Université Libre de Bruxelles

MEMO-F524 - Masters thesis

Mobility Data Exchange Standards in MobilityDB

Author:
Iliass El Achouchi
Affiliation:
ULB 000462349

Supervisor:
Esteban Zimányi

August 14, 2023

I declare on my honour that the following work is the product of a personal and individual project. All sources of information and literature used have been properly referenced.

Brussels, August 14, 2023

Acknowledgement

Above all, I would like to start by expressing my gratitude to my mother, whose dedication and unconditional love have been inexhaustible sources of motivation. Despite the difficulties we have faced throughout our lives, your determination and personal sacrifice have always fascinated me and continue to inspire me to this day. I thank you from the bottom of my heart for your unfailing support and your confidence in me.

I would also like to express my gratitude to my thesis supervisor, Esteban Zimányi, whose expertise, invaluable advice and encouragement guided me throughout this process. Your attentive guidance in pushing me towards excellence played a crucial role in the successful completion of this work. Your confidence in my abilities has been a constant source of motivation, and I greatly appreciate your valuable comments and suggestions.

I'd also like to thank my two best friends, Rami and Youssef, my lifelong accomplices, with whom I've shared so many unforgettable moments. Your unfailing support and presence by my side have been invaluable sources of comfort and motivation. From our giggles to our serious discussions, you've always been there to support and encourage me. Thank you for being exceptional friends. Our friendship will remain engraved in my heart, and I'm grateful to have been able to share this piece of life with you, hoping it lasts longer.

I'd also like to extend my warmest thanks to the coordinator of the youth center where I grew up, Ahmed. Your dedication to young people from popular areas has been a real catalyst in my educational journey. Thanks to your support, I have benefited from enriching learning opportunities and valuable resources that have contributed to my academic success. Your commitment to our community is inspiring and I am deeply grateful.

Finally, I'd like to thank everyone who has contributed in any way to my academic success. My classmates, Yass, JD, Mehdy, Marvin, Will, the people involved in my internship... Your encouragement, advices and confidence in my abilities have been essential to my success. I'm aware that my success and the person I've become are the fruit of a supportive ecosystem, and I'm extremely grateful to all of you. I'm proud of this milestone and I'm convinced that it will open up new perspectives for me. Thank you to all those who have contributed to my academic success, your impact on my life will remain forever engraved in my memory.

Abstract

Efficient management of urban mobility has become a major issue in modern cities. To meet this challenge, many cities and transit companies have adopted the General Transit Feed Specification (GTFS) as their standard for sharing public transportation data. However, as mobility continues to evolve, it is becoming increasingly important to integrate real-time data to enable dynamic management of transportation systems. In this thesis, we explore the various mobility standards in use today and propose a solution for integrating this data into the MobilityDB system, a database management system for mobile objects. MobilityDB enables geospatial data to be represented, stored and manipulated with temporal information, providing a suitable framework for managing mobility data. To integrate this data into MobilityDB, we examined various import technologies. We evaluated the tools and libraries available for importing public transport standards data such as GTFS Static, GTFS Realtime, NeTEx and SIRI taking into account criteria such as performance, ease of use and robustness. We selected the technologies best suited to our needs, making sure to take scalability and system scalability into account. Then we established a formula indicating from where we can transform data into MobilityDB type system. Our results show that choosing the right import standards and technologies is crucial to ensuring the smooth integration of public transport standards data into the MobilityDB system. This approach fully exploits MobilityDB's advantages in terms of temporal and geospatial data handling and provides a solid foundation for dynamic urban mobility management. This research provides valuable recommendations for transport agencies and urban mobility decision-makers, demonstrating the simplicity of importing public transport data, and some of the use cases that can be developed using MobilityDB.

Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

In our modern society, mobility is a major issue that requires innovative solutions. With increasing challenges such as inflation and CO_2 emissions, public transportation services and mobility institutions are faced with the need to grow and continuously improve their offerings. In this context, the use of information technology and data analysis to improve mobility is proving to be a promising avenue.

Spatial and temporal data play an essential role in understanding and optimizing mobility. It provides a detailed and accurate view of movements, trajectories and changes that occur across space and time. These data capture not only the geographic aspect of travel, but also its dynamic evolution. Thus, spatiotemporal data provides opportunities for in-depth analysis and informed decision-making in a variety of business domains.

In many industries, spatiotemporal data plays a key role in optimizing operations and improving services. For example, in healthcare, this data can be used to analyze patient movements, plan medical resources and improve accessibility to care. In urban planning, this data can be used to understand the travel patterns of residents, optimize transportation infrastructure and plan urban development in a more sustainable way. In addition, spatiotemporal data has applications in natural disaster management, environmental monitoring, logistics, fleet management, precision agriculture, and many other areas.

In the specific context of public transportation, the analysis of spatiotemporal data is of particular importance. This data helps to understand how public transportation systems work, identify potential problems, plan schedules and routes more efficiently, and optimize the user experience. With this in mind, public transportation agencies have begun to share their data so

that developers can integrate this information into their applications and provide more comprehensive and accurate mobility services. Google Maps, for example, based on these shared data, can provide an accurate route taking into account all available public transport by indicating the departure time of the bus from the station closest to the departure point.

Several data formats are used to share public transportation information, such as General Transit Feed Specification (GTFS), GTFS Realtime, Network Timetable Exchange (NeTEx), Service Interface for Real-time Information (SIRI), and many others. These formats enable the transmission of real-time data on schedules, vehicle positions, disruptions, and other relevant information. Sharing this data facilitates the creation of applications and services that provide accurate routing, real-time travel updates, and a better understanding of public transportation systems.

To facilitate the processing of this shared data, the use of a suitable Database Management System (DBMS) is essential. It is in this context that MobilityDB,[1] a moving object database (MOD), is positioned as a promising solution. MobilityDB offers the ability to store and query moving objects in an efficient manner, thus opening up new possibilities for analyzing and optimizing spatiotemporal public transportation data.

Therefore, this thesis aims to explore the import of spatiotemporal public transportation data, with a focus on using MobilityDB as a suitable DBMS, by developping a general method of import. Focusing on the analysis of different data formats (GTFS, GTFS Realtime, NeTEx, SIRI, etc.), existing standards, and integration methods, this research aims to demonstrate the suitability and compatibility of MobilityDB in the context of spatiotemporal public transportation data.

Finally, spatiotemporal data is of crucial importance in improving mobility, especially in the public transportation domain. The sharing of this data by public transportation companies helps to create better services and improve the user experience. The use of MobilityDB as an adapted DBMS opens new perspectives for the analysis and optimization of this data. This thesis proposes a suitable way to import public transportation data into MobilityDB. It also explores the importance of this data, highlighting the use of MobilityDB as a convenient data management tool.

[1] `https://mobilitydb.com/`

Chapter 2

State of the Art

2.1 Background and Definitions

This section will provide a detailed presentation about the different mobility standards, a panel of use cases and the importance of having these standards deployed, but also a detailed presentation about moving object databases and particularly MobilityDB.

2.1.1 Moving Object Databases and MobilityDB

A Moving Object Database is a database management system that allows to store and query objects that change with time [1]. MODs are concretely a kind of fusion of spatial and temporal databases. Basically, this extension is done by adding a temporal or spatial variable to a set of already existing variables.

Concretely, if a position of an object in a 2-dimensional plane is given by the pair (x, y). We can add the variable t which corresponds to the instant when the position is captured. Thus, the spatiotemporal position of our object is expressed by the tuple (x, y, t).

A moving object is thus a sequence of these positions. A concrete use case for this kind of database would be FlightAware,[1] a website that allows tracking the position of planes in the sky in real-time. By selecting a current or finished flight, the tool allows to visualise at any time the position of the plane on the map. We can therefore presume that the tool probably uses a MOD.

[1] `https://flightaware.com/`

The use cases are very diverse, going as we have already seen from vehicles tracking, as planes, ships, package delivery or even following the evolution of something much bigger, like a cyclone.

MobilityDB

MobilityDB [2] is an example of MOD. It is a PostgreSQL extension based on PostGIS, and adds functionality for managing objects in motion.

It should be noted that modelling, storing, and querying data about moving objects presents unique challenges and requires special techniques to effectively manage the spatiotemporal aspects of this data. As a result, MODs often have very different architectures and interfaces to traditional databases.

MobilityDB can handle a wide variety of moving objects, including points, lines and polygons that change position, shape or size over time. This enables MobilityDB to cover a wide range of use cases, from vehicle tracking and fleet management to weather analysis.

It offers a rich collection of functions for querying and manipulating spatiotemporal data, including basic functions such as obtaining the position of an object at a given time, as well as more advanced functions such as searching for objects that move together or determining the distance travelled by an object over a period of time.

MobilityDB continues to show its efficiency, for example by showing that it is efficient on distributed systems [3] or that it allows to analyse trajectories of moving objects [4, 5].

2.1.2 Mobility Standards Exchanges

As Mark Buchanan points out [6], the population living in urban areas is evolving and will continue to evolve. 20% of CO_2 emissions come from road traffic. Compacting people in urban areas is bound to increase this percentage over time. Buchanan concludes by saying that addressing these kinds of issues could lead to improvements in living conditions in the future.

Improving public transport networks could be beneficial. As Fadaei and Cats [7] have shown, by implementing several measures on public transports, Stockholm city can save more than 3 million Euros per year.

In their article [8], Antrim and Barbeau detailed many usages of public transit data. There are many applications that use geographic data that would benefit from having these data. The most notable example is Google

Maps. It contains all public transport stops on its basic map, and also allows you to plan your trips. Without the knowledge of the transport network, including its topology, its lines, its stops, and its schedule, this functionality would not exist. Another example given in the article, and one that is actually more significant, is related to accessibility. Sharing data allows some applications to simplify journeys for people with disabilities.

These improvements could have more impact if transport data sharing was effective. Indeed, [9] cites the benefits of having data shared. Among them, we have the reduction of energy consumption, and consequently the reduction of pollution. So we have a cause-and-effect situation which indicates that if the information is shared, the supply side represented by transport companies would improve its ability to meet the demand of passengers and society in general. This could improve deeper societal situations and dilemmas and it also explains and justifies the interest in sharing public transport data.

2.1.3 GTFS Static

General Transit Feed Specification [10] is a standard developed by Google and TriMet. It is today the most widely used and known standard. It provides a general and simple vision of a transport system.

GTFS consists of several text files, acting as comma-separated values (CSV) files. These files contain information about the transport service. The header of the file always contains the meaning of the different columns. To understand how this format is organised, the reference [11] is complete, but we will look at an example to be more concrete. We will analyse the structure of the public GTFS data of New York MTA Long Island Rail Road (LIRR). Data are obtained from TransitFeeds[2] which catalogues public GTFS data from many transportation services.

A GTFS folder consists of these different files:

- `agency.txt` provides informations about the company.
- `calendar_date.txt` specifies service dates as well as exceptions. These exceptions can be additions or deletions for specific dates and are represented by the value 1 or 2.

[2] `https://transitfeeds.com/p/mta`

- `routes.txt` presents the different lines that make up the transport service. (Line number, departure-terminus,etc.).
- `stops.txt` lists all the company's stops and also provides additional information such as the geographic coordinates of the stop. The location_type attribute is an integer value between 0 and 4 and represents whether it is a stop, a station, an entrance station, a generic node or a boarding area.
- `trips.txt` indicates the different routes on the service. This information is presented by indicating the `service_id` defined in `calendar` or `calendar_dates` and the `route_id` contained in routes.
- `stop_times.txt` indicates the schedule at the various stops.
- `shapes.txt` gives information to draw routes on a map.
- `feed_info.txt` gives information about the dataset and the publisher of the dataset.

It is important to note that the format here is not a generalization. In fact, the GTFS standard proposes a schema to follow, but this schema has many optional elements. There are many other files such as `fare_attributes.txt`, `calendar.txt`, `levels.txt`, etc.

The most striking example to give is that New York MTA has different formats for their services themselves. Between New York Long Island Rail Road[3] and New York MTA Bronx,[4] provided files are not the same. Both schemes have a common base of 7 files, 5 of which are mandatory. Among the optional files, we can see that the two schemes differ by one file. However, the facts are that this is not the only difference. The files also contain optional attributes, for example, Bronx MTA's `routes.txt` file contains a column named `route_desc` that is completely absent in the LIRR file.

In GTFS standard, files `agency, trips, routes, stops` and `stoptimes` are mandatory. The rest is optional or mandatory under certain conditions.

Use cases of GTFS are broad since it has been democratised and that people are using this data to do research. In their work [12], Goliszek and Polom analyse transport deviations during the morning rush hour. Piotr

[3] `https://transitfeeds.com/p/mta/86/20230419`

[4] `https://transitfeeds.com/p/mta/81/20230105`

G. et al. deployed gtfs2vec [12], a tool to interpret GTFS Static feeds as embeddings, allowing to cluster feeds by similarities and to then explore feeds with similar characteristics. Koragot Kaeoruean [13], established a study on the gaps between what a public transportation company offers and the demand of users. The evolution of machine learning in recent years has also set its sights on the field of mobility. Eva Chondrodima [14, 15] implemented machine-learning models to predict vehicles arrival times. Charvi Bannur [16, 17] also implements a traffic congestion prediction model. An interesting case is also explored in the reference [18], which couples GTFS data with smartphones GPS data of users. This allows to make an inference between both data sources and identify for example which route is mainly taken. All these studies are based on GTFS data.

2.1.4 GTFS Realtime

General Transit Feed Specification-realtime [19] is also a standard developed by Google. It is complementary to GTFS and provides real-time information from transport systems. It allows the notification of the position of vehicles or any changes on the lines, whether they be additions, modifications or even deletions of service. Like GTFS, it has been quickly adopted by many departments and researchers.

GTFS Realtime consists of a Protobuf[5] typed feed to be filled in according to the specification document. A GTFS Realtime feed is available and accessible via HTTP and is theoretically updated regularly.

Concretely, it is a sequence of entities that can represent 3 elements:

- `TripUpdate`: This entity represents the update of the progress of a trip in real-time, but can also represent the addition, deletion, or that a trip is proceeding normally based on attribute values.
- `VehiclePosition`: Provides information on the positions of a vehicle in real-time, as well as information on vehicle occupancy or the traffic congestion level in realtime.
- `Alert`: This is useful if an incident on a line occurs.

These messages follow a precise syntax, and some attributes request values from an enumeration.

[5] `https://developers.google.com/protocol-buffers`

Protobuf stream is a binary stream interpretable by the Protobuf library. The library allows to transform, interpret and process of real-time data provided by mobility agencies. For example, Long Island Rail Road real-time data provided by TransitFeeds[6] contains several messages of different types. The website has most likely used the Protobuf library to convert the binary stream into a human-readable text stream.

