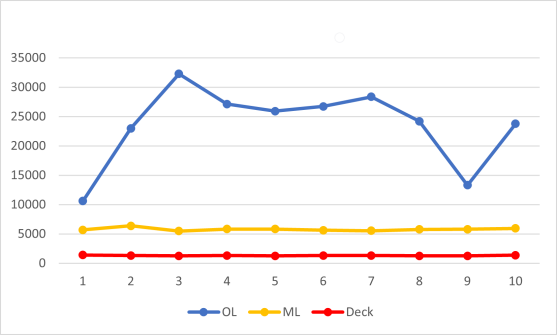
(a) Loading times for cities



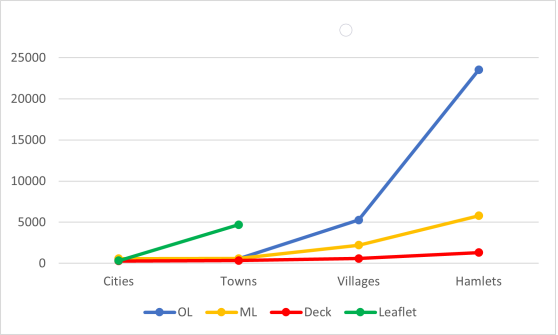
(b) Loading times for towns



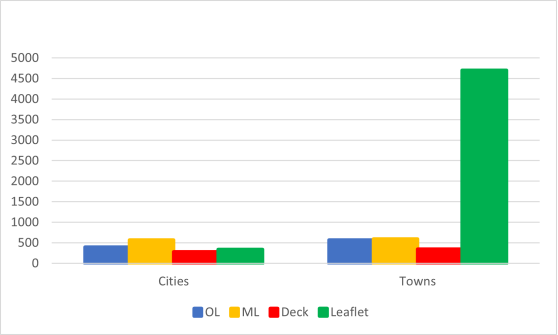
(c) Loading times for villages



(d) Loading times for hamlets



(e) Average loading times



(f) Average loading times of cites and towns

Figure 3.2: Loading times of each the cities, towns, villages and hamlets datatesets on OpenLayers, MapLibre, deck.gl and Leaflet in ms

illustrated in Figure 3.2e. Even with the use of WebGL, MapLibre and deck.gl also showed a linear increase in loading times as datasets increased in size. This highlights the challenge of managing large datasets and underlines the importance of taking these factors into account when choosing a mapping library. This observation reveals that performance when processing large scale geospatial data is a common concern, even for technologies that take advantage of advanced rendering technologies such as WebGL. Geospatial data can present unique challenges in terms of manipulation, visualization and rendering, particularly when datasets become large and complex. This can be due to the need to manage interactions, calculate geospatial transformations and render map elements accurately.

In the process of selecting a mapping library, it is essential to consider various factors, such as performance, functionality, ease of use, scalability and the specific needs of the project. By keeping in mind that each tool has advantages and disadvantages based on these criteria, developers can make wise decisions to create high performance, responsive geospatial applications that meet the needs of their users.

Chapter 4

Implementation

4.1 Vanilla JS

4.1.1 Berlin

Fabrício Ferreira da Silva wrote a master thesis dealing with a subject similar to that of this thesis.[26] In his work he conducted experiments using a data generator that produces mobility data in Berlin and made his implementation public.[1] On his repository, Fabrício generated a GeoJSON dataset that was also used in these experiments.[2]

In Fabrício's dataset, trips are represented by a `LineString` object that contains 2 lists. The first one indicates all the coordinates through which this object has passed and the second one (which has the same size as the first one) contains the timestamps of the coordinates. For example, let us take the first element of the list of coordinates $[x, y]$ and the first element of the list of timestamps t, then it means that the object was located at $[x, y]$ at the instant t.

```
var vectorsource = new VectorSource({
  url: file,
  format: new GeoJSON()
});
var vectorlayer = new VectorLayer ({
  source: vectorsource ,
  style: new Style({
    stroke: new Stroke({
      color: 'rgba(255, 255, 255, 0)'
    })
  })
});

map.addLayer(vectorlayer);
```

Listing 4.1: Creation of the GeoJSON layer

Listing 4.1 contains the code concerns the creation and integration of a vector layer within an OpenLayers map. The purpose of this vector layer is to display geographic data in GeoJSON format. The code starts by instantiating a vector source using the `VectorSource` class. This source is configured using a URL that

[1] *https://github.com/MobilityDB/MobilityDB-Deck*
[2] *https://github.com/MobilityDB/MobilityDB-Deck/geojsonvt/data/trips0_005.json*

points to the GeoJSON file of Fabricio containing the spatiotemporal data to be displayed later on the map. At the same time, the format of this data is defined by instantiating a `GeoJSON()` object which indicates the nature of the data format contained in the file. Once the vector source has been set up, the next step is to create a vector layer using the `VectorLayer` class. This layer is directly connected to the vector source established earlier. In this way, the layer will be able to display the geographic data contained in the GeoJSON file on the interactive map. The code goes on to configure the visual style applied to the newly created vector layer. This style is defined using the `Style` class.

After configuring the vector layer as described above, Listing 4.2 implements a function that responds to the vector layer's `postrender` event. This event is triggered once the layer has been rendered on the map.

```
vectorlayer.on('postrender', function (event) {
  ... // Timestamp management
  map.render();
});

map.render();
```

Listing 4.2: Making the map dynamic

The main idea is to create a system that displays all the coordinates corresponding to a certain instant. As the timestamp increases, the old values are replaced by new, updated coordinates. To update these values, a combination of the `map.render()` function and OpenLayers' `postrender` event is used.

In this implementation, the `map.render()` function is responsible for drawing the map and keeping it up to date. The `postrender` event is triggered after each rendering of the map, allowing a custom function to be executed to update the coordinates displayed. Then, this custom function executes a `map.render()` to trigger the `postrender` event until it reaches the last timestamp.

When the timestamp changes, the `postrender` event is activated, triggering the coordinate update function. This function retrieves the new coordinates corresponding to the current date and replaces them on the map. In this way, the old values are replaced by the new ones, providing a real-time view of the geographical data. There are several advantages to using the `postrender` event. For example, it ensures that the coordinates are updated after each map rendering, ensuring precise synchronization with timestamp changes.

4.1.2 Danish AIS

The second experiment uses AIS ship data from Denmark and is provided in CSV format and not in GeoJSON. The Danish Maritime Authority publishes AIS routes every day but for the sake of this experiment only a single CSV file that represents a day (or a month for older data) has been used.[3] The AIS dataset contains a lot

[3] *http://web.ais.dk/aisdata/*

features explained in Section 2.3.1 but the most important features of the dataset are the timestamp, MMSI, latitude and longitude.

The implementation of the visualization of these data is similar to the Berlin implementation as discussed above. The main difference is that the data do not have the same basic structure. A conversion was therefore carried out in order to generalize the implementation.

MobilityDB offers a workshop to understand how the extension works and makes use of AIS data.[4] That's why in this experiment, a part of the workshop was followed in the data processing. In particular the part on the upload of the data and the transformation of the latitude and longitude into an object of type `geometry` (All the queries coming from the workshop are in the appendix `mobilitydb.sql`) The idea of the workshop was to get the ships route data and then to upload it on a PostgreSQL database with the help of the `COPY` SQL function.[5] The first step is to convert the latitude and longitude from the `double` type to the `geometry` type. The `ST_MakePoint()` function from PostGIS allows to do the conversion and store results into a new column.[6]

Furthermore, in the context of a version of the experiment which focuses exclusively on the use of vanilla JS, without the need for a back end server, the process of obtaining the data remains similar to that detailed in Section 3.1. Once the AIS data has been successfully collected, the process of displaying it follows a similar trajectory to that used for the Berlin data as detailed in Section 4.1.1. In this way, the previous experience gained during the implementation for the Berlin data is naturally transposed to this specific configuration.

All the commands and SQL queries used in this implementation are available in the appendix in the name of `ais.sql, ais.bash` and `mobilitydb.sql`. The implementation can be found in a github repository as well with a visualization.[7]

For the sake of visibility, only the trips that went through the Skagerrak, the Kattegat and the south western side of the Baltic sea have been kept. Figure 4.1 illustrates the area in which we keep the values and has been generated by Google Maps. In order to achieve that, a SQL view was created keeping the points of the ships that were between:

- 53.76 and 60.46 of latitude
- 5.35 and 16.70 of longitude

A trip is a set of points. Hence, the points were grouped by the MMSI. In other words, the different locations of a ship were grouped all together in a single row and ordered by the timestamp. The next step was to transform and group all the `geometry` values that represented the points of a trip into another value of type

[4] *https://www.mobilitydb.com/tutorials.html*
[5] *https://docs.postgresql.fr/10/sql-copy.html*
[6] *https://postgis.net/docs/ST_MakePoint.html*
[7] *https://github.com/MobilityDB/MobilityDB-OpenLayers*

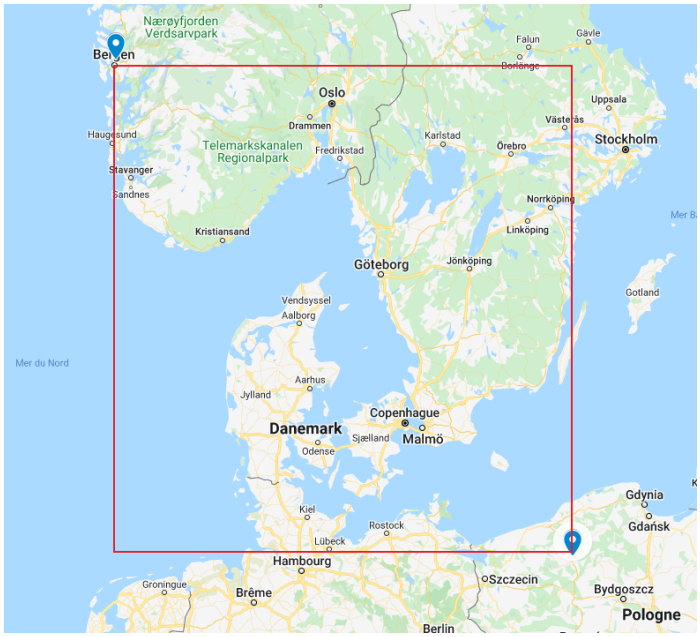


Figure 4.1: AIS dataset boundaries

`geometry` but this time representing a Line. Finally, with the help of the command `ogr2ogr` the AIS data from the database was extracted to a GeoJSON file that has a similar structure to the dataset used by Fabrício.[8]

4.1.3 Source Code Structure

When developing an application using React, it's important to organize the code in a clear and coherent way to ensure easy maintenance and optimal scalability. Here's a complete description of the structure of the code that displays mobility data on OpenLayers using React, including explanations of each key element:

- `node-postgres/`: This directory contains the code associated with the application's server component, which uses the PostgreSQL database to ensure that it functions correctly. Here are the elements needed to establish communication between the user interface and the database, ensuring that data is retrieved and handled smoothly, securely and efficiently.
  - `dbqueries.js`: This file contains the SQL queries and associated logic for interacting with the database. It manages communication between the application and the database, performing operations such as data retrieval.
  - `index.js`: The API endpoints are configured and created in this file. These endpoints are the gateways through which HTTP requests are received and processed. This file acts as an organizer, listening to the requests, calling the appropriate `dbqueries.js` functions and returning

[8] *https://gdal.org/programs/ogr2ogr.html*

the responses to the clients. It serves as the essential link between the user interface and the database.