Nishino et al [20] demonstrated that we could improve the quality of alert information diffusion by using GTFS Realtime data. They assumed that in the case of a disaster, sirens could be attached to public transport in addition to fixed towers. By retrieving the positions of vehicles, we can optimise the coverage of the sound signal in residential areas.

Combined with GTFS Static data, studies on GTFS Realtime are numerous. We can measure and classify transit delays using GTFS-RT data [21]. Cortes [22] uses both static and real-time data to detect deviations in traffic networks, demonstrating the importance of combining the two feeds.

To return to the importance of sharing public transport data, the quality of the feeds is important. This is shown by Queiroz[23], which analyzes conformance and finds anomalies between data feeds and reality, which also encourages public transportation companies to focus on the quality of their feeds.

Language choice

To set up our server, for catching GTFS Realtime data, we must choose from several languages the one we will use to implement it. We need a language that can make HTTP requests, that can interact with a PostgreSQL database and that support the Protobuf library.[7] If the first two conditions are quite simple to fulfil, the last one is not. There are only a few languages that can handle Protobuf. Among this list, several high-level languages are available like Java, Python or Golang.

It is important to choose a language that allows easy and efficient use of the database but also a language optimized for the management of a lot of data. For example, we know that two of the most used languages for setting up web servers are Golang and Node.js. And we also know that in general, in the vast majority of the time, Go is more efficient in managing a server [24]. This study being conducted only on MySQL and MongoDB, we will try

[6] `https://transitfeeds.com/p/mta/421`

[7] `https://protobuf.dev`

to make a simple comparison between several languages using PostgreSQL, which will serve to give an indication of the choice of the language. As Node.js is not in the list of languages that support the Protobuf library, we are going to compare Go, Python, Java, high-level languages, but also C++, which is less high-level. These languages are generally very used in web development. Of course, the results here will depend on a simple experiment and will have to be studied in depth in another subject to find out which is the best language for using PostgreSQL and Protobuf.

Tests were established by using `psycopg2` as PostgreSQL driver on Python, package `pgJDBC` in Java, `pqxx` in C++ and `pq` as PostgreSQL driver on Go. Tests are performed by doing insertions into the database. The data are generated randomly according to the GTFS Realtime scheme of vehicle positions.

Number of inserts	Golang	C++	Java	Python
1 000	0.795	0.921	1.142	0.993
10 000	7.959	8.377	9.515	9.690
100 000	84.075	80.758	88.899	92.671
1 000 000	787.932	838.755	992.708	1488.589

Table 2.1: Insertion time in seconds with different languages

Table 2.1 shows the insertion times on different languages that support the Protobuf library. Globally in terms of execution speed, the Go language seems to be more efficient. This is not surprising since Go was designed to meet the needs of performance, speed and handling large amounts of data [25] when it comes to web servers. This makes Go an ideal candidate.

2.1.5 NeTEx

Network Timetable Exchange[8] (NeTEx) is a European standard defined by a set of XSD (XML Schema Document) files. Some institutions regularly ask transport companies to submit their service information in the European NeTEx format.

Globally, the NeTEx format is divided into 6 distinct parts:

- Part 1: Public Transport Network topology

[8] `https://netex-cen.eu/`

- Part 2: Scheduled Timetables
- Part 3: Fare information
- Part 4: European Passenger Information Profile – EPIP
- Part 5: Alternative modes exchange format
- Part 6: European Passenger Information Accessibility Profile – EPIAP

All parts are standards in themselves, part 6 is currently being defined. The specification files are available on the NeTEx website but also on Github.[9] These 6 parts actually represent definition schemes. They are not mandatory and each nation/group of nations actually defines its own implementation based on the parts it wishes to use. For example, Italy[10] has its own implementation based solely on the first three parts. Norway[11] also has its own implementation, which is extended to all the Nordic countries.

From a format point of view, NeTEx being an XML format, the use of validators allows to easily validate a file. But from a content point of view, NeTEx wants to be complementary to GTFS. Where GTFS mainly indicates the schedules of a service in order to simplify travel planning, NeTEx differs in that it is intended to have a broader scope by having back-office use cases to generate or refine data. NeTEx uses more information than GTFS, such as network description, complete network fare information, and a versioning system to facilitate exchanges. The fact that NeTEx contains much more information than GTFS makes conversion from GTFS to NeTEx simply impossible, but the opposite is completely possible.

Today, NeTEx data are not widely shared. It is quite hard to find such data on open data portals of public transport companies. But it remains true that some countries and cities still do it, like Norway.[12]

Awareness of open mobility data is not just with GTFS, indeed, Keller et al. [26] discuss the benefits of sharing mobility data. They implement a prototype application using NeTEx data to advise tourists on important points to visit, travel times, etc. The uses are varied, as for GTFS and GTFS Realtime. We have here a case [27] of analysis of ferry routes. The use of

[9] `https://github.com/NeTEx-CEN`

[10] `https://netex-cen.eu/implementation/italian-implementation/`

[11] `https://netex-cen.eu/implementation/norwegian-project/`

[12] `https://developer.entur.org/stops-and-timetable-data`

NeTEx data has allowed us in this study to analyze the sources of problems of delays or cancellations of ferries and to propose solutions.

2.1.6 SIRI

Service Information for Real-Time Information[13] (SIRI) is also a European Standard. It can be analogized to GTFS Realtime with GTFS Static, as it is the complementary of SIRI. As well as NeTEx, its specification is divided into many parts. Here are the 5 distinct parts.

- Part 1: Context and framework
- Part 2: Communications infrastructure
- Part 3: Functional service interfaces
- Part 4: Functional service interfaces: Facility Monitoring
- Part 5: Functional service interfaces - Situation Exchange

All these parts define the way that data should be exchanged. All the specification are as well available on Github.[14] SIRI allows users to share a lot of information, and is often combined with NeTEx to match the informations. It communicates information that complete in realtime a static feed. SIRI being a heavy protocol for some usages due to its 5 parts, SIRI Lite was also released to fill this kind of problems.

SIRI Lite offers sub-queries to obtain complete data artifacts. Here are some of the most frequently used examples.

- **ET – Estimated Timetable**: Contains estimations of planned timetables.
- **VM – Vehicle Monitoring**: Contains information about vehicles.
- **SX – Situation Exchange**: Communicate informations to inform to users.

[13] `https://www.siri-cen.eu`
[14] `https://github.com/SIRI-CEN`

The codespace specified (ET, VM, SX,...) simplifies the exchange during API calls. By specifying VM, for example, we have information about the vehicles, including their positions, the line concerned, its speed, and much more.

The literature on SIRI and NeTEx is currently very limited, due to the popularity of GTFS Static and GTFS Realtime, as well as the fact that NeTEx and SIRI are European standards, whereas GTFS is global.

2.1.7 Other Standards

Other standards exist, generally national standards, as proposed by France with the NEPTUNE standard. Or as the UK may propose with TransXChange. As these standards are regional or local standards, they will not be studied in this paper.

2.2 Related Work

The main focus of this thesis will therefore be to analyse these different standard formats and successfully import them into MobilityDB for data analysis.

2.2.1 Feeds Quality

As standards are generally adopted by the various actors in the mobility sector, the quantity of publicly available feeds is quite substantial. A quantity such that quality can sometimes be omitted.

Although specification documents exist, as well as articles indicating the habits to have when creating and editing GTFS [28] data in 2014. We can find shortcomings in the quality of feeds. For example, when we use a validator on a current GTFS Static feed, De Lijn in this case, we can see a very large number of warnings as well as errors. This is what the team at the Center for Urban Transportation Research [29] have tried to demonstrate. In their work, they are trying to establish a way of validating GTFS Realtime streams, as there is currently no official validator for this data format. Their experiment revealed errors on 54 of the 78 GTFS Realtime feeds, and warnings on 58 of them. This represents 69% of feeds containing errors, and 74% containing warnings. Among the most frequent errors are the non-existence of a `stop_id`

in the GTFS Static feed, two consecutive timestamps not being increasing, the impossibility of matching identifiers with the GTFS Static feed and many others. The most frequent warnings are missing information.

It is important to understand that quality control of a feed is essential to the formation of quality derived applications. Lack of validation can lead not only to inferior studies, but also to negative effects on consumers of multimodal trip applications. In fact, the vast majority of research using open data transport feeds generally goes through a pre-processing phase, which may involve format pre-processing, or simply pre-processing errors contained in the feed.

Today, there are many ways to validate public transport feeds. For the best-known GTFS Static and GTFS Realtime, tools developed by Mobility-Data[15] enable validation. For NeTEx and SIRI, Data4PT has developed its own data validation tool.[16]

However, there are also several tools for designing public data. Patrick Brosi publishes his reference [30] as a transit map generation toolchain. It contains his own tool, `pfaedle` [31] for building vehicle trajectories based on an OpenStreetMap file and an incomplete GTFS feed. It also contains several tools for extracting graphs from different formats, enabling you to extend your search further.

2.2.2 Importation and Visualisation

While the majority of articles use their own means of importing and visualising data from transit companies, there are tools that can import and visualise GTFS data, among others. Zhang Tianchi [32] developed a web application in 2014 based on the Google Maps API to visualise a GTFS feed, and some simple calculations based on road distances. In 2017, Kunama N. develops a tool called GTFS-Viz [33] for preprocessing and visualising GTFS data. This tool transforms a GTFS stream into a format that can be used by the tool to enable visualisation of public data.

While both of these tools are interesting and powerful, a visualisation does not in reality enable in-depth research into mobility issues or a transit company as described by Pertence A. [34] in his article. While these tools also provide some metrics on the general network of a feed, vulnerability analysis represents different metrics that are also unique to each type of

[15] `https://github.com/MobilityData/gtfs-validator`
[16] `https://greenlight.itxpt.eu`

region studied. Pertence, for example, establishes metrics for smart cities, cities that live with the times and use the evolution of technologies (IoT, Big Data, Governance, etc.) to benefit all their sectors, including mobility.

We can, however, analyze how these tools are imported. GTFS-Viz works in two stages: a preprocessor followed by a visualiser. The preprocessing phase consists in transforming a GTFS feed into a format that can be used by its visualiser. In fact, the geometries representing stops and lines are converted from CSV to GeoJSON. The latter format is widely used in the geographical field, and many map visualisation tools accept it. This allows the visualiser to display transformations and thus animations of routes.

Beyond that, there is no real standard for importing or displaying this type of data. Given the wide availability of data formats (XML, CSV, Protobuf, etc.), developers and researchers generally use the method they feel is best for them. They use the DBMS of their choice, as well as various visualisation tools (QGIS, Leaflet, Google Maps API, Mapbox, etc.). To date, there are no tools like GTFS-Viz for formats other than GTFS Static, such as NeTEx, SIRI or other less widely used formats.

2.3 Contribution

The aim of this master's thesis will therefore be to propose MobilityDB, a MOD for importing and processing data from public transport companies in general. Godfrid et al. [35] have already done so in the past, using MobilityDB to analyze traffic in Buenos Aires using GTFS data. The scope here will therefore be a little broader, proposing a methodology and general tools that will then enable us to theoretically import any data stream, regardless of its format. This will include a pre-processing phase enabling us first to import a data stream, regardless of its quality and quantity of information in the database. Next, we will identify the format from which a MobilityDB import is feasible, and try to extend it to several types of standard. The study will be carried out on several datasets from different sources, to identify good and bad habits. As MobilityDB is a PostGIS-based extension, we will of course be using the PostGIS geometry system to build our trajectories. The master's thesis will also propose interesting uses cases from a research point of view, but also from a public transport company point of view, to be able to analyze a feed. All the work and provided code is available on

Github on the official MobilityDB repository for public transport.[17]

[17] https://github.com/MobilityDB/MobilityDB-PublicTransport

Chapter 3

Import GTFS into MobilityDB

The goal here will be to successfully import the GTFS data into a MobilityDB environment. In other words, beyond simply importing the data into the database, to succeed in transforming them into a format that can be used by MobilityDB and to visualise the routes. All the code developed as part of this work is publicly available on Github.[1] Figure 3.1 illustrates the guideline of necessary steps to our objective.

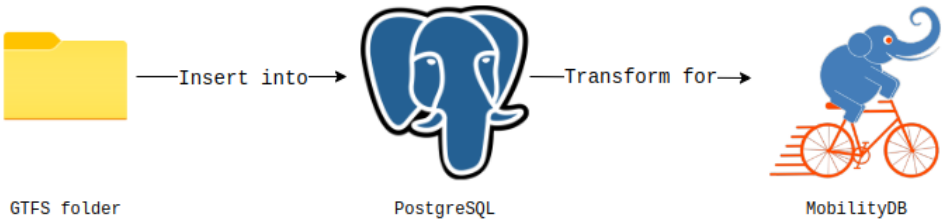


Figure 3.1: Guideline to import GTFS data into MobilityDB

3.1 GTFS to SQL

The advantage of GTFS is that it is a set of CSV files. This means that they are already interpretable as a table in some way. The GTFS reference schema contains 17 tables, all of which contain several attributes. Transportation

[1] https://github.com/MobilityDB/MobilityDB-PublicTransport

companies rarely use all of these tables and fields, which gives us much more simplified and uncluttered schemas, such as the Long Island Rail Road GTFS schema shown in Figure 3.2.