- `src/`: The `src` folder is the heart of the front end application, where the user interface is built using React. This is where components, styles and functionality combine to create an interactive user experience.
  - `mfjson/`
    - `MapJSON.js`: This file contains the code for manipulating and displaying data in MFJSON format. It contains functions for extracting, transforming and displaying spatiotemporal data in the application.
  - `mvt/`
    - `MapMVT.js`: It contains the code for working with data in MVT format. It contains functions for decoding and displaying vector tiles in the application.
  - `App.js`: It represents the root component of the React application. It brings together different components to build the overall structure of the user interface.
  - `Geo.js`: This file defines certain configuration information for OpenLayers.
  - `index.js`: It plays an important role in starting the React application. It renders the App component in the HTML entry point (for example, a div with the id "root").
- `public/`: This folder is generated automatically by React and contains the static files that will be distributed with the application. This generally includes the basic HTML file (`index.html`) as well as other static resources such as images or icons.
- `.gitignore`: The `.gitignore` file is important for excluding certain files and folders from the Git repository. This prevents sensitive files, local configuration files or generated files from being included in the repository. It helps to keep the repository clean and to avoid sharing private information or large folders such as packages generated by `npm install`.
- `package.json`: It is created by React and acts as a configuration for the application. It contains metadata about the application, dependencies, management scripts and other important information.
- `package-lock.json`: It is automatically generated when dependencies are installed or updated via NPM. It records the exact version details of the dependencies and their sub-dependencies to ensure reproducibility of future installations.

- `README.md`: It is a text file in Markdown format used to provide information about the project to other developers and users. It can contain installation instructions, usage examples, details of the code structure, screenshots and much more.

4.2 React

This implementation has the same objective as the implementation used to display AIS in vanilla JS but with improvements. First of all this solution makes use of the **React** framework (explained in Section 2.7) and instead of displaying the data directly from a file, we access the database directly. This implementation has 2 different architectures, the first uses the GeoJSON format and the other uses MVT format, and both access the same data that has been loaded onto the database as explained in the Section 4.1.2. An extra step from the MobilityDB workshop has been executed which is running the query described in the `ships.sql` appendix. This query creates a table `Ships` that contains the spatiotemporal trip and the MMSI of each ship. Figure 4.2 is a visualization of all the trips used in the implementation on OpenLayers.

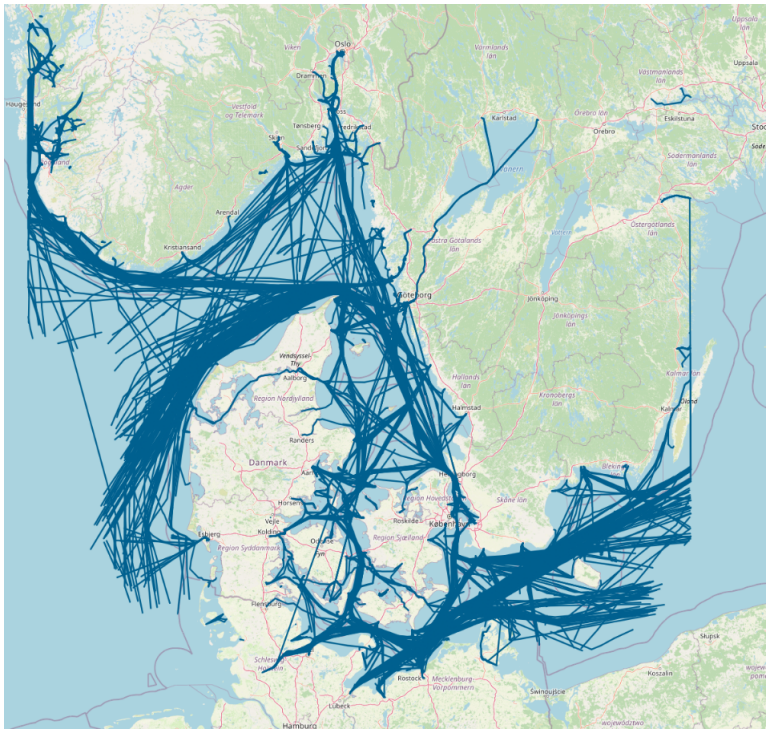


Figure 4.2: Preview of the trajectories of all the ships

4.2.1 GeoJSON

One of the biggest challenges when implementing GeoJSON data visualization on OpenLayers with React lies in accessing the database. Indeed, as React is primarily

a client-side framework, it doesn't intrinsically offer a direct and fluid solution for establishing a connection to the database, which becomes an essential aspect when it comes to spatial data visualization. In this context, integrating GeoJSON data visualization within the OpenLayers and React framework requires a thoughtful approach to overcoming this obstacle. This means developing an architecture that bridges the gap between the React front end and the database where the spatial data resides. This often involves the use of intermediate technologies, such as server-side APIs or middleware, to facilitate the retrieval and transfer of data between the React application and the database. The challenge is not only to establish the connection itself, but also to ensure that the process is efficient, secure and reliable.

This is why a server using the Express technology has been setup for our React application. Express.js [28] is a popular web framework for developing server side applications using JavaScript. It is a minimalist and flexible framework built on top of Node.js, which is a JavaScript based server-side development platform. A connection to the database can be then created. In order to do so, some database credentials are required such as username, password, IP address, etc. Listing 4.3 shows how the credentials are provided in the code.

```
const Pool = require('pg').Pool
const pool = new Pool({
  user: 'postgres',
  host: 'localhost',
  database: 'mobilitydbdev',
  password: 'postgres',
  port: 5432,
});
```

Listing 4.3: Database credentials

Then, API endpoints on the server are created. These endpoints are URLs that allow you to perform CRUD (Create, Read, Update, Delete) operations on the data in the database. (In our case, it is just necessary to read data.) For example, it is possible to have a `/api/trips` endpoint that returns the number of trips in the database. Here is a list of the endpoints that are used in the implementation:

`/json`	Returns JSON data containing all the trips
`/json/ts`	Returns JSON data containing the max and min timestamps of the data

Table 4.1: API endpoints for JSON data

When the `/json` endpoint is called, the server sends the query described in Listing 4.4 to the database to get the trip data.

```
SELECT asMFJSON(transform(trip, 4326), 2, 2)::json
FROM Ships
```

Listing 4.4: Returns the trips in MFJSON

We get two main parts by deconstructing this query:

- `transform()`:[9] A function that allows to transform the coordinates to a different Coordinate Reference System (CRS). In our case, we transform our data to one of the most used CRS which is the World Geodetic System 1984 defined by the EPSG:4326 code.
- `asMFJSON()`:[10] This function allows to get data in MFJSON format. The first parameter corresponds to the values we want to obtain, the second parameter limits the number of digits after the decimal point and the third and last parameter allows to add to the output the CRS.

When executing the query on an example of a trip of a boat that did not leave the port for a day. The following result can be observed:

```
{
    "type": "MovingPoint",
    "crs": {
        "type": "name",
        "properties": {
            "name":"ESPG:4326"
        }
    },
    "coordinates": [
        [
        12.69,
        56.04
        ],
        [
        12.69,
        56.04
        ]
    ],
    "datetimes": [
        "2006-02-03T00:00:46+01",
        "2006-02-03T23:56:18+01"
    ],
    "lower_inc": true,
    "upper_inc": true,
    "interpolations":[
        "Linear"
    ]
}
```

For the `/json/ts` endpoint, the query in Listing 4.5 is executed in the database.

[9] *https://docs.mobilitydb.com/MobilityDB/develop/ch04s06.html*
[10] *https://docs.mobilitydb.com/MobilityDB/develop/ch05s12.html#idm3166*

```
SELECT MIN(Tmin(transform(trip, 4326)::stbox)),
  MAX(Tmax(transform(trip, 4326)::stbox))
FROM Ships
```

Listing 4.5: Obtains the lowest and highest timestamp

Again, the function `transform()` is being used. The `Tmin()` and `Tmax()` function are used to obtain the minimum and maximum timestamp respectively for each trip.[11] Then the `MIN()` and `MAX()` functions allow to keep only the lowest/highest timestamp. Those values will be used later on when a slider will be implemented.

In the React application, it is now possible to use HTTP requests to send requests to the server and get the data from the database. And it is through the `fetch()` function built into JavaScript that the front end is now able to make GET, POST, PUT, DELETE, etc. requests to the API endpoints.[12] Listing 4.6 illustrates an example of calling the `/json/ts` endpoint in React.

```
useEffect(() => {
  fetch("http://localhost:3001/json/ts")
  .then(res => res.json())
  .then(
  (result) => {
    setTimestamps(result);
    setIsLoaded(true);
  }
  )
}, [])
```

Listing 4.6: API call from the front end in React

Finally, the implementation of the visualization is similar to the one carried out in the first solution except that the data we receive is in MFJSON format and not in GeoJSON. This implies that the trips are not of the `LineString` type but of the `MovingPoint` type defined by MFJSON and unfortunately OpenLayers only supports the GeoJSON format. A workaround was to manually change the type (`MovingPoint` to `LineString`) and it works perfectly. However the best solution would be for the OpenLayers developers to provide support for MFJSON.

Figure 4.3: Interaction between machines

[11] *https://docs.mobilitydb.com/MobilityDB/develop/ch04s04.html*
[12] *https://legacy.reactjs.org/docs/faq-ajax.html*

Figure 4.3 describes the architecture of the application that allows the visualization of ships moving in the Danish seas in GeoJSON format. The React front end running on port 3000 calls the API (Express.js) running on port 3001 at the following URL: `localhost:3001/json`. This triggers a call to the database which returns the data in MFJSON format allowing it to be displayed.

4.2.2 Vector Tiles

The same problem arises when displaying vector data: React does not allow direct access to the database. It is necessary to go through a back end that takes care of providing the data to our front end application The difference with GeoJSON data is that vector tiles do not display all the data but only the necessary tiles, i.e. the tiles currently visible on the map (illustrated by Figure 4.4). However, using vector tiles can also pose a major challenge. Data is fragmented and distributed within individual tiles, which can lead to a loss of consistency in spatial information. This means that to make effective use of vector tiles, an additional process is required to group and combine this dispersed data into a meaningful representation. This extra step in managing vector tiles is an important difference from GeoJSON data. Listing 4.7 describes the Code of the vector tile extra step.

```
// vector source that will keep the grouped features
var vectorSource = new VectorSource()

var listenerKey = vectorTileSource.on('tileloadend', function (evt) {

  let features = evt.tile.getFeatures();
  features.forEach((feature) => {
    let vsFeatures = vectorSource.getFeatures()
    let isAlreadyIn = false
    vsFeatures.forEach((vFeature) => {
      isAlreadyIn = vFeature.get('mmsi') === mmsi || isAlreadyIn
      if (vFeature.get('mmsi') === mmsi) {
        // appends feature inside vFeature
      }
    });

    // if it is not in the vector source, add the feature
    if (!isAlreadyIn) {
      vectorSource.addFeature(feature)
    }
  });
});
```

Listing 4.7: Code of the vector tile management

In order to obtain the necessary tiles from the database, a different approach to the previous implementation was carried out with the usage of a tile server. A tile server, as the name suggests, is a server that provides map data in the form of tiles. As mentioned in Section 2.2.3, tiles are predefined square images that represent a part of the map at a scale.

In general, a tile is defined by a `z`, `x` and `y`:

- `z` corresponds to the zoom level
- `x` corresponds to the column of the tile
- `y` corresponds to the row of the tile

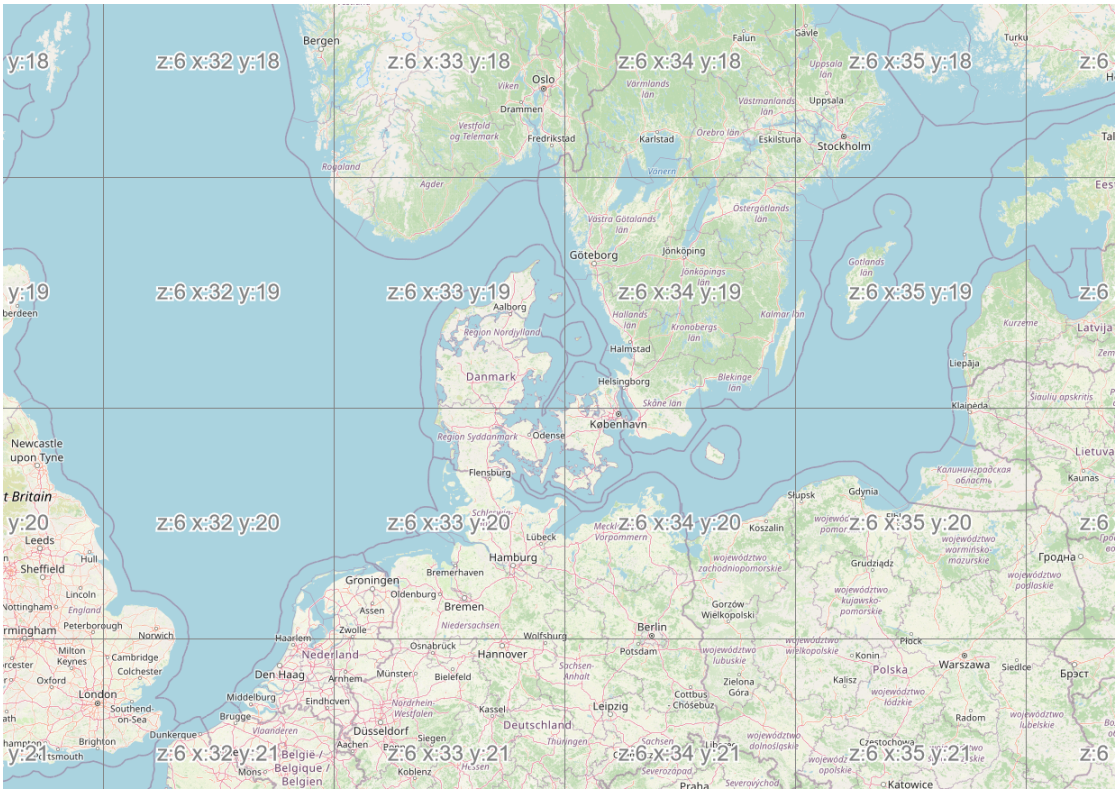


Figure 4.4: Tile layout

pg_tileserv

In our implementation we use **pg_tileserv** [6]. A PostGIS-only tile server which provides an HTTP interface to provide map tiles on demand. This tile server allows to transform an HTTP request into a SQL query to a database that uses the PostGIS extension. pg_tileserv is able to return MVT data in two different ways:

1. by detecting tables that contain a geometry column
2. by creating a function in the database that takes `z`, `x` and `y` as parameters and returns a bytea

It is with the second option that our solution was developed. Listing 4.8 contains the function that was used to return vector tiles.

```
1 CREATE OR REPLACE
2 FUNCTION public.tripsfct(
3       z integer, x integer, y integer)
```

```
RETURNS bytea
AS $$
  WITH bounds AS(
    SELECT ST_TileEnvelope(z, x, y) AS geom
  ),
  val AS (
    SELECT mmsi, asMVTGeom(transform(trip, 3857), (bounds.geom)::
    stbox)
      AS geom_times
    FROM Ships, bounds
  ),
  mvtgeom AS (
    SELECT mmsi, (geom_times).geom, (geom_times).times
    FROM val
  )
  SELECT ST_AsMVT(mvtgeom) FROM mvtgeom
$$
LANGUAGE 'sql'
STABLE
PARALLEL SAFE;
```

Listing 4.8: Returns the requested tile as a byte array

The main point of this function revolves around the invocation of the MobilityDB method `asMVTGeom()`.[13] This method accepts two parameters: the data to be transformed into MVT and the tile bounds. These tile bounds are derived from the `ST_TileEnvelope()` function, which relies on the `z`, `x` and `y` parameters.[14] The outcome of the `asMVTGeom()` function contains a couple composed of a geometry value and an array of timestamp values encoded as Unix epoch.[15] Subsequently, during the third SELECT statement, a separation between the timestamps and the coordinates takes place, facilitating the execution of the `ST_AsMVT()` function.[16] This function is responsible for encoding the data in MVT.

Once the front end receives the MVT data it will now proceed to visualize this data. In order to do that, the use of the `tileloadend` event is of particular importance.[17] When working with mobility data, it is common to represent trips on a map. However, these trips can be scattered across different tiles, which can make them difficult to analyze and interpret. This is where the `tileloadend` event comes in, providing a powerful feature for grouping trips scattered across these tiles.

When we display a map based on vector tiles, each tile represents a small part of the region we are viewing. These tiles are downloaded as we explore the map or zoom the current visualization. Each time a tile is fully loaded, the `tileloadend` event is triggered. By exploiting this event, a custom function is executed at the end of each tile load. This allows to group trips scattered across different tiles and group

[13] *https://docs.mobilitydb.com/MobilityDB/develop/ch05s12.html#idm3824*
[14] *https://postgis.net/docs/ST_TileEnvelope.html*
[15] Op. cit.
[16] *https://postgis.net/docs/ST_AsMVT.html*
[17] *https://openlayers.org/en/latest/apidoc/module-ol_source_Tile.TileSourceEvent.html*

them together for a more complete analysis. For example, suppose we are tracking the movements of a fleet of vehicles on a map. The positions of these vehicles may be spread over several tiles, making it difficult to understand them all. By using the `tileloadend` event, we can group these dispersed positions together and obtain a global view of the routes studied. This is particularly important from a research point of view, where in-depth data analysis is essential. Bringing together fragmented data using the `tileloadend` event enables more precise analyses and interpretations.

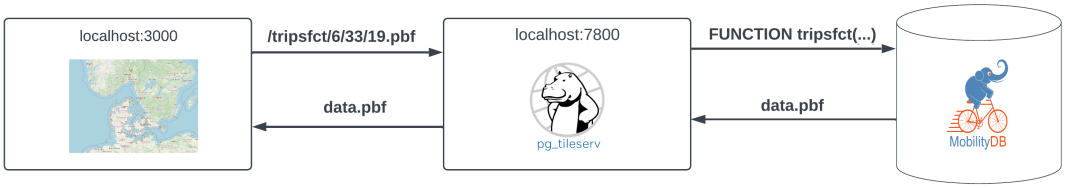


Figure 4.5: Interaction with tile server

Figure 4.5 describes the architecture of the application that allows the visualization of ships moving in the Danish seas in MVT format.

The React front end running on port 3000 calls the tile server running on port 7800 (default port of pg_tileserv) at the following URL: `localhost:7800/public.tripsfct/{z}/{x}/{y}.pbf`. This triggers a call to the database which returns the data in MVT format allowing it to be displayed.

Express

Another possible approach to managing the visualization of tiles in OpenLayers is to configure OpenLayers and Express so that they send requests containing the parameters `z`, `x` and `y`. This approach provides additional flexibility for customizing tile URLs and allows for integration with OpenLayers.