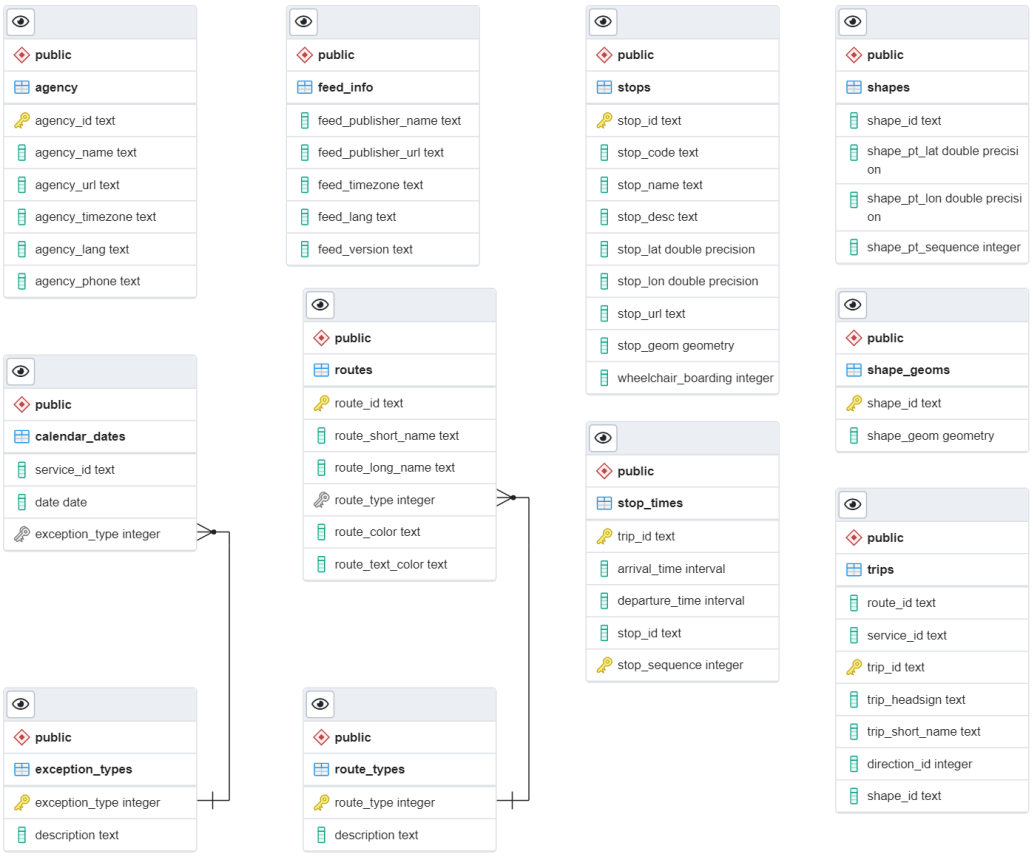


Figure 3.2: Long Island Rail Road GTFS Schema

Table `shape_geoms` does not come from the GTFS schema; it is an addition that will be used to store PostGIS geometries. Works allowing to import GTFS data in SQL already exist and are numerous.[2] However, the means used by these works generally use the association of several technologies like Bash for the treatment of files coupled with SQL for the importation in a

[2] `https://github.com/kausaltech/gtfs-sql-importer`

database. We will therefore see if it is possible to create a function in SQL allowing simple import from a PostgreSQL environment. The diagram displayed in Figure 3.3 represents the guideline for importing GTFS data into PostgreSQL.

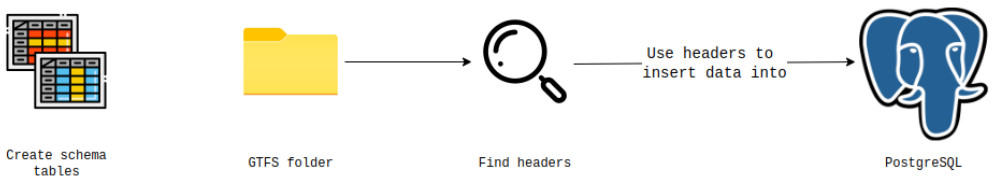


Figure 3.3: Guideline to import GTFS data into PostgreSQL

3.1.1 Header

Once the tables have been created, the files must be imported into the corresponding tables. We will use the `COPY` function in SQL to import the CSV files. The problem with this method is that GTFS is very flexible and allows us not to use all the tables and fields we have created.

For example, to import the `routes` table, we would have to use the command described by Listing 3.1:

```
COPY routes(
    route_id, agency_id, route_short_name, route_long_name,
    route_desc, route_type, route_url, route_color,
    route_text_color, route_sort_order, continous_pickup,
    continuous_drop_off)
FROM 'path/to/routes.txt' DELIMITER ',' CSV HEADER;
```

Listing 3.1: Static copy from CSV

This command would cause an error on the LIRR dataset. Indeed, not all the fields are used and it would not find a large part of the columns. Instead, the following command should be used:

```
COPY routes(
    route_id, route_short_name, route_long_name,
    route_type, route_color, route_text_color)
FROM 'path/to/routes.txt' DELIMITER ',' CSV HEADER;
```

Listing 3.2: Dynamic copy from CSV

In order to add this dynamism, we need to be able to detect the CSV header. Listing 3.3 shows a function in SQL that allows this.

```
DROP FUNCTION IF EXISTS find_header(filepath text);
CREATE OR REPLACE FUNCTION find_header(filepath text) RETURNS
    text AS $$
DECLARE
  content_text TEXT;
  first_pos INTEGER;
  head TEXT;
BEGIN
  SELECT pg_read_file(filepath) INTO content_text;
  SELECT position(E'\n' in content_text) INTO first_pos;
  SELECT left(content_text, first_pos) INTO head;
  RETURN head;

  END;
$$ LANGUAGE 'plpgsql';
```

Listing 3.3: Find header function

This function is quite simple, it opens the CSV file, locates the end of the first line using the line break and returns the contents of everything before the line break.

3.1.2 GTFS Main Files

With this function, we can now write a larger function that will read the headers of each file and use the `COPY` function to feed all the tables. That function is written in Listing 3.4.

```
FOREACH tablename IN ARRAY ARRAY['feed_info', 'stop_times', '
    trips', 'routes', 'calendar_dates', 'calendar', 'shapes',
    'stops', 'transfers', 'frequencies',  'attributions', '
    pathways', 'levels', 'fare_attributes', 'fare_rules', '
    agency'] LOOP
    BEGIN
      -- Find the file header
      SELECT find_header(format('%s%s.txt', fullpath,
      tablename)) INTO head;

      -- Add column if column specified in header does not
      exist
      FOREACH field IN ARRAY string_to_array(head, ',') LOOP
        EXECUTE format('ALTER TABLE %s ADD COLUMN IF NOT
        EXISTS %s TEXT;', tablename, field);
```

```
9         END LOOP;
10
11        -- Insert into table
12        EXECUTE format('COPY %s(%s) FROM ''%s%s.txt'' DELIMITER
          '','' CSV HEADER', tablename, head, fullpath, tablename
          );
13        EXCEPTION
14          WHEN SQLSTATE '42601' THEN
15          RAISE NOTICE 'file not found, ignoring';
16          WHEN SQLSTATE '58P01' THEN
17          RAISE NOTICE 'file not found, ignoring';
18      END;
19  END LOOP;
```

Listing 3.4: Main loop for CSV copy

We will loop over each possible file in the GTFS specification and apply the same treatment. Find the header and use the `COPY` function. In case the file on which we loop does not exist, this is not a problem and we can simply move to the next file.

However, a problem may arise. The flexibility of GTFS allows transport companies to add fields to the files. For example, STIB-MIVB, in its `translations.txt` file uses the header `trans_id,translation,lang`. None of these values is among the possible fields for the `translations.txt` file. To avoid any problems during the import, we will loop over each element of the header and add the column if it does not exist.

3.2 Transform to MobilityDB

All the data of the GTFS feed being now in the database, we can concentrate on what interests us: spatiotemporal data. The interest of coupling this data to a MOD such as MobilityDB is that we will be able to analyze the trajectories of the corresponding means of transport.

3.2.1 PostGIS Geometries

First of all, let us remind that MobilityDB is an extension of PostgreSQL and PostGIS. PostGIS being an extension allowing to treat geometric and geographical data, it does it with its own types which are geometries. We will therefore first transform the data from the GTFS stream that we are interested in into PostGIS geometries. As we are dealing with transit lines

mainly, we have stops and roads, that we can represent by geometries. Listing 3.5 shows the SQL code to compute these informations into PostGIS geometries.

```
-- Geometries setup
INSERT INTO shape_geoms
SELECT shape_id, ST_MakeLine(array_agg(
ST_SetSRID(ST_MakePoint(shape_pt_lon, shape_pt_lat), 4326)
    ORDER BY shape_pt_sequence))
FROM shapes
GROUP BY shape_id;

UPDATE stops
SET stop_geom = ST_SetSRID(ST_MakePoint(stop_lon, stop_lat),
    4326);
```

Listing 3.5: Compute routes and stops geometries

The first part of the script sets up the linear layout of the transport lines. `array_agg()` function coupled with `ST_MakeLine()` compute lines with the points of the routes ordered. While the second part of the script creates the geometry of the stops. As we are building a general way to import data into MobilityDB, Spatial Reference System Identifier (SRID) is set here at 4326, but the value may be changed.

By using a PostGIS visualisation tool like QGIS, we can see the points and routes of the transportation lines. Figures 3.4 and 3.5 illustrate the result on the Long Island Rail Road and New York City subway system GTFS streams respectively.

3.2.2 Service Dates

So far we have only used geographic data to get the current output, no temporal information has been used yet. As the GTFS data is on several tables, we will now have to find in all the tables the information necessary to create our routes.

We will therefore first use the tables `calendar` and/or `calendar_dates` to establish a view showing all the days of service for each service.

The `calendar` file describes the regular services that are available in the public transport system on a weekly basis. It contains information about the days on which services are active, i.e. the days of the week on which buses, trains or other modes of transport operate on a regular basis.

In the `calendar` file, we can find the following fields:

Figure 3.4: QGIS visualisation of New York Long Island Rail Road

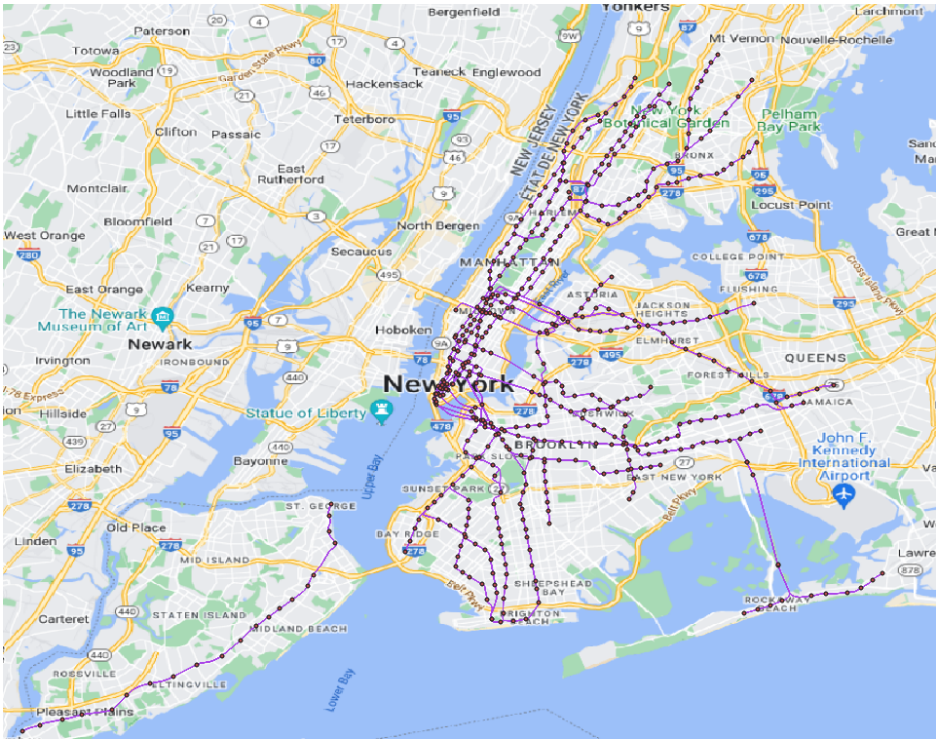


Figure 3.5: QGIS visualisation of New York Subway service

- `service_id`: a unique identifier for each service.

- `monday`, `tuesday`, `wednesday`, `thursday`, `friday`, `saturday`, `sunday`: Boolean indicators (0 or 1) to indicate whether the service is active for each day of the week.

Example: If a service operates from Monday to Friday, the fields `monday` to `friday` will have the value 1, while the fields `saturday` and `sunday` will have the value 0.

The `calendar_dates` file is used to manage exceptions to the regular schedules defined in the `calendar` file. It is used to specify special dates when services may be modified, such as public holidays, summer or winter schedules, or other exceptional events.

In the `calendar_dates` file, you can find the following fields:

- `service_id`: the identifier of the service to which the exception applies.
- `date`: the special date to which the exception applies.
- `exception_type`: a numeric flag to indicate the type of exception (adding or deleting a service). Typically, the value `"1"` is used to add a service on the special date, and `"2"` to remove the service on that date.

Example: If a service has a different schedule on Christmas Day, an entry will be added to the `calendar_dates` file with the service ID, the Christmas date and the `exception_type` indicating that the service will be modified or added on that date.

The `calendar` and `calendar_dates` files are two essential components of the GTFS Static data format used to provide timetable and journey information in public transport systems. However, it is important to note that their presence is not mandatory in all GTFS feeds. In some cases, only one or the other file may be included, depending on the complexity of the public transport system's timetables.

The `calendar` file describes the regular services available on a weekly basis, while the `calendar_dates` file is used to manage exceptions to the regular timetable. By using these two files together, transit system developers and managers can provide complete and accurate information about transit schedules, including days of regular activity and occasional exceptions, enabling users to plan their journeys efficiently. However, if a transport system does not have regular services or specific exceptions, one of these files may be omitted from the GTFS feed.

3.2.3 Trips Points

The following code is shown in the MobilityDB workshop, but it is well worth pointing out how it works. To design a MobilityDB trip, we theoretically need positions and their timestamps. At the moment, the only information we have available is the stops, the trajectory curves and the timetables that indicate when a means of transport passes a stop. With this information, we will have to build the trajectories from scratch.

Trip Observations

The first step is to build trajectories using stops. In the MobilityDB workshop, this step is performed to get a table called `trip_stops`. The name `trip_positions` will be used to be more general and applicable on more datasets, even if they do not have the notion of stops. We are going to create a table which, for each trajectory, will store the positions (here the stops) on the trajectory, and indicate the percentage of completion of the entire trajectory by stop. We will also store the estimated time of arrival at this stop.

```
CREATE TABLE trip_positions(
    trip_id text,
    stop_sequence integer,
    no_stops integer,
    route_id text,
    service_id text,
    shape_id text,
    stop_id text,
    arrival_time interval,
    perc float
);

INSERT INTO trip_positions (trip_id, stop_sequence, no_stops,
        route_id, service_id,
    shape_id, stop_id, arrival_time) (
SELECT t.trip_id, stop_sequence,
    MAX(stop_sequence) OVER (PARTITION BY t.trip_id),
    route_id, service_id, t.shape_id, st.stop_id,
    arrival_time
FROM trips t JOIN stop_times st ON t.trip_id = st.trip_id
);

UPDATE trip_positions t
```

```
SET perc = CASE
    WHEN stop_sequence =  1 then 0::float
    WHEN stop_sequence =  no_stops then 1.0::float
    ELSE ST_LineLocatePoint(shape_geom, stop_geom)
END
FROM shape_geoms g, stops s
WHERE t.shape_id = g.shape_id
AND t.stop_id = s.stop_id;
```

Listing 3.6: Query for computing trips positions

We use the tables `trips` and `stop_times` to obtain the first important informations. The query uses a subquery to obtain the value `no_stops` which corresponds to the total number of stops for each journey using the windowed function `MAX(stop_sequence) OVER (PARTITION BY t.trip_id)`. This retrieves the total number of stops for each journey and adds it to each row of the trip_positions table.