To implement this approach, the `tileUrlFunction` attribute of `ol/source/vectorTile` class from OpenLayers is used.[18] This attribute is used to define a custom function that generates tile URLs based on the parameters `z`, `x` and `y`. Listing 4.9 contains the code used for building the custom URL. When the OpenLayers map needs to load a tile, this function is automatically called to create the corresponding URL and send a request. The template of the URL looks like this: `localhost:3001/public.tripsfct/{z}/{x}/{y}`. The `{z}`, `{x}` and `{y}` parameters are dynamically replaced by the corresponding values when the URL is generated for each tile requested.

```
let vectorTileSource = new VectorTileSource({
  format: new MVT({
    featureClass: Feature,
```

[18] *https://openlayers.org/en/latest/apidoc/module-ol_source_VectorTile.html*

```
  }),
  tileGrid: createXYZ({ maxZoom: 18 }),
  tileUrlFunction: function (tileCoord) {
    const z = tileCoord[0];
    const x = tileCoord[1];
    const y = tileCoord[2];

    const url = `http://localhost:3001/tiles/${z}/${x}/${y}`;
    return url;
  },
});
```

Listing 4.9: VectorTileSource using tileUrlFunction

Then, with Express, it is possible to configure the server to respond to tile requests using the custom URL. This can be achieved by defining a corresponding route in Express and executing a specific function when a request is received. This function reads the `z`, `x` and `y` parameters of the request, accesses the database, executes the function that retrieves the tiles (described in Listing 4.8) and returns the appropriate tile to the application in response.

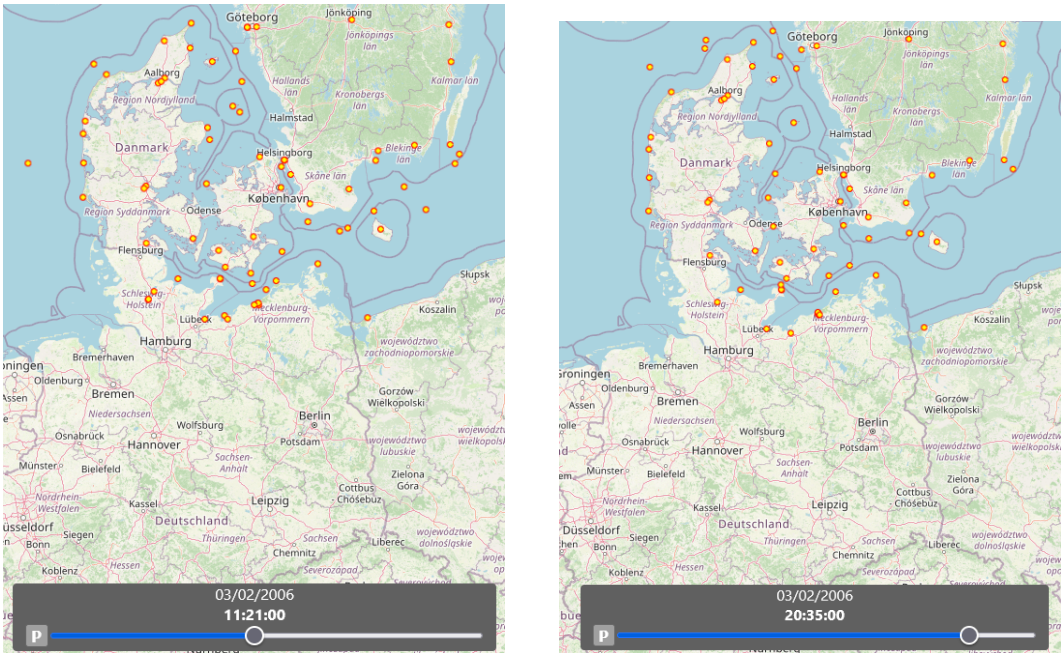
This approach offers an alternative solution to that used with pg_tileserv, while sharing a similar architecture. By using Express to manage tile requests, it is possible to further customize the tile loading process and integrate features specific to any application. The advantage of this approach is the flexibility it offers. By customizing tile URLs and using Express, you can tailor the tile loading process to the needs of your application. This allows you to control the flow of data between the server and OpenLayers, which can be particularly useful when handling large amounts of data or when using external data sources.

4.2.3 Well-Known Binary

The WKB solution was not implemented due to the lack of support in OpenLayers' WKB parser for reading temporal data added by MobilityDB as described in Section 2.2.4 The OpenLayers WKB parser is essential for converting WKB-encoded data into usable geometric objects in the code. However, it was not designed to handle temporal data enriched by MobilityDB. MobilityDB offers a powerful solution for representing spatiotemporal data but it requires proper support in OpenLayers' WKB parser. Without this support, the temporal data may be ignored or misinterpreted during processing with OpenLayers. To overcome this obstacle, the WKB parser would need some adjustments to ensure the integration with MobilityDB. It is important to understand that developing this support may be complex and time-consuming but it would unlock new possibilities for using MobilityDB in the context of OpenLayers.

4.3 Visualization

With the implementations described above, it is in fact possible to display data as it moves around the map. Figure 4.6 shows an overview of the data at 2 different points in time. To visually represent the data, the implementation uses markers on the map to indicate specific data points. Each marker is associated with its identifier (MMSI or for example trip_id), location and timestamp. These markers are positioned on the map according to their geographical coordinates. As time progresses, the data points can move, reflecting the dynamic nature of the information being observed. The algorithm continuously updates the positions and attributes of the markers on the map to reflect these changes. This ensures that the data displayed remains synchronized with the time set by the slider.



(a) Danish AIS data at 11h21

(b) Danish AIS data at 20h35

Figure 4.6: Danish AIS data at 2 separate instants

The time control slider is a powerful feature that allows users to interact with the data displayed on the map in a precise and flexible way. This slider offers several options for managing time and controlling the movement of markers. On the right is a horizontal bar representing the time range available for displaying data. To the left of the bar is a 'Stop' button (defined by the letter S), which pauses the movement of the markers. When this button is activated, the markers remain stationary and the user can examine the data at a specific point in time. When the 'Stop' button is pressed, it is transformed into a 'Play' button (defined by the letter P), which activates the continuous movement of the markers as time elapses. When this button is activated, the markers automatically move across the map,

reflecting the progression of time. The slider also allows users to manually drag the cursor along the horizontal bar to select a precise moment in time. As the user drags the cursor, the markers on the map move accordingly, displaying the data corresponding to the chosen instant. One of the key features of the slider is its display of the exact time. Just above the horizontal bar, real-time text displays the current date and time, with minutes and seconds. This allows users to see exactly when the markers are positioned.

Another feature has been integrated into the application, improving user experience. This was achieved by integrating the FPS component of the `react-fps-stats` library. This component brought an additional layer of interactivity and performance monitoring to the application, ensuring a smoother, more responsive experience for users. It enables real time monitoring of the number of FPS displayed during user interactions. This provides an accurate view of the fluidity of the user experience, and facilitates the rapid detection of any performance problems. It was also thanks to this functionality that it was possible to compute the number of FPS. Figure 4.7 displays an overview of the feature.

Figure 4.7: FPS component in the application

The integration of the component was relatively simple. First, the `react-fps-stats` library was installed with the `npm install` command. After that, the component was imported into the `App.js`. The component was then added to the existing component hierarchy, enabling real-time display of FPS information. Listing 4.10 describes the code in the `App.js` of how to import and use the component.

```
1 import FPSStats from "react-fps-stats";
2   ...
3   <FPSStats />
```

Listing 4.10: FPS import code

Brooklyn GTFS

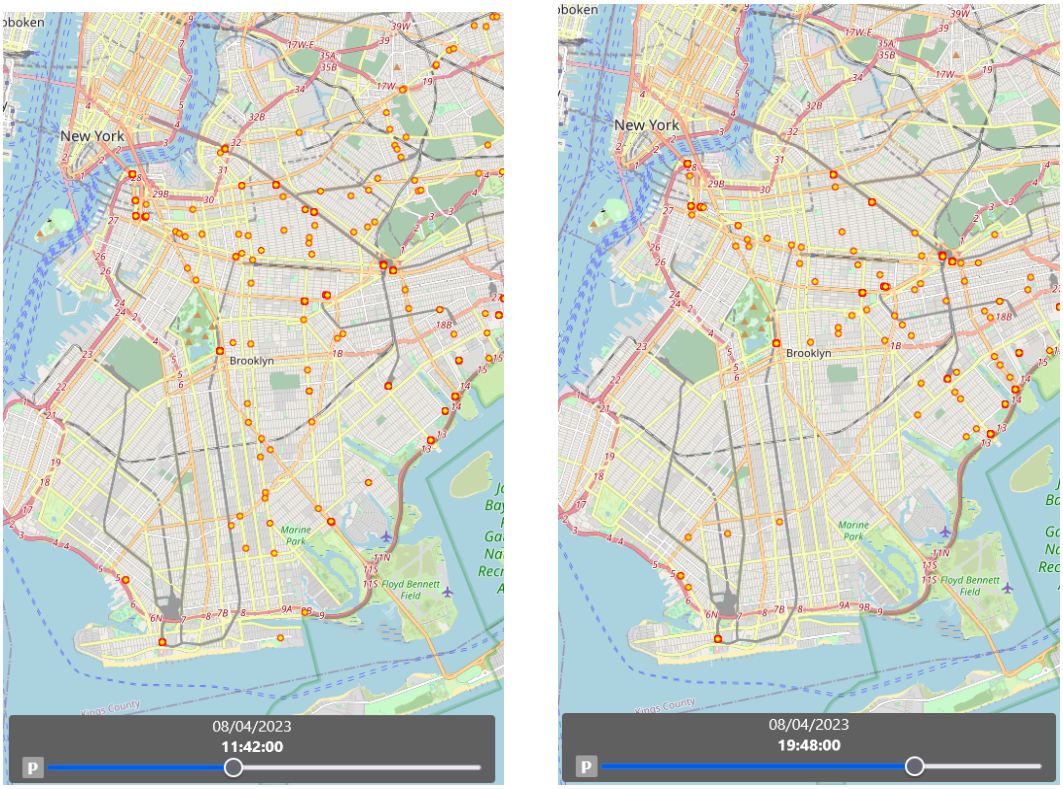
In order to verify that the implementation was working correctly, tests were carried out using another dataset separate from the Danish AIS. These tests were executed on bus data from Brooklyn, New York. The data comes from the General Transit Feed Specification (GTFS), a standardized format used by many public transport agencies to provide information on bus and rail schedules, routes and stops. The Metropolitan Transportation Authority (MTA), which is responsible for public transport in New York, distributes this GTFS data free of charge.[19]

[19] *http://web.mta.info/developers/developer-data-terms.html*

However, in order to display and analyze this data in the context of the study, it was processed using MobilityDB. The GTFS data supplied by the MTA was imported into a MobilityDB database to make it compatible with analysis and visualization on OpenLayers. This pre-processing step (described in the thesis of another student) makes it possible to take advantage of MobilityDB's advanced functionalities, such as distance calculation, trajectory detection and temporal queries, to obtain accurate information about bus routes in Brooklyn.[20] For this visualization, the implementations that display the data simply need a table in the database that contains the trip identifier and a column of type `tgeompoint`. The algorithm transforms this `tgeompoint` into GeoJSON or vector tiles and displays them according to time.