An update is then performed to calculate the relative position of the paths according to the stops. Linear interpolation involves estimating intermediate values between known points using a linear approach. In this case, the code calculates the percentage `perc` of the location of each stop along the route, using linear interpolation between the first stop (`stop_sequence = 1`) and the last stop (`stop_sequence = no_stops`) in relation to the geometric shape of the route.

More specifically, the `CASE` instruction in the code deals with three different cases:

- When the stop is the first in the route (`stop_sequence = 1`), the location percentage `perc` is set to 0, which means that the stop is located at the beginning of the trip.
- When the stop is the last in the trip (`stop_sequence = no_stops`), the location percentage `perc` is set to 1.0, which means that the stop is located at the end of the journey.
- For stops which are neither the first nor the last, the percentage location is calculated using the function :

  `ST_LineLocatePoint(shape_geom, stop_geom)`

  This function calculates the relative position of the stop along the geometric shape of the route using linear interpolation.

Trips Segments

The second step is to create all the geometric segments between two stops.

```
CREATE TABLE trip_segs (
    trip_id text,
    route_id text,
    service_id text,
    stop1_sequence integer,
    stop2_sequence integer,
    no_stops integer,
    shape_id text,
    stop1_arrival_time interval,
    stop2_arrival_time interval,
    perc1 float,
    perc2 float,
    seg_geom geometry,
    seg_length float,
    no_points integer,
    PRIMARY KEY (trip_id, stop1_sequence)
);

INSERT INTO trip_segs (trip_id, route_id, service_id,
   stop1_sequence, stop2_sequence, no_stops, shape_id,
   stop1_arrival_time, stop2_arrival_time, perc1, perc2)
WITH temp AS (
SELECT t.trip_id, t.route_id, t.service_id, t.stop_sequence,
    LEAD(stop_sequence) OVER w AS stop_sequence2,
    MAX(stop_sequence) OVER (PARTITION BY trip_id),
    t.shape_id, t.arrival_time, LEAD(arrival_time) OVER w,
    t.perc, LEAD(perc) OVER w
FROM trip_positions t WINDOW w AS (PARTITION BY trip_id ORDER
    BY stop_sequence)
)
SELECT * FROM temp WHERE stop_sequence2 IS NOT null;
```

Listing 3.7: Query for computing trips segments

The first step is to store all the consecutive stops. The `LEAD` function is used to retrieve the next stop and its `arrival_time` easily. The use of a CTE is also used to filter the result. The condition here is that `stop_sequence2` is not NULL, indicating that we are looking for all segments that have a next stop, thus excluding terminal stops.

```
UPDATE trip_segs t
SET seg_geom =
```

```
   (CASE WHEN perc1 > perc2 THEN seg_geom
    ELSE ST_LineSubstring(shape_geom, perc1, perc2)
    END)
FROM shape_geoms g
WHERE t.shape_id = g.shape_id;


UPDATE trip_segs t
SET seg_length = ST_Length(seg_geom), no_points =
   ST_NumPoints(seg_geom);
```

Listing 3.8: Update trips segments

We can now extract the sub-geometry corresponding to the observed segments using the function `ST_LineSubstring(shape_geom, perc1, perc2`. This extracts a geometry according to 2 percentages of the overall geometry. We also store the size of each sub-segment for later calculation.

Trips Points

As a reminder, a trajectory is a set of space-time coordinates. At this stage, we have space-time segments. So we are going to transform these segments into a consecutive series of points in the table `trip_points`.

```
DROP TABLE IF EXISTS trip_points;
CREATE TABLE trip_points (
    trip_id text,
    route_id text,
    service_id text,
    stop1_sequence integer,
    point_sequence integer,
    point_geom geometry,
    point_arrival_time interval,
    PRIMARY KEY (trip_id, stop1_sequence, point_sequence)
);


INSERT INTO trip_points (trip_id, route_id, service_id,
    stop1_sequence, point_sequence, point_geom,
    point_arrival_time)
WITH temp1 AS (
    SELECT trip_id, route_id, service_id, stop1_sequence,
        stop2_sequence, no_stops, stop1_arrival_time,
        stop2_arrival_time, seg_length, (dp).path[1]
        AS point_sequence, no_points, (dp).geom as point_geom
```

```
22      FROM trip_segs, ST_DumpPoints(seg_geom) AS dp
23  ),
24  temp2 AS (
25      SELECT trip_id, route_id, service_id, stop1_sequence,
26          stop1_arrival_time, stop2_arrival_time, seg_length,
27          point_sequence, no_points, point_geom
28      FROM temp1
29      WHERE (point_sequence <> no_points OR
30      stop2_sequence = no_stops) AND temp1.seg_length <> 0
31  ),
32  temp3 AS (
33      SELECT trip_id, route_id, service_id, stop1_sequence,
34          stop1_arrival_time, stop2_arrival_time,
35          point_sequence, no_points, point_geom,
36          ST_Length(ST_Makeline(array_agg(point_geom) OVER w))
37          / seg_length AS perc
38      FROM temp2 WINDOW w AS (PARTITION BY trip_id, service_id,
        stop1_sequence
39      ORDER BY point_sequence)
40  )
41  SELECT trip_id, route_id, service_id, stop1_sequence,
42      point_sequence, point_geom,
43      CASE
44      WHEN point_sequence = 1 then stop1_arrival_time
45      WHEN point_sequence = no_points then stop2_arrival_time
46      ELSE stop1_arrival_time + ((stop2_arrival_time -
47          stop1_arrival_time) * perc)
48      END AS point_arrival_time
49  FROM temp3;
```

Listing 3.9: Query for computing trips points

The idea is to calculate these points using a succession of three important operations.

- We first retrieve the information from the table `trip_segs` and use the function `ST_DumpPoints(seg_geom)` to extract the points along the geometry of each trip segment.
- We filter the `temp1` data to include only those points which do not correspond to the last point of the segment (`point_sequence <> no_points`) or which are associated with a path segment whose length is non-zero (`temp1.seg_length <> 0`).
- Finally, we calculate the percentage `perc` of the position of each point along the path segment using the function described by Listing 3.10 :

```
ST_Length(ST_Makeline(array_agg(point_geom) OVER w)) /
    seg_length
```

Listing 3.10: Calculate percentage of positions

It uses the windowed function `array_agg(point_geom) OVER w` to group together all the points on the same path, then calculates the total length of the line formed by these points (`ST_Length(ST_Makeline(...))`). Finally, the percentage is obtained by dividing this length by the total length of the path segment seg_length.

Once these 3 steps have been completed, we can fill in our `trip_points` table, as we have obtained the points for each trip. The arrival time at each stop is calculated using the `CASE` clause. The operation is very similar to that used to construct the `trip_positions` table, but on a smaller scale.

Trips Input

We finally reach the final stages of our transformation. For each trip we now have a series of spatiotemporal points describing the public transport trajectory. The final step is to match these with the service days calculated earlier using the `calendar` and `calendar_dates` tables. Listing 3.11 shows the code required for this operation.

```
CREATE TABLE trips_input (
    trip_id text,
    route_id text,
    service_id text,
    date date,
    point_geom geometry,
    t timestamptz
);


INSERT INTO trips_input
SELECT trip_id, route_id, t.service_id,
    date, point_geom, date + point_arrival_time AS t
FROM trip_points t
JOIN service_dates s ON t.service_id = s.service_id;

CREATE INDEX idx_trips_input
    ON trips_input (trip_id, route_id, t);
```

Listing 3.11: Query for computing trips inputs

By performing a join between the table `service_dates` and `trip_points` we then have a table full of spatiotemporal data. The format of this table is in line with the classic way of storing spatiotemporal information, i.e. one line per position. It is also the format required by MobilityDB to convert to its spatiotemporal trajectory storage format.

3.2.4 Duplicate Timestamps

It is possible that, due to the GTFS feed definition, two points for the same trip are inadvertently obtained at the same timestamp. In the MobilityDB `tgeompoint` definition itself, a point cannot be in two places at the same time. In general, this problem rarely occurs, and if it does, only a few points generate this error. To remedy this problem, a simple duplicate deletion algorithm with point retention is performed. Listing 3.12 shows how this is done. This query deletes by counting the number of occurrences of the tuple (`trip_id`, `route_id`, `t`). In the case of duplicates, the tuple with the lowest `ctid` is kept.

```
DELETE FROM trips_input WHERE
(trip_id, route_id, t) IN (
    SELECT trip_id, route_id, t
    FROM trips_input
    GROUP BY trip_id, route_id, t
    HAVING COUNT(*) > 1
)
AND ctid NOT IN (
    SELECT MIN(ctid)
    FROM trips_input
    GROUP BY trip_id, route_id, t
    HAVING COUNT(*) > 1
);
```

Listing 3.12: Delete points with duplicate timestamps

3.2.5 MobilityDB Trips

All this allows us for each trip identified by its `trip_id` to have the geometry of all the trip points, from the moment of arrival on the point. So we have all our spatiotemporal data.

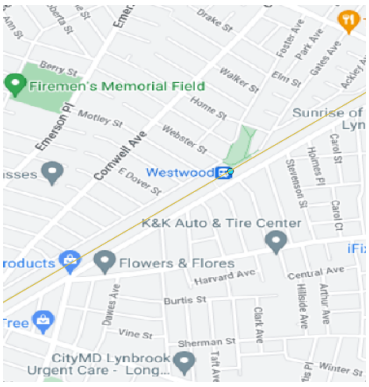
We will then finally create our last table `trips_mdb`, which contains the trips in the column `trip`, aggregation of the spatiotemporal data. The type

of the trip is a `tgeompoint`, i.e. a temporal geometrical point, a position that evolves in time. Listing 3.13 show how to compute the aggregation and insert the trips into the final table.

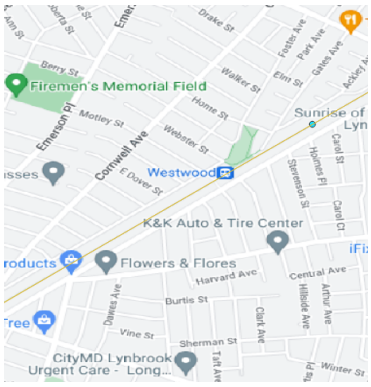
```
INSERT INTO trips_mdb(trip_id, route_id, date, trip)
SELECT trip_id, route_id, date,
    tgeompoint_seq(array_agg(tgeompoint_inst(point_geom, t)
ORDER BY T))
FROM trips_input
GROUP BY trip_id, route_id, date;


ALTER TABLE trips_mdb ADD COLUMN traj geometry;
UPDATE trips_mdb SET traj = trajectory(trip);


ALTER TABLE trips_mdb ADD COLUMN starttime timestamp;
UPDATE trips_mdb SET starttime = startTimestamp(trip);
```

Listing 3.13: Create and update MobilityDB trips

By using a visualisation tool like MOVE[3] for QGIS, we can then visualise the coordinates of our means of transportation moving in time. Figure 3.6 shows the theoretical position of a train on West Hempstead Branch on April 7, 2023, at 04:55:00 and 04:55:30.



(a) Theoretical position of LIRR train on April 7, 2023, at 04:55:00



(b) Theoretical position of LIRR train on April 7, 2023, at 04:55:30

Figure 3.6: Theoretical position of LIRR train

[3] https://github.com/mschoema/move

This approach shows us that it is possible to import trips from public transport data as long as we have certain precise information: the positions of stops, the stops timetable, as well as a geometry representing the trajectory taken by a vehicle. This information will lead us to create the table `trip_positions`, helping to transform for MobilityDB.

3.3 Preprocessing GTFS Data

It is essential to note that the feasibility of processing GTFS data is highly dependent on the quality of the data available. Although the GTFS format is designed to facilitate the exchange of information on public transport systems, there are cases where pre-processing is necessary due to formatting problems, typing errors or missing data.

3.3.1 File Formating Preprocessing

One of the most common problems encountered is related to typing errors in GTFS files. For example, fields that would normally contain integers may be incorrectly formatted with double quotes (`""`), introducing a typing inconsistency. This can complicate automated data processing and require pre-processing to correct these formatting errors.

Another common example is the presence of blank characters written as a space (`" "`). Although this may seem insignificant, it can lead to ambiguity as to the real meaning of the field. Is it really an empty character string or a string containing just a space? This ambiguity can lead to misinterpretation when analysing the data, so it is crucial to detect and treat these empty characters appropriately.

Another example specific to Madrid data, which highlights the importance of having quality data for processing GTFS data. In GTFS files from Madrid, an invisible character has been observed at the beginning of the file. This invisible character may be the result of encoding or formatting problems during the creation or the generation of these GTFS files.

This invisible character at the beginning of the file can lead to errors when data is imported into processing systems. These errors can compromise the integrity of the data and make subsequent analysis difficult or impossible. Consequently, it is essential to detect and remove this invisible character

when pre-processing GTFS data in order to guarantee the validity and consistency of the imported data.

It is essential to highlight the importance of close collaboration with data providers and increased vigilance when handling GTFS files to resolve formatting and typing errors. Quality control procedures should be established to identify and resolve potential problems with invisible characters or similar errors when importing GTFS data.

These formatting problems and typing errors are not inherent in the GTFS format itself, but rather in the process of creating and maintaining GTFS files by data providers. Close collaboration with these suppliers is therefore essential to ensure the quality of GTFS files before they are used in data processing.

It should be noted that these specific cases are not necessarily considered errors or warnings by the GTFS validator.[4] However, they can lead to errors when importing GTFS data into systems such as SQL. Therefore, it is crucial to take them into account in order to avoid problems later on when processing GTFS data.

With this in mind, we have developed a simple script in Python that will rewrite files containing this type of error. The reason for choosing to write this script in a language other than SQL is that we want to be able to process files more simply than in SQL. Given that the only task here is to rewrite with a few subtleties, Python seemed an appropriate candidate.

The first step is to write a dictionary containing the various files in the GTFS reference and store the types of their attributes. Storing all the integer types will enable us to filter later on the columns that do or don't require quotes in their column. Once this dictionary has been created, we can parse the files and clean up all the fields at the same time. Blank spaces indicating an empty field will be replaced by the absence of a character, and formatting errors that falsify the header as with the Madrid metro will be removed. Unnecessary spaces at the start and end of columns are also removed.