These tests validated the adaptability of the implementation to different types of spatiotemporal data such as AIS or GTFS data. Using data from buses in Brooklyn, it was possible to check that the implementation was capable of correctly processing and displaying information specific to buses, updating the positions of markers as a function of time. The success of these tests on GTFS data in New York demonstrates the flexibility and robustness of the implementation in the visualization of mobility data. It also highlights its ability to be adapted to different application areas, such as public transport management, urban planning and other sectors where the monitoring and visualization of spatiotemporal data is essential.

[20] *https://github.com/MobilityDB/MobilityDB-PublicTransport*



(a) Brooklyn data at 11h42

(b) Brooklyn data at 19h48

Figure 4.8: Brooklyn data at 2 separate instants

Chapter 5

Results

5.1 Outline

This chapter presents a series of findings and insights gained from the assessment of different aspects and technologies. It aims to provide a comprehensive understanding of the strengths and weaknesses of the solutions and implementations explored earlier.

The core content of this chapter revolves around a series of evaluations each focusing on different aspects. The study aims to assess the performance of data visualization using OpenLayers with React. It starts with a comparative study of the loading times of MVT data and GeoJSON data. For the MVT data visualization, two different architectures were used: one using Express and the other with pg_tileserv. Then, there is an assessment that compares the performance of a deck.gl solution with the implementation that uses OpenLayers. The third evaluation focuses on assessing the difference between a visualization of GTFS data and a visualization of AIS data. This helps to understand if the data source has an impact on visualization or not. Furthermore, an evalution on the framework has been executed. It compares a solution using React with another that do not use any framework (Vanilla JS). The final experiment is based on Frames Per Second (FPS). It highlights the limitation of OpenLayers' visualization of mobility data.

Before presenting the results, it is important to recall the resources used in this study. They are the same as those described in Table 3.1, with the addition of other resources described in Table 5.1

pg_tileserv version	1.09
Express version	4.18.2
Linux	Ubuntu for windows 22.04.2

Table 5.1: Additional ressources used for the performance test

The main dataset used for this benchmark is the GTFS data of the Brooklyn buses. The data was inserted into the MobilityDB database as described in Section 4.3 and was obtained on our OpenLayers application in MVT and GeoJSON formats. The number of data evaluated can go up to 10,000 trips. For some other evaluations,

Danish AIS was also used. It was also inserted into the MobilityDB database as described in Section 4.1.2

5.2 Remarks

In the following Section:

- **Loading time** correspond to the time from the instant where the web page starts loading to the instant where the ships start moving.
- **Average loading times** correspond to the average of the loading times on 10 different executions.
- In order to prevent any further instabilities or problems, the tests were stopped at 10,000 trips. This decision is designed to guarantee the reliability of the results obtained and to maintain the integrity of the performance of the solutions evaluated.

5.3 Results

This section focuses on presenting the results obtained during the study and are observed in milliseconds (ms). The data analyzed in this section is determined in terms of this unit of measurement, which allows a precise assessment of the times involved in different processes and actions. The millisecond metric is essential for assessing the speed and efficiency of the systems observed. The results will be interpreted in this context, identifying significant trends and conclusions for each measurement.

5.3.1 Evaluation of Loading Times

The analysis of loading times of mobility data on OpenLayers using different formats and architectures. The graph shows the average loading times on 10 executions of the solutions used with OpenLayers for different quantities of trips. The three solutions compared are GeoJSON, MVT with pg_tileserv and MVT with Express. The raw values of the 10 executions can be seen in Figure 5.1

The results of the graph show conclusively that the GeoJSON solution offers the fastest loading times of the three options. Regardless of the number of trips considered (1,000, 2,000, 3,000, 4,000 or 5,000), GeoJSON remains in the lead in terms of loading performance. This suggests that using GeoJSON files to render geospatial data is an effective and fast approach on OpenLayers.

With regard to MVT solutions, two variants were tested: MVT with pg_tileserv and MVT with Express. The results show that the two vector tile solutions have similar performance, with MVT using Express being slightly faster than MVT using pg_tileserv. Although the differences are small, this can be attributed to factors such as server configuration or the specific optimizations of each solution.

(a) Loading times for 1,000 trips

(b) Loading times for 2,000 trips

(c) Loading times for 3,000 trips

(d) Loading times for 4,000 trips

(e) Loading times for 5,000 trips

(f) Average loading times

Figure 5.1: Loading times for 1,000 to 5,000 trips in ms

However, it is important to note that both MVT options offer good performance overall and are good alternatives to GeoJSON.

When evaluating the performance of different solutions, it was found that the use of GeoJSON offers a better loading time compared to the vector tile approach. The main reason for this performance difference is the additional processing required by the vector tile solution. Vector tiles require a pre-processing step, which can be an expensive process in terms of time and resources. In contrast, the GeoJSON format allows geospatial data to be represented directly without the need for pre-processing. Thus, by avoiding this additional step, the GeoJSON solution has a faster execution.

The data used for the analysis was observed over different quantities of trips, including 100, 500, 1,000, 2,000, 3,000, 4,000, 5,000 and 10,000 trips. However, for the purposes of this test, only the data corresponding to 1,000, 2,000, 3,000,

4,000 and 5,000 trips were retained in order to study their evolution. This specific selection of trip quantities gives a better understanding of how loading times evolve as the number of trips increases. By focusing on these five values, we can observe a gradual progression and measure the general trend in loading times for different solutions. By choosing evenly spaced trip quantities, we get a more complete picture of the performance of the solutions tested. This allows us to determine whether loading times increase linearly or whether there are significant variations as the number of journeys increases.

The conclusions drawn from this analysis provide solid confirmation of the linear trend in loading times for all the solutions examined. This trend clearly indicates that as the number of trips increases, loading times rise proportionately with remarkable consistency. This clear linearity offers an insight into reliable and predictable performance, a major advantage for system scalability and wise resource planning.

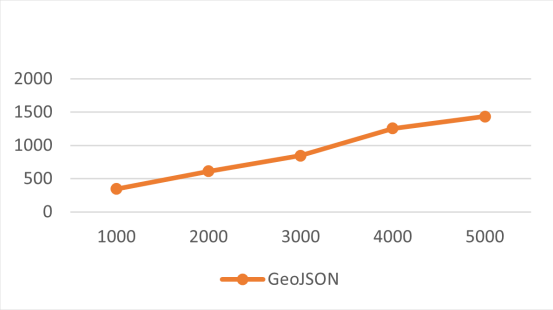


Figure 5.2: Average loading times of GeoJSON data

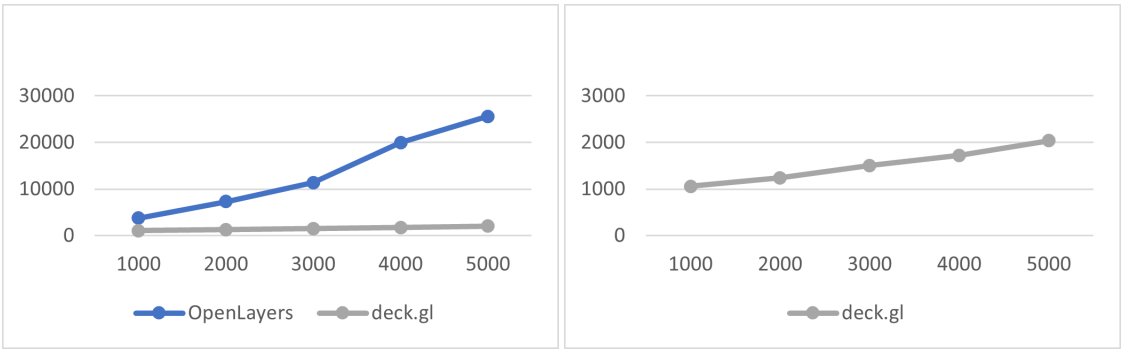
In addition, a close look at the Figure 5.2 highlights the use of GeoJSON data justified by its smaller scale compared to other formats. An intriguing observation emerges: despite this smaller scale, GeoJSON data also follows a linear trajectory in its loading time evolution, in parallel with other formats. This observation reinforces the idea that the linear trend remains constant, even when applied to GeoJSON-type data, underlining the system's ability to handle a variety of data types consistently, while maintaining its exemplary performance.

5.3.2 Evaluation of Deck.gl and OpenLayers

To analyze and compare the performance of our OpenLayers implementation, the deck.gl solution, based on the recommendations in Fabricio's thesis, was reproduced by adapting it to the data used in the experiments (GTFS data from New York buses and AIS data from Danish ships). Using deck.gl, it was possible to highlight the differences in performance, interactivity and visual quality between our OpenLayers implementation and and the one from Fabricio. These comparisons enabled us to identify the strengths and weaknesses of each solution.