The function takes three parameters, `input_directory` which indicates the extracted GTFS folder, `output_directory` which indicates the destination of the processed files, and finally `expected_types`, which is the dictionary of expected types for each column of each table in the GTFS format.

```
1 def remove_anomaly_characters(in_dir, out_dir, exp_types):
2     # Create output directory if it doesn't exist
```

[4] `https://gtfs-validator.mobilitydata.org/`

```
    if not os.path.exists(out_dir):
        os.makedirs(out_dir)

    # Loop through each file in the input directory
    for filename in os.listdir(in_dir):
        # Check if the file has a .txt extension (considering
        # it as CSV)
        if filename.endswith(".txt"):
            # Path to the input CSV file
            csv_file = os.path.join(in_dir, filename)
            # Path to the output modified CSV file
            new_csv_file = os.path.join(out_dir, filename)
            with open(csv_file, 'r') as file:
                csv_reader = csv.reader(file)
                # Read the header from the CSV file
                header = next(csv_reader)

                # Remove non-alphabetic characters from the
                # beginning of each column name in the header
                for i in range(len(header)):
                    if not header[i][0].isalpha():
                        header[i] = header[i][1:]

                anomalies = []
                # Get expected types for the current file
                file_exp_types = exp_types.get(filename, {})
                for i, column_name in enumerate(header):
                    if column_name in file_exp_types and \
                        file_exp_types[column_name] in [int,\
                        float]:
                        # Detect anomalies in columns with
                        #numeric expected type that have
                        # quotation marks
                        if column_name.startswith('"') and \
                            column_name.endswith('"'):
                            column_name = column_name.\
                                strip('"')
                            anomalies.append(i)

                modified_rows = []
                for row in csv_reader:
                    modified_row = []
                    for i, value in enumerate(row):
                        # Strip leading and trailing spaces
                        # from each value
                        modified_value = value.strip()
```

```
                        if i in anomalies:
                            # Remove first and last
                            # characters if anomaly is
                            # detected
                            modified_value = value[1:-1]
                        modified_row.append(modified_value)
                    modified_rows.append(modified_row)


                # Write the modified CSV data
                #to the new CSV file
                with open(new_csv_file, 'w', newline='')\
                    as new_file:
                    csv_writer = csv.writer(new_file)
                    csv_writer.writerow(header)
                    csv_writer.writerows(modified_rows)

                print("Anomaly characters have been\
                    successfully removed for", filename)
```

Listing 3.14: GTFS Static feeds format preprocessing

The function will then generate a new folder with the preprocessed stream.

3.3.2 Missing Shapes

A case that can happen quite often is the omission of one of the most important files in our GTFS Static feed: `shapes.txt`. Without this file, trajectories are not represented in any way and our space-time point sequences cannot be calculated. For example, in Belgium, the national railway company (SNCB) does not provide any `shapes.txt` file. The same phenomenon can be observed with data from the Kobe metro company in Japan.

The toolchain for Generating Transit Maps from Schedule Data [30] proposes a tool called *pfaedle* described by Brosi and Bast [31]. *pfaedle* is a precise map-matching tool for public transit feeds. It generates high-quality GTFS shapes from OSM[5] data. Based on the GTFS feed including Stops and Modes of Transports, it performs consistent map-matching on a given set of OSM input data.

So if data is missing, as it is for the SNCB, we can generate the `shapes.txt` file easily using *pfaedle*. First, we need to generate the corresponding OSM content. There are several ways of accessing the OSM dataset. The `planet.osm`

[5]https://www.openstreetmap.org/

file is the result of a detailed mapping of the entire earth. In the context of the SNCB we are not going to go that far and are therefore going to use Overpass Query Language (OQL) to call up the OSM API. We will be using Overpass Turbo,[6] a tool that allows you to make requests on the OSM API. But before going into the details of the query, we need to know what we are going to query. As the SNCB is Belgium's rail transport company, it is logical to think that we need to request a map of the Belgian railways. However, if we investigate the `routes.txt` file, we can see that there is not a single type of road in the dataset. Table 3.1 represents a `COUNT` performed on a `GROUP BY route_type` clause on the contents of the `routes.txt` file.

route_type	Count
3	603
2	133

Table 3.1: Route type count for SNCB

The result of this query tells us that the SNCB, the railway company, actually has 603 routes using trains, but also 133 routes using buses rather than trains. It is possible, for example, that buses are put in place because of works on railway lines. With this in mind, we need to query an OSM map that includes not only the Belgian railways but also all the bus routes. As the resulting datasets are fairly large files, Overpass Turbo may not be powerful enough to render all the data requested. It is therefore preferable to make the request in several parts, and later merge the data. Listing 3.15 represents a query written in OQL to retrieve Belgian bus lines.

```
[out:xml] [timeout:25];
 {{geocodeArea:Belgium}} -> .area_0;
(
    node["route"="bus"](area.area_0);
    way["route"="bus"](area.area_0);
    relation["route"="bus"](area.area_0);
);
(._;>;);
out body;
```

Listing 3.15: OSM query for belgian bus routes

This query extracts nodes, routes and relationships from the OSM database. The condition `["route"="bus"]` is used to select only bus routes. To obtain

[6] `https://overpass-turbo.eu`

railway routes, simply replace the condition with `["railway"]` to obtain all Belgian railways. Setting `geocodeArea` to the value "Belgium" means that these values can only be selected for Belgium.

Once we've obtained the two OSM files containing the Belgian railways and bus routes, we will need to merge all this together to give it to `pfaedle`. We can use the utility library `osmium` and its function `merge` which allows this. It is quite simple to use with the following command line:

```
osmium merge belgian_railways.osm belgian_bus_routes.osm -o sncb.osm
```

where `belgian_railways.osm` is the file containing the railways OSM data, `belgian_bus_routes.osm` is the file containing the bus routes OSM data, and `sncb.osm` is the desired name of the output file.

With this resulting OSM file, we can then use `pfaedle` to generate our GTFS feed with the `shapes.txt` file.

The following command line demonstrates the use of `pfaedle` with our data:

```
pfaedle -x sncb.osm sncb/
```

where `sncb/` is the folder containing our initial GTFS feed. A custom output folder can be made via the `-o` parameter. By default, the tool will put the result in a folder named `gtfs-out/`.

This series of operations therefore gives us a GTFS feed output folder containing a `shapes.txt` file for a dataset which did not initially contain one. Figure 3.7 shows the result after importing the `shapes` geometries.

We can see that the routes are nice and well-matched and that the stops are well-connected.

3.3.3 Reducing Study Scope

GTFS data can be voluminous due to its wide geographic coverage at a national, and in some cases international, level. As the GTFS format is intended to provide information about public transport systems across large areas, datasets can contain a significant amount of information, including timetables, routes, stops, etc.

However, it is important to note that in some specific study cases it may be necessary to reduce the dataset studied. Where the analysis focuses on a

Figure 3.7: Static feed visualisation for SNCB GTFS static feed

specific region, city or transport network, it may be appropriate to restrict the GTFS data to this limited geographical area. By reducing the dataset to only the relevant information for the study, the complexity and size of the data can be reduced, facilitating further processing and analysis.

The GTFS dataset can be reduced by filtering information according to specific criteria, such as the geographical location of stops or routes of interest. By determining which areas and data elements are relevant to the analysis, a more manageable dataset can be obtained, tailored to the objectives of the study.

It is important to note that the reduction of the dataset should be done thoughtfully and with the specific objectives of the study in mind. Careful selection of the data to be included allows the focus to be on the most relevant information, while reducing the complexity and size of the GTFS data to be processed.

The queries in Listing 3.16 can be used to filter and reduce the size of tables by specifying the study data to a limited area. For example, De Lijn, the Flemish public transport company in Belgium, extends its services throughout Flanders, the equivalent of 13656 km^2 including Brussels. The GTFS static dataset is therefore very large. Not restricting the calculation area would result in very long import calculations.

```
DELETE FROM shape_geoms
WHERE not ST_Intersects(shape_geom, ST_MakeEnvelope(4.2146,
    50.6836, 4.4912, 50.9289, 4326));

DELETE FROM stops
WHERE not ST_Intersects(stop_geom, ST_MakeEnvelope(4.2146,
    50.6836, 4.4912, 50.9289, 4326));

CREATE TABLE tmp_trips AS (
  SELECT DISTINCT t.*
    FROM shape_geoms AS s
  INNER JOIN trips AS t ON s.shape_id = t.shape_id
);
DROP TABLE trips;
ALTER TABLE tmp_trips RENAME TO trips;

CREATE TABLE tmp_stop_times AS (
    SELECT DISTINCT s.*
    FROM stop_times s
    INNER JOIN trips t ON t.trip_id = s.trip_id
);
DROP TABLE stop_times;
ALTER TABLE tmp_stop_times RENAME TO stop_times;

CREATE TABLE tmp_calendar as (
  SELECT DISTINCT c.* FROM calendar c
  INNER JOIN trips t ON c.service_id = t.service_id
);
DROP TABLE calendar;
ALTER TABLE tmp_calendar RENAME TO calendar;

CREATE TABLE tmp_calendar_dates as (
  SELECT DISTINCT c.* FROM calendar_dates c
  INNER JOIN trips t ON c.service_id = t.service_id
);
DROP TABLE calendar_dates;
ALTER TABLE tmp_calendar_dates RENAME TO calendar_dates;
```

Listing 3.16: Reduce study scope to Brussels

The first two delete queries filter out the geometries included in the bounding box defined by the `ST_MakeEnvelope()` function. Having done this, we now need to keep the trips that pass through the bounding box, i.e. those that have the same `shape_id` as the shapes we've kept.

We will do the same for the timetables defined by the `stop_times` table,

this time based on the `trip_id`. And finally, the same applies to the `calendar` and `calendar_dates` tables, which will allow us to define the days of service. After several tests, simply deleting rows that do not meet the conditions is a costly exercise. It is better to create a new table as a result of the corresponding selection query. This operation, in the context of De Lijn with Brussels, enables much less time-consuming calculations to be made. It is better to reduce the amount of data at source, even if it is still possible to calculate everything and apply these post-process filters.

Chapter 4

Import Other Standards

4.1 GTFS Realtime

As GTFS static data can now be imported into MobilityDB, let us focus on the standard often seen as its complement, GTFS Realtime. Again, we will see how to import this data into MobilityDB from the data source successfully.

4.1.1 Data Fetching

As said earlier, GTFS Realtime data is in Protobuf format but is only accessible via an HTTP stream. So we will have to establish a way to receive and process this data. Depending on the transport company, access to this stream is more or less simple. Often, requests or identifications are required to access a GTFS stream. And the ways of accessing, URL to the feeds, request headers, can differ.

We will see how to build a minimal and functional code in Go that allows to retrieve, insert and transform GTFS Realtime data.

Listing 4.1 is an example of code in Go to perform a GET request on the NY LIRR data stream. The content of the request will be stored in the `body` variable.

```
1 //Create request
2 req, err := http.NewRequest("GET", "https://api-endpoint.mta.
    info/Dataservice/mtagtfsfeeds/lirr%2Fgtfs-lirr", nil)
3 req.Header.Set("x-api-key", "API Key")
4
5 // Send GET request
```

```
client := &http.Client{}
resp, err := client.Do(req)

defer resp.Body.Close()

// Response handling
body, err := ioutil.ReadAll(resp.Body)
```

Listing 4.1: HTTP GET request in Go on New York LIRR trains

This code will allow fetching data from remote servers of public transport companies. The content will obviously be binary to be decoded thanks to the Protobuf library. Protobuf requires the user to compile the schema used with the Protobuf compiler and then import the compiled file for the language. The command to compile the file for Golang is as follows:[1]

```
protoc -I=$SRC_DIR --go_out=$DST_DIR $SRC_DIR/gtfs.proto
```

Where `$SRC_DIR` is the source directory, `$DST_DIR` is the destination directory and `gtfs.proto` is the file that defines the GTFS Realtime schema.[2] In our case, the file is compiled and the output is in a new package called `transit_realtime`. When imported in the main Go script, the Protobuf library for Go requires inserting the flow in function `Unmarshal()`, this will decode the binary data. Listing 4.2 shows the syntax in Go for the `Unmarshal()` function.

```
// Decode Feed Message
feed := &gtfs.FeedMessage{}
err = proto.Unmarshal(body, feed)
```

Listing 4.2: Decode GTFS feed

The `FeedMessage` object being the one that wraps all the entities of the GTFS Realtime stream, we now have to parse it. In our case, we are mainly interested in vehicle positions to be able to transform them into MobillityDB trips. We can therefore create a loop that checks that we are dealing with a `VehiclePosition`, and do an insertion in the corresponding case. Listing 4.4 demonstrates how the main loop is built. Listing 4.3 describes an example of a VehiclePosition message.

[1] https://protobuf.dev/getting-started/gotutorial/

[2] https://developers.google.com/transit/gtfs-realtime/gtfs-realtime-proto

```
entity {
  id: "PSGRKO_2023-05-29_V"
  vehicle {
    trip {
      trip_id: "PSGRKO_2023-05-29"
      start_date: "20230529"
    }
    position {
      latitude: 40.80691146850586
      longitude: -73.1199951171875
    }
    current_status: IN_TRANSIT_TO
    timestamp: 1685340900
    stop_id: "179"
    vehicle {
      id: "9066_PSGRKO"
      label: "9066"
    }
  }
}
```

Listing 4.3: Protobuf GTFS Realtime VehiclePosition response example

This entity provides an overview of vehicle position information, giving details of the vehicle itself, the current trip and its current location. The `trip_id` is given, as well as the vehicle's position given by its `latitude` and `longitude`. The attributes `current_status` and `stop_id` are used to identify the vehicle's status and the stop associated with its position. In our case, we see that the vehicle is in transit to the stop identified by 179.

Figure 4.1 shows the data that can be retrieved from the `VehiclePositions` messages according to GTFS Realtime. As a reminder, not all fields are mandatory and therefore depend on the GTFS Realtime specification.But even if some fields are not mandatory, they are still meaningful and provide important information to developers. For example, the `occupancyStatus` field lets users know whether the vehicle is crowded or not, which in turn indicates whether the line is busy or not. The `congestionLevel` field is a traffic study in itself, indicating whether the vehicle is currently in free-flowing or congested traffic.

In order to facilitate the transformation into MobilityDB, an additional check before insertion can be performed. Indeed, between two calls to a GTFS Realtime stream, the position of a vehicle may not have been updated. This implies that we could receive and insert the same position for the same

vehicle at the same timestamp twice. When creating our `tgeompoint` later, we must not have two coordinates with the same timestamp. On New York City buses GTFS Realtime stream, doing this check reduces the amount of inserted lines by a factor of 4. This gain is not negligible.

```
// Decode Feed Message
for _, p := range feed.Entity {
    if p.Vehicle != nil {
        // Get attributes
        // Check if exists in database
        // Insert in database
    }
}
```

Listing 4.4: VehiclePositions loop

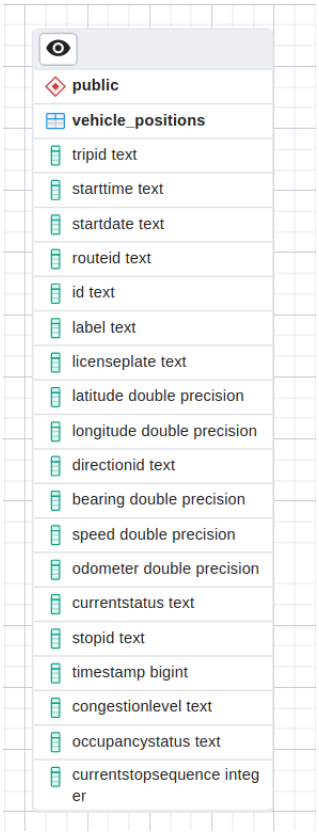


Figure 4.1: GTFS Realtime VehiclePositions schema

4.1.2 Data to MobilityDB

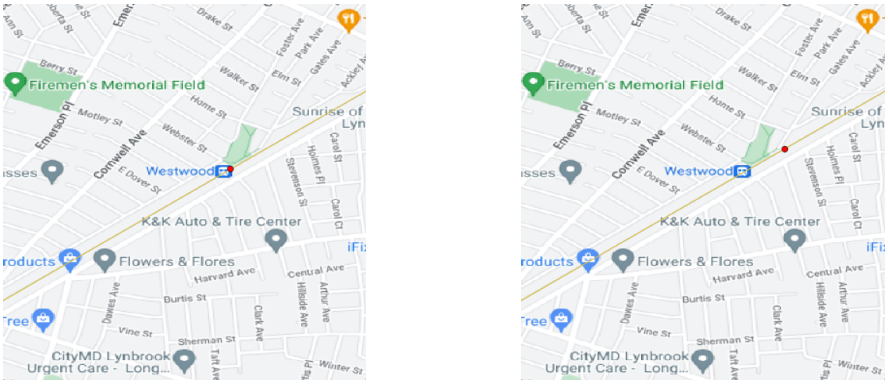
Now that the data is inserted and available, we have to transform all these vehicles coordinates into MobilityDB trips. Here, the treatment can be theoretically simpler than with GTFS static. Indeed, GTFS static required that we retrieve data from several tables, backward calculations with the days of service, find the timestamp according to the schedules of service,... Whereas here with GTFS Realtime, we immediately have a `Trip_Id`, `Vehicle_Id`, `latitude`, `longitude`, and the `timestamp` at which the position was captured. It is then sufficient to just group by `Trip_Id`, and `Vehicle_Id`, create a geometry by recovering the coordinates and finally aggregate this sequence in a `tgeompoint` ordering by `timestamp`. Listing 4.5 represents the SQL code to perform this operation. The `timezone` is to be adapted according to the working area. In the case of New York, the timezone 'America/New_York' is used.

```
INSERT INTO trips_mdbrt (
    trip_id,
    vehicle_id,
    trip
)
SELECT
    trip_id,
    vehicle_id,
    tgeompoint_seq(array_agg(tgeompoint_inst(point,
        (to_timestamp(timestamp) at time zone 'timezone'))
        ORDER BY timestamp))
FROM positions
GROUP BY trip_id, vehicle_id
ON CONFLICT (trip_id, vehicle_id)
DO UPDATE SET trip = EXCLUDED.trip;
```

Listing 4.5: Transform GTFS Realtime to MobilityDB

Again, we can use MOVE to visualise the imported data. Figure 4.2 shows the real-time position of a train on West Hempstead Branch on April 7, 2023, between 04:55:00 and 04:55:30.

From a macro point of view, the results obtained may be sufficient. However, a problem arises when analyzing the data from a micro point of view. Figure 4.3 represents the problem. The two red dots represent two train positions captured by the server, the blue curve represents the trip constructed from the GTFS-static data for that train line, and the orange curve is what our data catcher constructed.



(a) Real-time position of LIRR train on April 7, 2023, at 04:55:00

(b) Real-time position of LIRR train on April 7, 2023, at 04:55:30

Figure 4.2: Real-time position of LIRR train

We can immediately see that the orange curve represents a straight line between the two captured positions, while the blue curve does indeed follow the railway line. As a reminder, the theoretical curve is built according to the `shapes` file, present precisely to indicate perfectly the trajectory that the vehicles take. The frequency of update and reception of the data in real-time is not perfect (the coordinates are updated on intervals ranging from 20 to 30 seconds minimum), it is impossible at the time of the reception of the data, to design the perfect curve of the train, based only on these data.

However, we can easily overcome this problem. By coupling our positions to the data in the static data stream, we can build the right curve. As a reminder, the most important elements used in the MobilityDB transformation from GTFS Static were stops, stop arrival times, and the shape defined by the `shapes.txt` file. In this case, we can replace stops with vehicle positions, arrival times with observation timestamps, and the trajectory shape can be retrieved from the GTFS Static feed. The `stop_sequence` values can be interpreted as timestamps in ascending order.

The first step is to add the sequence number. Listing 4.6 describes this operation. This query isolates each `trip_id` by start date and assigns a line number to each captured position.

```
1 ALTER TABLE proto_vals ADD COLUMN no_seq INT DEFAULT 0;
2
3 UPDATE proto_vals AS p
4 SET no_seq = subquery.row_num
```

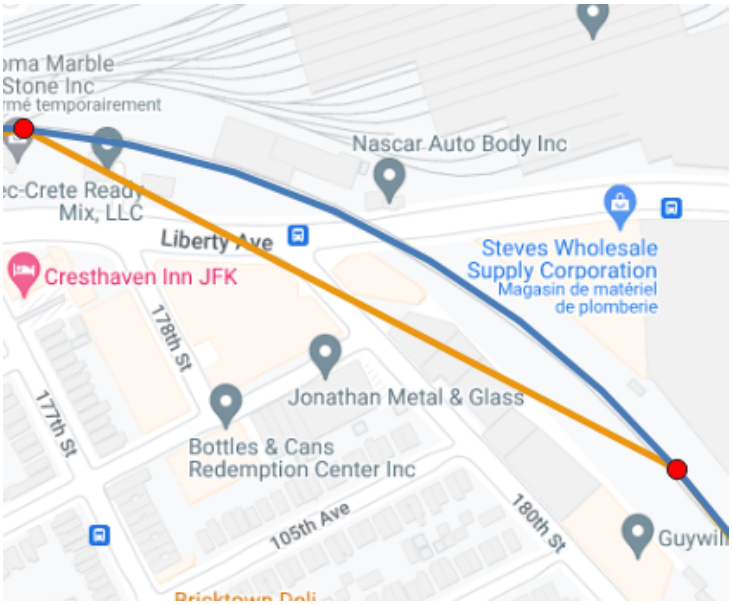


Figure 4.3: Map matching problem visualisation

```
FROM (
    SELECT
        tripID,
        startdate,
        timestamp_posix,
        ROW_NUMBER() OVER (PARTITION BY tripID, startdate
    ORDER BY timestamp) AS row_num
    FROM proto_vals
) AS subquery
WHERE p.tripID = subquery.tripID AND p.startdate = subquery.
    startdate AND p.timestamp = subquery.timestamp;
```

Listing 4.6: Add sequence number in Protobuf values

Once this has been done, we can follow the same transformation process as for GTFS Static, beginning by the **Trip Positions** step. For simplicity's sake, we are going to add some information to the transformation tables, such as the `start_date` which will replace the `service_date` table, and also transform the `timestamp` in POSIX format given by the Realtime stream into a `interval` type. The following command performs this transformation.

```
TO_CHAR((TO_TIMESTAMP(timestamp))::time, 'HH24:MI:SS')::interval
```

A final slight concern arises: duplicate positions. We saw in section 3.2.4

that MobilityDB imposes the uniqueness of positions for a single timestamp. When fetching data, we may have filtered so as not to insert the same line twice, but it can happen that we insert a new line for the same timestamp and `trip_id`. This happens when the complete line is different. In fact, the data provider may find it useful to give new information on a vehicle that hasn't changed position for a long time. We therefore need to remove duplicates in the same way as in section 3.2.4 for tuples `trip_id, start_date, stop_sequence`.

Continued execution of the slightly modified queries will then produce new `trips` in our table. Using the `shape` will have enabled us to perform offline map-matching. Figure 4.4 shows the new curves obtained.

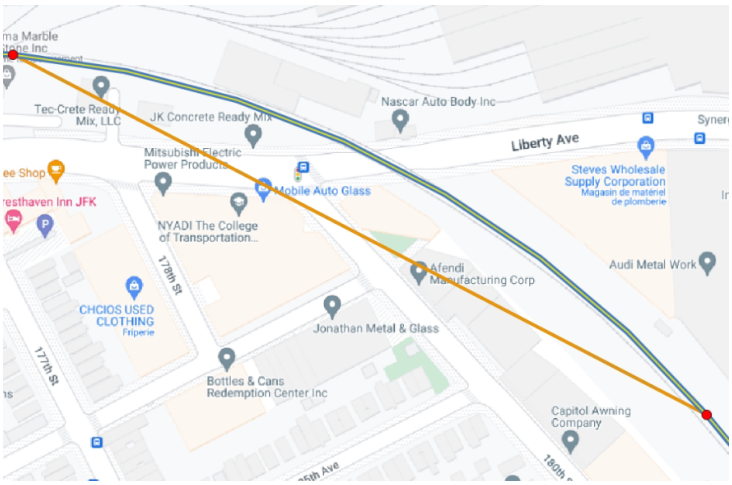


Figure 4.4: Solved map matching visualisation

We still have the blue curve representing the theoretical curve, the orange curve representing the Realtime curve if the points were linked sequentially, and in yellow we can observe a curve that perfectly matches the theoretical curve. This method shows us that it is possible to map-match GTFS Realtime easily if we have the corresponding GTFS Static files. This is logical, given that GTFS Realtime and GTFS Static are intended to complement each other.

4.2 NeTEx

GTFS Static and GTFS-Realtime flows confirmed that to build trips from public transport information, we needed transit points, transit times and a geometry representing the trajectory. The NeTEx standard has this information. Depending on which parts of NeTEx are used, the data will be accessible.

Parts 1 and 2 of the format can already be used to build NeTEx trajectories. Indeed, part 1 contains the topology of a network, i.e. its stops(`Stops`, `Quays`), its lines(`Line`), the arrangement of the stops within the lines(`PointsInSequence`), but also the planned crossing points per line(`JourneyPatters`), which makes it possible to define the trajectory of each line. Part 2 essentially contains the vehicle timetables for each stop and line (`ArrivalTime` and `DepartureTime`).

During preparatory work, we explored the STIB's NeTEx format, which uses part 4 of NeTEx (EPIP). This part has the particularity of proposing a minimum specification that meets the expectations of certain European Union bodies. This format contains a lot of information, such as network topology, but it also contain temporal information on vehicle timetables, declared as `passingTimes`. As a result, we were able to locate stops and lines, but we did not convert it successfully to MobilityDB.

There are tools allowing to transform an XML Schema Document but they have several disadvantages. XSD2DB[3] allows to create a database directly from the file. However the tool only works with SQL Server or OleDB. Another more general candidate that allows to give a schema is Altova's XMLSpy.[4] The tool is perhaps already more general but requires a financial contribution. We thus carried out an analysis of the XML file and establish a diagram according to the guide of conversion of Microsoft from XML to relational.[5]

4.2.1 Nordic Specifications

In short, parts 1 and 2 of NeTEx can be used to build MobilityDB trips, but also part 4. In this part, instead of parsing large XML files, we will use a

[3] `http://xsd2db.sourceforge.net/`

[4] `https://www.altova.com/xmlspy-xml-editor#database`

[5] `https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/dataset-datatable-dataview/deriving-dataset-relational-structure-from-xml-schema-xsd`

library developed by Entur,[6] a government-owned transportation company in Norway, and available in the toolbox of `gtfs.org`. Their tool actually converts a NeTEx stream into a GTFS stream as long as the stream complies with the Nordic specification.

The Nordic specification simply consists of the design of 2 archives, one containing an XML file containing all the stops and platforms in the service. The second archive contains the timetables for each line in the service, with the equivalent of one file per line.

The tool performs intelligent conversion by matching NeTEx elements with GTFS elements. For example, by locating a `Stop` or `Quay` in the NeTEx specification files, the tool will generate an additional line for the `stops.txt` file. Once the conversion is done, we can simply follow the steps to transform from GTFS to MobilityDB.

4.2.2 Other Specifications

In preparatory work we built an XML parser that allows us to extract informations from STIB-MIVB EPIP NeTEx feed. We used a python library, `xml.etree.ElementTree`, a simple and efficient library for XML parsing. The choice of python is due to its efficiency in managing files, but also to the various libraries that enable XML files to be processed. The general idea is to be able to extract information from XML files. To do this, the code will parse the XML file, converted into a tree using ElementTree. We perform recursive depth parsing, and when an element is encountered, we write it to a file in CSV format. The advantage of this technique compared to adding line by line directly is firstly to avoid the problems of interrelation dependencies. But also because it does not require a direct connection to the database. It is therefore a more general solution that can be used on all DBMS that can interact with CSV files. This led us to the schema described by Figure 4.5.

This schema shows us the data we can extract from a basic EPIP formatted NeTEx file, but also that the conditions to create a MobilityDB trip are respected. Indeed, if we do the analogy with GTFS Static, `stop_times` table is here `passingTimes`, `stops` are stored in `stopPlaces` and `shapes` table can be built with `pointsInSequence` and `scheduledStopPoints`. Once this step is reached, we can therefore follow the steps for GTFS from **Trip Positions**.