Vector Tiles

Figure 5.3a shows the evolution of loading times according to the number of trips loaded. For small datasets, loading times were relatively similar between the two libraries, with only a small difference. However, as the number of trips increased, the difference in performance became increasingly significant. The graph highlights the trend for deck.gl to outperform OpenLayers in terms of loading times, particularly as the number of trips increases. It highlights deck.gl's advantage in handling large workloads and processing large amounts of MVT data efficiently. The graphical representation does not show a clear view of the evolution of loading times for deck.gl due to the difference in performance between the two libraries. For this reason, Figure 5.3b only shows the results for deck.gl

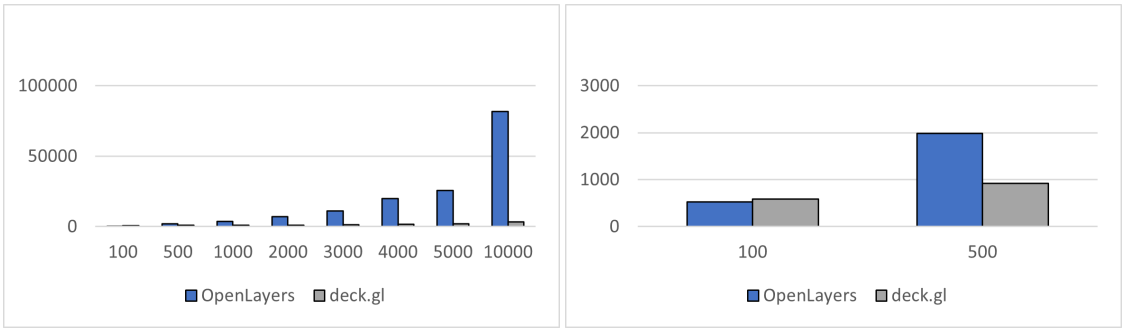


(a) OpenLayers vs. deck.gl *(b) deck.gl loading times*

Figure 5.3: Average loading times of MVT data in deck.gl and OpenLayers in ms

Figure 5.3b shows that for deck.gl there is a linear progression in loading times. As the number of trips loaded increases, loading times increase consistently, but at a moderate rate. This demonstrates deck.gl's ability to manage resources efficiently, maintaining consistent performance even as workloads increase. In the case of OpenLayers, the evolution of loading times is more pronounced. As the amount of MVT data increases, loading times increase more significantly. This indicates that OpenLayers may have difficulty managing large amounts of data with the same efficiency as deck.gl, resulting in faster growth in loading times. Figure 5.4a clearly shows the difference between the two solutions and the gap that widens as the number of data increases.

It is important to note that there is a correlation between the amount of MVT data loaded and the difference in performance between deck.gl and OpenLayers. When only a few trips were loaded, the difference in loading time between the two libraries was negligible. However, as the number of trips increased, the performance difference became more and more significant. This suggests that deck.gl is particularly effective at managing large amounts of MVT data. Its optimized approach and GPU-accelerated rendering allow it to maintain high performance even with large amounts of data. In contrast, OpenLayers took a relatively long time to load MVT data. Although OpenLayers offers a wide range of features for map-based applications, this seems to have an impact on performance in terms of data



(a) Loading times for 100 ro 10,000 trips *(b) Loading times for 100 and 500 trips*

Figure 5.4: Average loading times of MVT data in deck.gl and OpenLayers in ms

loading speed. Additional functionality, such as extended layer and data source management, can lead to latency when loading large amounts of MVT data.

GeoJSON

Fabricio's deck.gl implementation contains one solution that uses vector tiles and another that uses GeoJSON as the geospatial data format. However, Fabricio's deck.gl solution that uses GeoJSON does not take advantage of MobilityDB's `asMFJSON` function, which extracts data in MFJSON format directly from the database. The absence of this functionality in Fabricio's implementation limits the application's adaptability to use other data sources. As Fabricio's solution does not support MobilityDB's `asMFJSON`, it is difficult to make a direct and fair comparison with the OpenLayers solution in GeoJSON.

5.3.3 Evaluation of GTFS and AIS

In this section, two sources of data used to visualize trips will be compared: GTFS and AIS. We will see here how the loading times of these two types of data differ and how centralized management of GTFS data can influence its accuracy compared with AIS data, issued by each ship with its own equipment. GTFS is a static data format widely used to display public transport timetables. Developers can use it to provide information on public transport routes, timetables and stops. This is usually managed by a single entity, such as the Metropolitan Transportation Authority, which ensures regular updates and consistent data. On the other hand, AIS is an automatic tracking system used in the maritime sector. It allows ships to exchange information in real time on their position, speed, direction and other navigational details. Each boat is equipped with an AIS transponder, which means that the data is generated and transmitted by each ship independently. To compare the two datasets, the query in Listing 5.1 is executed:

```
1 SELECT MIN(numInstants(trip)), MAX(numInstants(trip)),
2   round(AVG(numInstants(trip)), 2) FROM Ships;
```

Listing 5.1: Returns the trips in MFJSON

The purpose of this SQL query is to obtain statistical information on the number of instants in each trip recorded.

1. `MIN(numInstants(trip))`: This allows to calculate the minimum number of instants in a trip. The `numInstants(trip)` function returns the number of instants in the specified trip.
2. `MAX(numInstants(trip))`: This allows to calculate the maximum number of instants in a trip.
3. `ROUND(AVG(numInstants(trip)), 2)`: This allows to calculate the average number of instants in the trips. The `AVG(numInstants(trip))` function calculates the average, and `round(..., 2)` rounds the result to two decimal places.

This query was executed on the Brooklyn GTFS data provided by the Metropolitan Transportation Authority and the Danish AIS data to obtain an overview of the distribution of trip times. This statistical analysis provides a better understanding of how journeys are distributed in terms of duration, highlighting the most common values as well as the extreme values. Table 5.2 contains the results for GTFS and AIS.

	MIN	**MAX**	**AVG**
GTFS	15	511	171.42
AIS	1	2367	284.00

Table 5.2: GTFS and AIS instant distribution

The results show that in the GTFS data, the length of the trips varies from 15 to 511 recorded instants, with an average value of around 171 instants. This means that the majority of trips have a fairly low number of instants, but there are also some fairly long trips, reaching up to 511 instants. As for the AIS data, the trips have a number of instants ranging from 1 to 2367 instants, with an average number of around 284 instants. Again, the majority of trips are relatively short, while some longer trips can reach up to 2367 recorded instants.

AIS data show greater variability in the length of trips, with some trips being shorter and others much longer, compared to GTFS data where trips generally have shorter average durations and fewer extreme variations. For example, in the AIS data we find boats that do not leave the port for a whole day, and therefore have only one instant recorded for 24 hours.

Despite the significant differences in the minimum, maximum and average trip lengths between the GTFS and AIS data, it is interesting to observe that the average loading times for 100, 500 and 1,000 trips do not differ significantly between the two datasets as seen in Figure 5.5. Minimum and maximum trip lengths can be strongly influenced by special cases which may be relatively rare in the data. For example, in the AIS dataset there may be a few exceptionally short or long trips, but these may represent only a small proportion of all recorded trips.

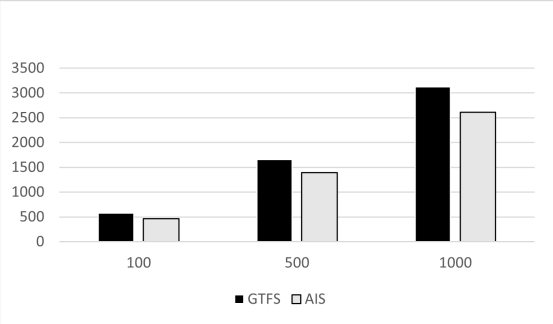


Figure 5.5: Average loading times of GTFS data and AIS data

Thus, although the minimum and maximum trip times can vary considerably between GTFS and AIS data, the average loading times for a given number of trips are close. This can be explained by the nature of the AIS data, which can contain both very short and very long trips, but when a large number of trips are taken into account, the effects of extreme cases are attenuated and do not have as much impact on the averages.

5.3.4 Evaluation of Vanilla JS and React

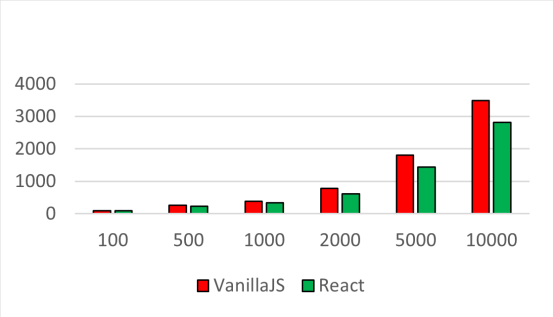


Figure 5.6: Average loading times of GeoJSON data in Vanilla JS and React

The graph shows average loading times for OpenLayers trips comparing an implementation using React to one without a framework (Vanilla JS). It's essential to note that the tests were carried out using data in GeoJSON format. The results of the graph show that although the React implementation is slightly faster than the Vanilla JS solution, this difference is insignificant in terms of trip loading times. The result is that, from the point of view of loading performance, the two approaches are practically similar.

However, it's worth pointing out that React's advantage does not lie primarily in its trip-loading performance, but rather in its modularity and other benefits. In fact, React allows you to break down your application into reusable components, which makes state management, user interface responsiveness and code maintenance much easier. These advantages translate into better code organization, increased scalability and greater ease of development.

Even if the difference in performance between the React and vanilla JS approaches is negligible, the choice to use React may be motivated by other considerations. These include the complexity of the project, the size of the development team, the need for long-term maintenance and the availability of a large community of developers. By opting for React, you can take advantage of the rich ecosystem of complementary tools and libraries, simplifying the development of complex applications.

In short, although the loading performance between React and the vanilla JS approach presents an insignificant difference, the choice between the two can be guided by factors other than raw performance. The modular advantages and development features offered by React can play an important role in larger, more complex projects requiring long-term maintenance. What's more, the ability to draw on a vast community of developers and a rich ecosystem of tools can greatly facilitate the application development and evolution process.