The problem with this simple technique is that it is depending on the

[6] `https://github.com/entur/netex-gtfs-converter-java`

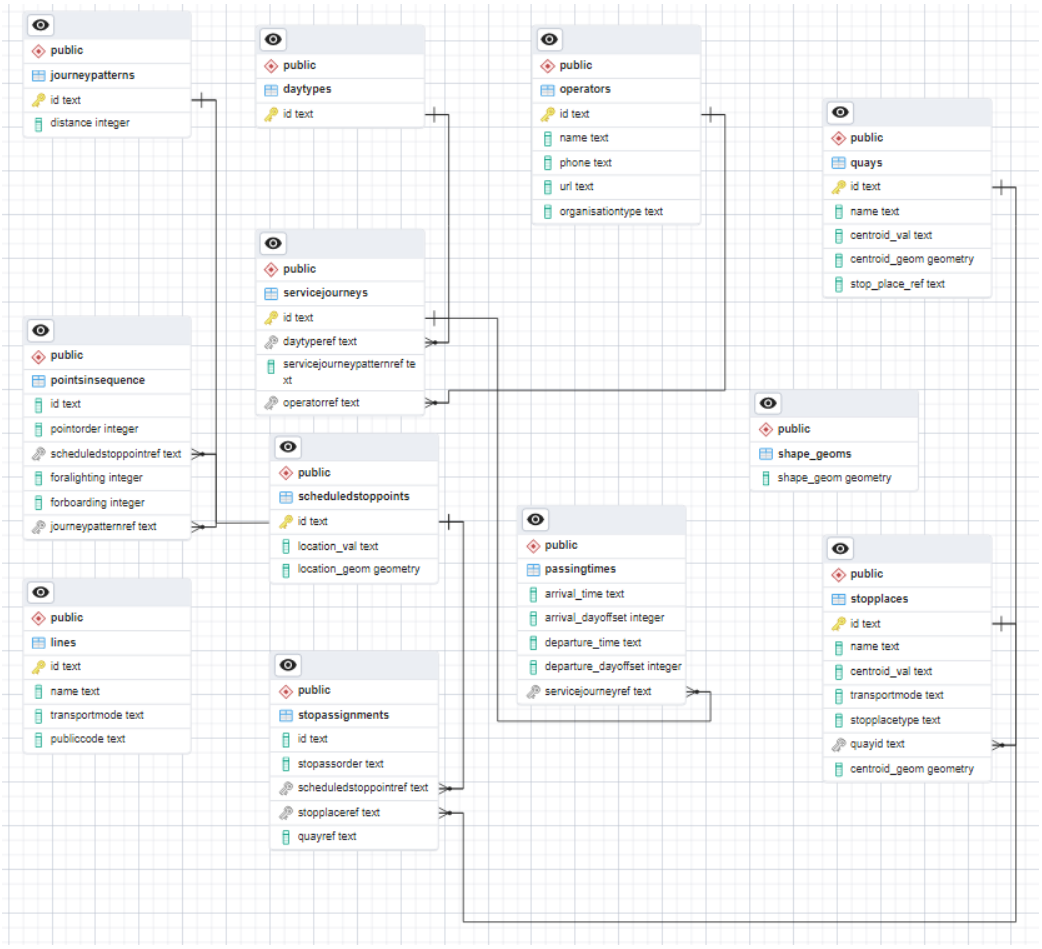


Figure 4.5: STIB NeTEx EPIP Schema

data in the NeTEx file, and no other information. We can see in the example of the STIB that even we take the points in sequence, the routes are not well map-matched, so we should maybe work on a more complete dataset, as they do in nordic specifications for example, or to develop a way that matches with other datasets such as GTFS or OSM.

4.3 SIRI

4.3.1 Data Fetching

Like GTFS Realtime, the data can be accessed regularly via HTTP streams. So we can take our Go server script and adapt it to SIRI's requirements. First of all, Listing 4.7 represents some example information extracted from an API call with codespace VM to the Norwegian transport company. The response is an XML output.

```
<RecordedAtTime>2023-08-04T13:47:05+02:00</RecordedAtTime>
<OriginRef>NSR:Quay:12011376</OriginRef>
<DestinationRef>GAR4.402</DestinationRef>
<VehicleLocation>
    <Longitude>5.38436737842858</Longitude>
    <Latitude>60.4827898554504</Latitude>
</VehicleLocation>
```

Listing 4.7: SIRI response extract

There is a lot more information than displayed, but we can already see a lot of interesting information. We can see the date and time of recording, where the observed vehicle is coming from, its destination and even its geographical position in real time. We can also see that some keywords correspond strongly to those in NeTEx, such as `Quay`, which is a type of information also used in NeTEx.

The Go script must therefore be adapted to retrieve the corresponding information. We will no longer be using the Protobuf library, but rather an XML library. Figure 4.6 shows the data that can be retrieved from a vehicle response.

Golang's XML library allows users to define the general structure of their XML. The XSD specification then makes it easy to remove this information. Listing 4.8 allows you to tell the library what the nested elements will look like. A Vehicle Monitoring contains several Vehicle Activies. These contain recording times, and finally vehicle details.

```
type SiriVehicleMonitoring struct {
  XMLName xml.Name `xml:"Siri"`
  VehicleActivities []SiriVehicleActivity `xml:"
   ServiceDelivery>VehicleMonitoringDelivery>VehicleActivity"
   `
}

```

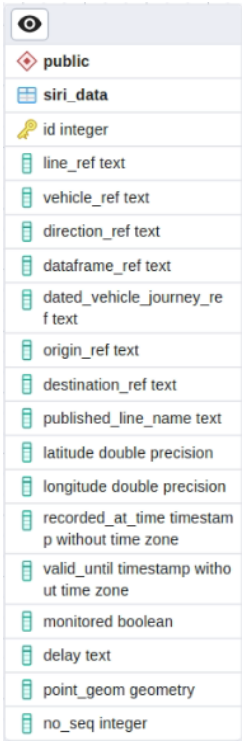


Figure 4.6: SIRI Vehicle Monitoring schema

```
type SiriVehicleActivity struct {
  RecordedAtTime string `xml:"RecordedAtTime"`
  ValidUntil string `xml:"ValidUntilTime"`
  MonitoredVehicleJourney MonitoredVehicleJourney `xml:"
   MonitoredVehicleJourney"`
}

type MonitoredVehicleJourney struct {
  LineRef string `xml:"LineRef"`
  VehicleRef string `xml:"VehicleRef"`
  DirectionRef string `xml:"DirectionRef"`
  OriginRef string `xml:"OriginRef"`
  DestinationRef string `xml:"DestinationRef"`
  PublishedLineName string `xml:"PublishedLineName"`
  VehicleLocation VehicleLocation `xml:"VehicleLocation"`
  FramedVehicleJourneyRef FramedVehicleJourneyRef `xml:"
   FramedVehicleJourneyRef"`
  JourneyPatternRef string `xml:"JourneyPatternRef"`
  Monitored bool `xml:"Monitored"`
```

```
  Delay string `xml:"Delay"`
}

type VehicleLocation struct {
  Longitude float64 `xml:"Longitude"`
  Latitude float64 `xml:"Latitude"`
}

type FramedVehicleJourneyRef struct {
  DataFrameRef string `xml:"DataFrameRef"`
  DatedVehicleJourneyRef string `xml:"DatedVehicleJourneyRef"
    `
}
```

Listing 4.8: Go XML types definitions

The same steps are performed as for GTFS Realtime: fetch, parsing, database check for duplicates, and finally insertion. We did not store every information that we can. Indeed, SIRI, as NeTEx for GTFS Static, contains way more information than can provide GTFS-Realtime. The schema defined in Listing 4.8 contains the information that we catch.

4.3.2 Data to MobilityDB

Once again, a naive insertion can represent trajectories in MobilityDB, but may not match the route. A coupling is then necessary with NeTEx to be able to transform the data correctly. To do this, a mapping on the lines must be performed. Listing 4.9 describes the necessary code to reach the **Trip Positions** steps on the transformation. The `trip_id` is matched with `dated_vehicle_journey_ref`, `route_id` with `line_ref`, `timestamp` with `recorded_at_time` and `startdate` with `dataframe_ref`. Once the matching is done, we can continue with the traditional way.

```
ALTER TABLE siri_data
ADD COLUMN no_seq INT DEFAULT 0;
UPDATE siri_data AS p
SET no_seq = subquery.row_num
FROM (
    SELECT dated_vehicle_journey_ref,
        dataframe_ref,
        recorded_at_time,
        ROW_NUMBER() OVER (
            PARTITION BY dated_vehicle_journey_ref,
            dataframe_ref
```

```
            ORDER BY recorded_at_time
        ) AS row_num
    FROM siri_data
) AS subquery
WHERE p.dated_vehicle_journey_ref = subquery.
   dated_vehicle_journey_ref
AND p.dataframe_ref = subquery.dataframe_ref
AND p.recorded_at_time = subquery.recorded_at_time;

DROP TABLE IF EXISTS trip_positions;
CREATE TABLE trip_positions (
    trip_id text,
    start_date date,
    stop_sequence integer,
    no_stops integer,
    route_id text,
    service_id text,
    shape_id text,
    stop_id text,
    arrival_time interval,
    perc float,
    point_geom geometry
);
INSERT INTO trip_positions (
    trip_id,
    start_date,
    stop_sequence,
    no_stops,
    route_id,
    service_id,
    shape_id,
    arrival_time,
    point_geom
)
(SELECT t.dated_vehicle_journey_ref,
    to_date(dataframe_ref, 'YYYYMMDD'),
    no_seq,
    MAX(no_seq) OVER (PARTITION BY t.
   dated_vehicle_journey_ref),
    static_routes.route_id,
    service_id,
    shape_id,
    TO_CHAR((recorded_at_time), 'HH24:MI:SS')::interval,
    point_geom
FROM siri_data t
    JOIN (
```

```
56          SELECT distinct trip_id,
57              route_id,
58              service_id,
59              shape_id
60          FROM trips
61      ) as static_routes on static_routes.trip_id = t.
    dated_vehicle_journey_ref
62 );
63 UPDATE trip_positions t
64 SET perc = ST_LineLocatePoint(shape_geom, point_geom)
65 FROM shape_geoms g
66 WHERE t.shape_id = g.shape_id;
```

Listing 4.9: Matching SIRI information with GTFS

Chapter 5

Experiments and Results

5.1 Importing Tests

Our means of importing are now developed, we will test them on different datasets of GTFS-static and GTFS Realtime.

5.1.1 GTFS Static

For these tests, the data was obtained from TransitFeeds dedicated feeds. We will import the static feed of the subway networks of Madrid, Spain and Kobe, Japan, as well as the GTFS data of Nairobi, Kenya. The data from the 3 cities will first be imported with our SQL import function, then it will be transformed for MobilityDB. The goal is to be able to visualise the static public transportation map but also to have a filled column in the table `trips_mdb`.

Metro de Madrid

When using the import function into SQL, an error is raised. It indicated that there were missing data in the columns. After investigation, it seems that in the header of the files of the Madrid-based company, the first character is an invisible character that causes this malfunction. This anomaly is visible for example on the TransitFeeds website in the files extracts. The character is interpreted as `"ï»¿"`. After using the preprocessing tool to delete the character in all files, the import is possible. Figure 5.1 shows the Madrid metro network and Figure 5.2 shows an extract from the trip column, indicating

that the import is successful.

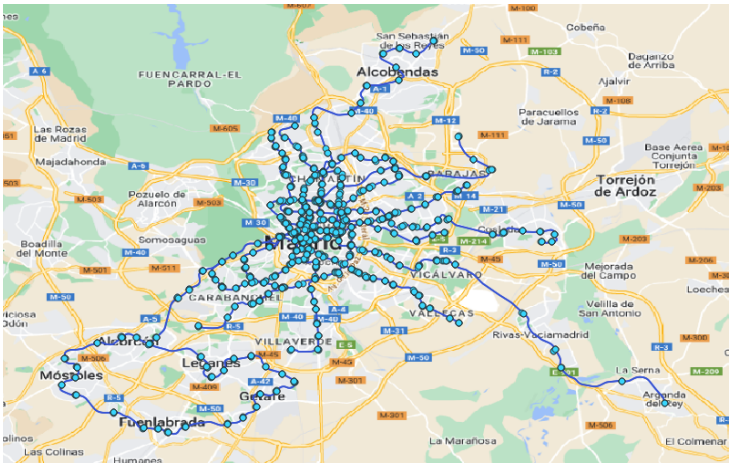


Figure 5.1: Static feed visualisation for Madrid GTFS static feed

```
"trip"
------------------------------------------------------------
"[0101000020E61[...]384440@2023-12-06 00:02:37+01, 010100...]"
"[0101000020E61[...]3D4440@2023-03-20 00:03:10+01, 010100...]"
"[0101000020E61[...]354440@2023-05-01 00:01:15+02, 010100...]"
"[0101000020E61[...]3D4440@2023-04-06 00:03:10+02, 010100...]"
"[0101000020E61[...]3D4440@2023-04-07 00:03:10+02, 010100...]"
```

Figure 5.2: Trip column for Madrid GTFS static feed

Kobe Subway

The import of the data into SQL is successful. The transformation for MobilityDB requires that the file `shapes.txt` is present and completed, indicating the geometry of the transportation means. Kobe Metro does not meet this requirement, which makes it impossible for us to transform this data into `tgeompoint`. However, we can preprocess our data by using *pfaedle* to generate the `shapes`. Figure 5.3 shows the map of Kobe, after generating the geometrical data that we could use.

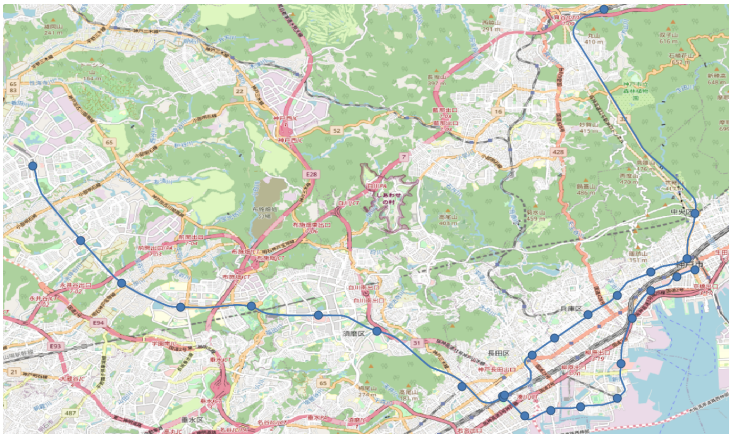


Figure 5.3: Static feed visualisation for Kobe GTFS static feed

```
"trip"
------------------------------------------------------------
"[0101000020E61[...]534140@2017-11-16 07:15:00+01, 010100...]"
"[0101000020E61[...]544140@2017-11-16 09:51:00+01, 010100...]"
"[0101000020E61[...]584140@2017-11-16 10:11:00+01, 010100...]"
"[0101000020E61[...]584140@2017-11-16 10:21:00+01, 010100...]"
"[0101000020E61[...]584140@2017-11-16 10:31:00+01, 010100...]"
```

Figure 5.4: Trip column for Kobe Subways GTFS static feed

Nairobi GTFS

No trouble importing the Nairobi data from DigitalMatatus. Figure 5.5 shows the Kenyan network, including stops and route lines, and Figure 5.6 shows a snippet of the trip column, indicating that the import is successful.