5.3.5 Evaluation of Frames Per Second

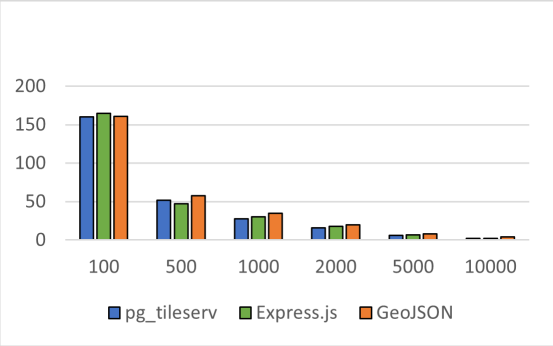


Figure 5.7: Average FPS

The results highlight an interesting perspective: whether it's the loading of GeoJSON data, the use of Mapbox Vector Tile (MVT) with pg_tileserv or the MVT with Express, these elements had no significant impact on frames per second (FPS) in our experiment. However, it should be noted that it is the number of trips displayed in OpenLayers that has a significant influence on the observed performance.

In our experiments, several datasets were used, starting with 100 trips and going up to 10,000 trips, as mentioned above. As we increased the number of trips, we noticed a decrease in the FPS. This can be explained by the fact that displaying a large number of trips places an increased load on the system. The results highlight the crucial importance of the number of trips displayed as a performance factor in the context of geospatial applications. Consequently, it is imperative to take this variable into consideration during the design and development phase of this type of application.

As previously mentioned, the tests were interrupted after attempting to display

10,000 trips, due to technical problems with the computer used. Unfortunately, the computer's performance was insufficient to handle the large-scale data processing and display load. As Figure 5.7 shows, at 10,000 trips the data was displayed at a mere 2 FPS.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The evolution of navigation technologies like the Global Positioning System (GPS) or the Automatic Identification System (AIS) contributed to the large production of spatiotemporal data that offers a lot of different uses cases for business, particulars and even governments. For example, air traffic control entities needs information about the current position and direction of a plane, car rental companies have to keep track of their rented cars or even someone who orders food has access to the live location of the delivery person.

In this work, the topic that has been studied is the **visualization of mobility data on OpenLayers**. There are several tools available to achieve this objective such as deck.gl, Mapbox GL JS, Leaflet or OpenLayers. Each tool has its advantages and disadvantages compared to the others and in order to have an understanding of the different tools, **a benchmark was carried out** with OpenStreetMap data obtained through the Geofabrik's download server with the objective to assess the loading times of the visualization of multiple datasets of various sizes : from 300 cities to 300,000 hamlets in different european countries.

The benchmark allowed us to observe that:

- OpenLayers stands out in its flexibility as it is handles maps from different sources and projections better than any other tool but it is not as popular as Leaflet or Mapbox.
- Leaflet's main advantage is simplicity. However this tool is not actively developed and it is the slowest in loading and displaying data.
- Mapbox is a pioneer in vector map technologies and has a big community. The main issue of Mapbox GL JS (the JavaScript library for visualizing maps of Mapbox) is that it is not open source anymore. However an open source version named 'MapLibre' exists. As it makes use of WebGL, MapLibre is highly performant.
- deck.gl is a framework that allows to visualize large datasets that also uses the WebGL technology and performs slightly better than MapLibre in general.

The first visualization on OpenLayers in this thesis was made on a dataset that comes from Fabricio's thesis who conducted similar experiments to those presented

in this work but on other tools.

Secondly, the same implementation was performed but this time on AIS data of ships navigating in the waters close to Denmark (the Skagerrak, the Kattegat and the Baltic sea). The difference with Fabrício's data is that the AIS data is provided in CSV format but after a conversion work with a MobilityDB database, the AIS data could be visualized. For this experiment, 3 different implementations were benchmarked:

1. Using the GeoJSON format
2. Using the MVT format (to encode vector tiles) with Express as technology to access the database
3. Using the MVT format with pg_tile as tile server

This study also explores other types of data to enrich the analysis including spatiotemporal data from OSM and Danish AIS along with integrating General Transit Feed Specification (GTFS) datasets for the Brooklyn, New York area. The GTFS data provides information on public transport schedules, stops and routes, adding an extra dimension to our assessment of mobility data visualization.

The integration of GTFS data from Brooklyn has enabled us to explore how OpenLayers can be used to display information about public transport. We were able to visualize trips, providing a more comprehensive view of mobility in the region. This expansion of our study gave us a better understanding of the capabilities and limitations of OpenLayers in visualizing different types of mobility data.

In addition to the findings presented in the thesis, there was an attempt to work with the Well-Known Binary (WKB) format alongside GeoJSON and vector tiles. WKB is a widely used standard for representing geometric objects such as points, lines and polygons in binary format. However, challenges arose as OpenLayers' parser does not support the temporal information added by MobilityDB.

In this study, the comparative analysis provided valuable information about the loading times and performance of different data formats and architectures for mobility data visualization. Experiments were carried out on several measurements, including the calculation of loading times for different volumes of data. We evaluated load times for 100, 500, 1,000, 2,000, 3,000, 4,000, 5,000 and 10,000 trips.

However, the focus was primarily on comparing loading times for scales of 1,000, 2,000, 3,000, 4,000 and 5,000 trips in order to observe the evolution on a logical scale. This experiment has shown that the GeoJSON solution offers faster loading times than MVT because of the necessary pre-processing of the vector tiles. Furthermore, the evolution of loading times remains linear whatever the solution.

Furthermore, a comparison was conducted between Fabricio's deck.gl solution and the OpenLayers implementation. Since Fabricio's solution does not make use of MobilityDB's `asMFJSON` function, a comparison between deck.gl's solution and

OpenLayers implementation using GeoJSON data was not feasible in the study. However, using vector tiles it was possible to compare them and the results revealed that for a small number of trips, no significant differences were observed but as the number of trips increased, deck.gl demonstrated greater efficiency in managing larger volumes of data compared to OpenLayers. This can be explained by deck.gl's optimized approach for and GPU-accelerated rendering.

Another interesting observation appeared when comparing the loading times of a solution displaying data coming from Brooklyn's GTFS with another one with the Danish AIS as data source. Despite variations in the minimum, maximum and average number of instants in a trip between GTFS and AIS data, the loading times showed similarity.

Moreover, a comparison of loading times was carried out between an implementation using React and one without (using Vanilla JS). This comparison allowed to measure the impact of using a framework on the performance of mobility data visualization. The results showed that the two solutions had similar loading times.

Finally, the study involved measuring the frames per second (FPS) of each solution to assess the fluidity of the animation and the responsiveness of the interface when displaying data. The different implementations do not affect FPS, it is the number of trips that has an effect on this metric. It is therefore important to take this variable into account during the design and development of such applications

In conclusion, our study highlights the importance of taking into account various factors, such as the volume of data, the use of frameworks and the choice of format, architecture and visualization library to ensure efficient spatiotemporal applications. This information is essential for developers and decision-makers in the field of data visualization, in order to create optimal user experiences in areas such as transport, urban planning and travel analysis.

6.2 Future Work

6.2.1 Visualization of WKB Data with OpenLayers

As part of this study, a shortcoming was identified in the integration of the Well-Known Binary solution with the temporally enriched data provided by MobilityDB, within the OpenLayers environment. The lack of adequate support in the OpenLayers WKB parser severely limits its usefulness for accurately processing and manipulating temporal data. However, identifying this limitation is not a dead end, but rather an opportunity for future research and development. With this in mind, further work could be undertaken to overcome this obstacle and open up exciting new perspectives for the joint use of MobilityDB and OpenLayers. One of the main ideas of research would be to design and implement adjustments to the existing WKB parser to allow seamless interaction with temporal data from MobilityDB. The task of integrating support for temporal data into the WKB parser

could be complex and demanding, requiring a deep understanding of temporal data models and the structure of MobilityDB. However, the potential benefits of this integration more than justify the effort required. Once achieved, this technical breakthrough would enable developers and data analysts to manipulate and exploit spatiotemporal data more accurately and consistently within OpenLayers.

6.2.2 Visualization with WebGL and OpenLayers

While OpenLayers may seem limited in its support for WebGL, a hardware accelerated 3D graphics technology, there are ways around these limitations to create dynamic and interactive mobile data visualizations. OpenLayers is widely recognized for its flexibility in creating interactive maps using different sources of geospatial data. However, its native support for WebGL has been criticized for its lack of functionality compared with other specialist libraries. Despite these initial impressions, the `WebGLPoints` class does exist. This class can serve as a solid foundation for mobile data visualization, even with OpenLayers' seemingly limited WebGL support. The `WebGLPoints` class can be used to display points on a map using WebGL. It offers basic functionality such as customising the size, colour and style of the points. This may be sufficient for many mobile data visualization scenarios. By inspecting the mobility data track with the `WebGLPoints` class, developers can leverage WebGL to make data points more expressive and interactive.

Bibliography

[1] Gennady Andrienko et al., "Space, time and visual analytics", in: *International journal of geographical information science* 24.10 (2010), pp. 1577–1600.

[2] Gennady Andrienko et al., "Visual analytics of mobility and transportation: State of the art and further research directions", in: *IEEE Transactions on Intelligent Transportation Systems* 18.8 (2017), pp. 2232–2249.

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

[4] Howard Butler et al., "The geojson format", in: *Internet Engineering Task Force (IETF)* (2016).

[5] Open Geospatial Consortium et al., "Simple Feature Acess, Part 1: Common Architecture", in: (2011), URL: *http://www.opengeospatial.org/standards/sfa*.

[6] Crunchy Data, *pg_tileserv*, URL: *https://access.crunchydata.com/documentation/pg_tileserv/latest/* (visited on 10/08/2023).

[7] Irene De Vreede, "Managing historic Automatic Identification System data by using a propre Database Management System structure", in: (2016).

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

[9] Julien Gaffuri, "Toward web mapping with vector data", in: *Proceedings 7, Geographic Information Science: 7th International Conference, GIScience 2012, Columbus, OH, USA, September 18-21, 2012.* Springer, 2012, pp. 87–101.

[10] George AM Gomes, Emanuele Santos, and Creto A Vidal, "Interactive visualization of traffic dynamics based on trajectory data", in: *2017 30th SIBGRAPI Conference on Graphics, Patterns and Images(SIBGRAPI)*, IEEE, 2017, pp. 111–118.

[11] Anita Graser, "Evaluating spatio-temporal data models for trajectories in PostGIS databases", in: *GI Forum Journal*, vol. 1, 2018, pp. 16–33.

[12] Anita Graser and Melitta Dragaschnig, "Open geospatial tools for movement data exploration", in: *KN-Journal of Cartography and Geographic Information* 70 (2020), pp. 3–10.

[13] Anita Graser, Esteban Zimányi, and Krishna Chaitanya Bommakanti, "Visualizing mobility of public transportation system", in: *arXiv preprint arXiv:2006.16900* (2020).

[14] Hideki Hayashi et al., "OGC Moving Features Access. Version 1.0.", in: (2017).

[15] Jing He et al., “Diverse visualization techniques and methods of moving-object-trajectory data: A review”, in: *ISPRS International Journal of Geo-Information* 8.2 (2019), p. 63.

[16] Yuqin He et al., “Application of OpenLayers in marine information monitoring”, in: *E3S Web of Conferences*, vol. 118, EDP Sciences, 2019, p. 03006.

[17] Kyoung-Sook Kim and Hirotaka Ogawa, “OGC Moving Features Encoding Extension-JSON. Version 1.0.”, in: (2017).

[18] *Leaflet - a JavaScript library for interactive maps*, URL: *https://leafletjs.com/* (visited on 10/08/2023).

[19] Mapbox, *Mapbox GL JS*, URL: *https://docs.mapbox.com/mapbox-gl-js/* (visited on 10/08/2023).

[20] MapLibre, *MapLibre*, URL: *https://maplibre.org/* (visited on 10/08/2023).

[21] Rostislav Netek et al., “Performance testing on vector vs raster map tiles-comparative study on load metrics”, in: *ISPRS International Journal of Geo-Information* 9.2 (2021), p. 101.

[22] OpenLayers, *OpenLayers - Welcome*, URL: *https://openlayers.org/* (visited on 10/08/2023).

[23] Tony Parisi, *WebGL: up and running*, " O’Reilly Media, Inc.", 2012.

[24] QGIS project, *Working with Vector Tiles*, URL: *https://docs.qgis.org/3.28/en/docs/user_manual/working_with_vector_tiles/vector_tiles_properties.html* (visited on 10/08/2023).

[25] A Pushkarev and O Yakubailik, “A web application for visualization, analysis, and processing of agricultural monitoring spatial-temporal data”, in: *Proceedings CEUR Workshop*, vol. 3006, 2021, pp. 231–237.

[26] Fabrıcio Ferreira da Silva, “Visual Analytics for Moving Objects Databases”, PhD thesis, tu-berlin, 2021.

[27] Imas Sukaesih Sitanggang et al., “Integration of spatial online analytical processing for agricultural commodities with OpenLayers”, in: *2017 International Conference on Electrical Engineering and Computer Science (ICECOS)*, IEEE, 2017, pp. 167–170.

[28] IBM StrongLoop and other expressjs.com contributors, *Express*, URL: *https://expressjs.com/* (visited on 10/08/2023).

[29] Ludéric Van Calck, “Visualizing MobilityDB Data using QGIS”, PhD thesis, Université Libre de Bruxelles, 2020.

[30] Vis.GL, *deck.gl*, URL: *https://deck.gl/* (visited on 10/08/2023).

[31] Shaohua Wang et al., “An Integrated Visual Analytics Framework for Spatiotemporal Data”, in: *Proceedings of the 1st ACM SIGSPATIAL Workshop on Advances on Resilient and Intelligent Cities*, 2018, pp. 41–45.

[32] Fang Yin and Min Feng, "A webGIS framework for vector geospatial data sharing based on open source projects", in: *Proceedings. The 2009 International Symposium on Web Information Systems and Applications (WISA 2009)*, Academy Publisher, 2009, p. 124.

[33] Wei Zeng et al., "Visualizing mobility of public transportation system", in: *IEEE transactions on visualization and computer graphics* 20.12 (2014), pp. 1833–1842.

[34] Esteban Zimányi and Mahmoud Sakr, *Towards a Well-Known Binary Format for Moving Features*, URL: *https://docs.mobilitydb.com/pub/WKB-OGC-2021-June-slides.pdf* (visited on 10/08/2023).

[35] Esteban Zimányi, Mahmoud Sakr, and Arthur Lesuisse, "MobilityDB: A mobility database based on PostgreSQL and PostGIS", in: *ACM Transactions on Database Systems (TODS)* 45.4 (2020), pp. 1–42.

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

Appendix

Benchmark_views.sql

```sql
CREATE OR REPLACE VIEW cities AS
  SELECT way, place, name
  FROM planet_osm_point
  WHERE planet_osm_point.place LIKE 'city';

CREATE OR REPLACE VIEW towns AS
  SELECT way, place, name
  FROM planet_osm_point
  WHERE planet_osm_point.place LIKE 'town';

CREATE OR REPLACE VIEW hamlets AS
  SELECT way, place, name
  FROM planet_osm_point
  WHERE planet_osm_point.place LIKE 'hamlet';

CREATE OR REPLACE VIEW villages AS
  SELECT way, place, name
  FROM planet_osm_point
  WHERE planet_osm_point.place LIKE 'village';

-- Do not forget to have all the permissions
GRANT ALL ON cities TO souf;
GRANT ALL ON towns TO souf;
GRANT ALL ON hamlets TO souf;
GRANT ALL ON villages TO souf;

-- Queries to obtain the number of points
SELECT COUNT(*) FROM cities;
SELECT COUNT(*) FROM towns;
SELECT COUNT(*) FROM villages;
SELECT COUNT(*) FROM hamlets;
```

Listing 6.1: benchmark_views.sql

Bench_commands.bash

The following commands have been executed on the belgian dataset:

```
ogr2ogr -f "GeoJSON" ./be_cities.geojson PG:"host=localhost user=souf
    password=souf dbname=MobilityDB" cities

ogr2ogr -f "GeoJSON" ./be_towns.geojson PG:"host=localhost user=souf
   password=souf dbname=MobilityDB" towns

```

```
ogr2ogr -f "GeoJSON" ./be_hamlets.geojson PG:"host=localhost user=
    souf password=souf dbname=MobilityDB" hamlets

ogr2ogr -f "GeoJSON" ./be_villages.geojson PG:"host=localhost user=
    souf password=souf dbname=MobilityDB" villages

osm2pgsql -c -H localhost -U souf -W -d MobilityDB -C 7000 belgium-
    latest.osm.bz2
```

Listing 6.2: bench_commands.sql

Ais.sql

```
CREATE OR REPLACE VIEW trips AS
  SELECT mmsi, array_agg(geom) as lines
  FROM aisinputfiltered
  WHERE latitude BETWEEN 53.76 and 60.46 AND longitude BETWEEN 5.35
    and 16.70
  GROUP BY mmsi;

CREATE OR REPLACE VIEW trips_conv AS
  SELECT mmsi, ST_Transform(ST_MakeLine(array_agg(lines)), 4326) as
    geom
  FROM trips
  GROUP BY mmsi;

GRANT ALL ON trips TO souf;
GRANT ALL ON trips_conv TO souf;
```

Listing 6.3: ais.sql

Ais.bash

The following commands have been executed on the belgian dataset:

```
ogr2ogr -f "GeoJSON" trips2.json PG:"host=localhost user=souf
    password=souf dbname=tmp" trips_conv
```

Listing 6.4: ais.bash

Mobilitydb.sql

```
CREATE TABLE AISInput(
  T timestamp,
  TypeOfMobile varchar(50),
  MMSI integer,
  Latitude float,
  Longitude float,
  navigationalStatus varchar(50),
  ROT float,
```

```
  SOG float,
  COG float,
  Heading integer,
  IMO varchar(50),
  Callsign varchar(50),
  Name varchar(100),
  ShipType varchar(50),
  CargoType varchar(100),
  Width float,
  Length float,
  TypeOfPositionFixingDevice varchar(50),
  Draught float,
  Destination varchar(50),
  ETA varchar(50),
  DataSourceType varchar(50),
  SizeA float,
  SizeB float,
  SizeC float,
  SizeD float,
  Geom geometry(Point, 4326)
);

COPY AISInput(T, TypeOfMobile, MMSI, Latitude, Longitude,
    NavigationalStatus, ROT, SOG, COG, Heading, IMO, CallSign, Name,
    ShipType, CargoType, Width, Length, TypeOfPositionFixingDevice,
    Draught, Destination, ETA, DataSourceType, SizeA, SizeB, SizeC,
    SizeD)
FROM '/home/mobilitydb/DanishAIS/aisdk_20180401.csv' DELIMITER  ','
    CSV HEADER;

UPDATE AISInput SET
  NavigationalStatus = CASE NavigationalStatus WHEN 'Unknown value'
    THEN NULL END,
  IMO = CASE IMO WHEN 'Unknown' THEN NULL END,
  ShipType = CASE ShipType WHEN 'Undefined' THEN NULL END,
  TypeOfPositionFixingDevice = CASE TypeOfPositionFixingDevice
  WHEN 'Undefined' THEN NULL END,
  Geom = ST_SetSRID( ST_MakePoint( Longitude, Latitude ), 4326);

CREATE TABLE AISInputFiltered AS
  SELECT DISTINCT ON(MMSI,T) *
  FROM AISInput
  WHERE Longitude BETWEEN -16.1 and 32.88 AND Latitude BETWEEN 40.18
    AND 84.17;
```

Listing 6.5: mobilitydb.sql

Ships.sql

```
CREATE TABLE Ships(MMSI, Trip, SOG, COG) AS
  SELECT MMSI,
    tgeompoint_seq(
      array_agg(
```

```
5         tgeompoint_inst(
6           ST_Transform(Geom, 25832), T)
7         ORDER BY T)
8     ),
9     tfloat_seq(
10      array_agg(
11        tfloat_inst(SOG, T) ORDER BY T)
12        FILTER (WHERE SOG IS NOT NULL)
13    ),
14    tfloat_seq(
15      array_agg(
16        tfloat_inst(COG, T) ORDER BY T)
17        FILTER (WHERE COG IS NOT NULL)
18    )
19  FROM AISInputFiltered
20  GROUP BY MMSI;
```

Listing 6.6: bench_commands.sql