In these examples, we have observed that the method developed to import GTFS static data is efficient and functional. However, we have seen that this efficiency is conditional. Indeed, depending on how public transportation companies define their specification, the import may be subject to pre-processing as seen in the Madrid data. In addition, other cases may be found that have not been discussed here. For example, anomalies can be found in the Tunisian Medinine feed. Some fields are badly filled in, to name an empty field, it may be that the comma sequence in the CSV is of the following form: `", ,"`. This form can be used to pretend during the copy that it is a string containing a space. For a field requiring an integer, this

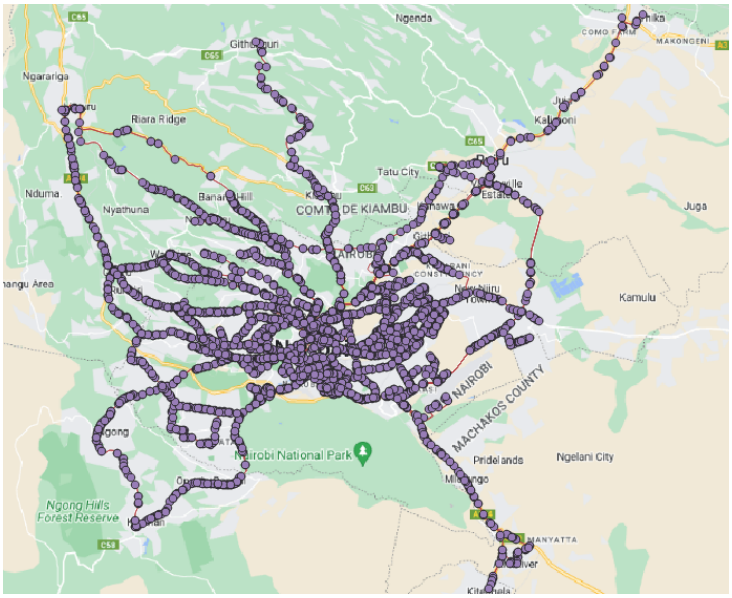


Figure 5.5: Static feed visualisation for Nairobi GTFS static feed

```
"trip"
------------------------------------------------------------
"[0101000020E61[...]ADF4BF@2020-12-12 06:00:00+01, 010100...]"
"[0101000020E61[...]ADF4BF@2019-10-20 06:00:00+02, 010100...]"
"[0101000020E61[...]ADF4BF@2019-12-12 06:00:00+01, 010100...]"
"[0101000020E61[...]ADF4BF@2020-10-20 06:00:00+02, 010100...]"
"[0101000020E61[...]ADF4BF@2019-12-12 06:00:00+01, 010100...]"
```

Figure 5.6: Trip column for Nairobi GTFS static feed

case causes an error. The quality of the encoding of the files is therefore very important.

Moreover, we have seen that a GTFS folder not containing the completed `shapes.txt` file cannot be transformed into a MobilityDB trip, since the calculations are performed on the geometries established on the basis of this file. This can also be extended to all the tables used for the calculations.

5.1.2 GTFS Realtime

Again, we will try to import real-time data from several streams. First, we will try to import data from the New York transit company, MTA. Buses and subways will be imported. In order to bring diversity to the sources of the feeds, we will also try to import train positions from another company, Transport NSW, which operates in Sydney, Australia.

New York Long Island Rail Road

After running the server script in Go, we can see that the table `proto_vals` does indeed contain data collected on vehicle positions and that the table `trips_mdbrt` does indeed contain trips in the form of `tgeompoint`.

Sidney Trains

Same observation as for the New York buses, the tables are correctly filled and the positions are correctly transformed for MobilityDB, the map matching problem is as expected also present.

New York Subways

After running the server script in Go, we can see first that the table `proto_vals` contains information but that the columns `longitude` and `latitude` are factually empty. This is not really surprising. Indeed, the position of the subways being indoor and underground, it is very difficult to give a precise geographical position. The server code being based on `latitudes` and `longitudes` to establish the spatiotemporal geometries, it is impossible to establish a trip from so little data.

Even if it is harder, it is not completely impossible to establish a trip. Other information can be used to establish approximate or even precise positions. For example, thanks to the column `CurrentStatus`, we can know if the subway is stopped or moving, this information coupled with the column `StopID` can allow an estimation of the position. The `TripUpdate` message, if present, can also be used to establish positions. A trip date can indicate what time a subway will arrive at the stop. This gives precise and valuable information.

Apart from the problems of map-matching and the potential lack of geographic coordinates when receiving the data, the results are quite conclusive. The trips can be imported into MobilityDB. The next section will show some use cases of MobilityDB on public transportation company data.

5.1.3 NeTEx and SIRI

We are going to show you how to use the NeTEx import function. As indicated, we are going to use the NeTEx to GTFS converter to obtain a more universal and less versatile GTFS feed for importing.

NeTEx Oslo Ruter

Here we will try to convert the Oslo buses NeTEx dataset. As mentioned earlier, obtaining NeTEx datasets is complicated, but Entur provides it easily. The Norwegian company provides the archives required for conversion. Once we've obtained a GTFS file, we can use our simple import method. Figure 5.7 shows the topology of the bus network in central Oslo. We had to reduce the scope of the study. Oslo is a very large city, and bus services cover hundreds of square kilometers. Listing 5.8 shows an extract from the `trips_mdb` table, demonstrating the success of importing city data into MobilityDB. Interestingly, the routes are indeed map-matched, and the conversion generates a more than adequate `shapes.txt` file. We can also see that in the Oslo fjord, we have a trajectory that starts from the port. The NeTEx stream also contained information on the trajectories of ferries crossing the fjord and performed the conversion.

SIRI Norway Realtime Vehicle Positions

We are going to try converting the data caught on Entur's SIRI feed using the server developed in Chapter 6. After launching the server, we see that the table `siri_data` is correctly filled. After transformation, the table `trips_mdbrt` is correctly filled with `tgeompoint`.

5.2 MobilityDB Use Cases

With our means of importing data now developed and tested, we can explore use cases with MobilityDB, the Moving Object DBMS. This section will

Figure 5.7: Static feed visualisation for Oslo NeTEx feed

```
"trip"
-------------------------------------------------------------
"[0101000020E61[...]F54D40@2023-10-08 06:46:00+02, 010100...]"
"[0101000020E61[...]F54D40@2023-10-15 06:46:00+02, 010100...]"
"[0101000020E61[...]F54D40@2023-10-22 06:46:00+02, 010100...]"
"[0101000020E61[...]F54D40@2023-10-29 06:46:00+01, 010100...]"
"[0101000020E61[...]F54D40@2023-10-05 06:46:00+01, 010100...]"
```

Figure 5.8: Trip column for Oslo NeTEx feed

therefore show the uses of MobilityDB on GTFS Static and GTFS Realtime data, but also on comparisons of the two streams. The data used will come from New York Long Island Rail Road and Brooklyn Buses, New York, but the tests are reproducible on any GTFS dataset importable by the means deployed in this thesis. For some of the visualisations, the Grafana tool will be used to dashboard some results in a more visual way. Some of the queries are inspired from Juan Godfrid's paper [35], which uses MobilityDB to analyze public transportation traffic flows in Buenos Aires using MobilityDB. So this section will also be used to check that these queries are still up to date. By way of comparison, we will also write and test queries written using PostGIS

alone, to see the computational as well as written benefits of MobilityDB. Tests for PostGIS alone will be carried out based on the `trips_input` tables containing all the GTFS Static route points, and an additional table named `trips_traj` which will contain a geometry representing the line connecting all the route points. For GTFS Realtime, the table `proto_vals` will be used for pure PostGIS. The given execution times are computed on average on 50 executions.

5.2.1 Closest Line to a Point of Interest

We will detail a query that lists the train lines that pass closest to New York Times Square. PostGIS functions allow to calculate distances between several geometries, but MobilityDB allows to calculate distances between time points. We can use the `shortestLine()` function of MobilityDB which takes as parameters a `tgeompoint` and another geometry and returns the distance. So we will use the static feed to find out which Long Island Rail Road lines are closest to Times Square. Figure 5.9 shows the distance between each train line and Times Square. Listing 5.1 shows the query to compute this. Where table `trainlines` is a simple table containing basic information about the train lines obtained from a join between `trips` and `routes` table.

```
WITH trip_distances as (
SELECT tl.route_long_name as line, ST_Length(shortestLine(
    trip, ST_SetSRID(ST_MakePoint(-73.9851, 40.7589),2831)))
    AS distance
FROM trips_mdb AS st
JOIN trainlines AS tl ON st.trip_id = tl.trip_id
)
SELECT line AS metric, AVG(distance) as value
FROM trip_distances
GROUP BY line
ORDER BY value ASC;
```

Listing 5.1: Compute distance between a trip and a point query

The metric used in the example here is the meter. We can see that the Far Rockaway, Port Washington and Long Beach branches are approximately equidistant from Times Square. This is because these 3 lines pass through Penn Station, the closest train station to Times Square, located a little more than 1 km from it.

If pure PostGIS had been used, the query would have been very similar. From the geometry, the distance between the point and the line is easily cal-

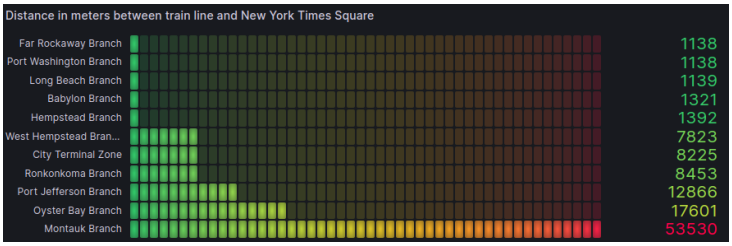


Figure 5.9: Closest train lines to New York Times Square

culated. The difference is mainly computational. The pure PostGIS query is slightly slower than the query using MobilityDB. 1.22 seconds on average for MobilityDB versus 2.091 seconds for PostGIS. We see here that MobilityDB is interesting to compute distances between `tgeompoint` and other geometries.

5.2.2 Average Speed of Vehicles Grouping by Day or Hour

Another feature of MobilityDB is the ability to calculate the average speed of a trip. It is calculated by combining the function `speed()` and the function `twavg()` which calculates a time-weighted average. This can be calculated on both feeds, static and real-time, which will allow us to see the difference in speed between the theoretical times and the real times observed. Listing 5.2 represents the code to perform this operation.

```
1  with th AS (
2  -- Average speed by starting hour (ST)
3  SELECT AVG(twavg(speed(trip))) AS static, date_trunc('hour',
       startTimestamp(trip)) AT time zone 'America/New_York' AS
       time
4  FROM trips_mdb
5  GROUP BY time
6  ORDER BY time),
7
8  rt AS (
9  --Average speed by starting hour (RT)
10 SELECT AVG(twavg(speed(Trip))) AS realtime, date_trunc('hour'
11     , startTimestamp(trip)) AT time zone 'America/New_York'
12     AS time
13 FROM Trips_mdbrt
14 GROUP BY time
```

```
ORDER BY time)
SELECT realtime, static, rt.time
FROM rt INNER
JOIN th ON rt.time = th.time;
```

Listing 5.2: Compute average speed of trains grouping by hour using MobilityDB

`date_trunc` function allows us to group by the unit of time we want. If we want to compute the same result but group by day, we just have to replace `'hour'` by `'day'` in the function parameters. On the PostGIS/PostgreSQL side, it is more difficult to write the query, indeed Listing 5.3 shows how to simply compute the average speed of trips from GTFS Realtime data.

```
SELECT
    tripid,
    DATE_TRUNC('hour', timestamp) AS hour,
    AVG(speed) AS average_speed
FROM (
    SELECT
        tripid,
        TO_TIMESTAMP(timestamp) AS timestamp,
        ST_Distance(point, LAG(point) OVER (PARTITION BY
        tripid ORDER BY timestamp)) AS distance,

        EXTRACT(EPOCH FROM TO_TIMESTAMP(timestamp) -
        LAG(TO_TIMESTAMP(timestamp)) OVER
        (PARTITION BY tripid ORDER BY timestamp)) AS
        time_diff,
        ST_Distance(point, LAG(point) OVER
        (PARTITION BY tripid ORDER BY timestamp)) /
        NULLIF(EXTRACT(EPOCH FROM TO_TIMESTAMP(timestamp) -
        LAG(TO_TIMESTAMP(timestamp)) OVER
        (PARTITION BY tripid ORDER BY timestamp)), 0) AS
        speed
    FROM
        proto_vals
) AS subquery
WHERE time_diff != 0
GROUP BY tripid, DATE_TRUNC('hour', timestamp)
ORDER BY tripid, hour;
```

Listing 5.3: Compute average speed of trains grouping by hour using PostGIS/PostgreSQL

The query requires manually computing the speed between each succession of consecutive points. In order to have the same result as for Mobil-

ityDB, we have to compute the same query for GTFS Realtime data. In terms of performance, in average, MobilityDB computes the average speed from both feeds in 2.401 seconds while PostGIS/PostgreSQL requires more than 20 seconds to compute the whole query.

Figure 5.10 shows the results on the Long Island Rail Road trains.

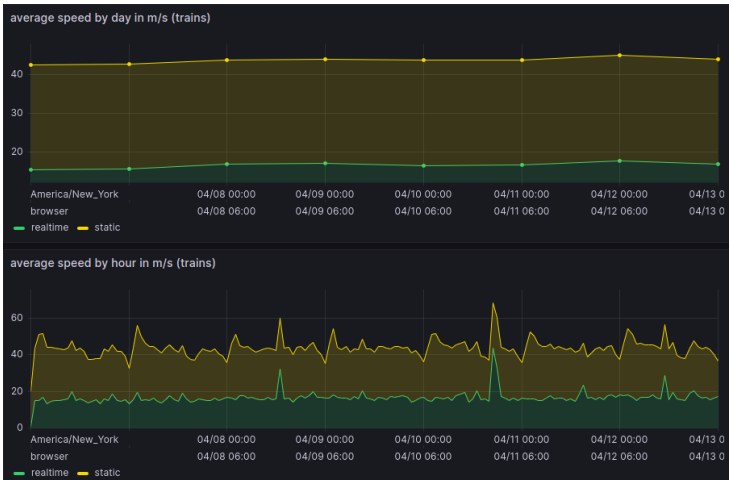


Figure 5.10: Long Island Rail Road Trains Average Speed

We see immediately that the static feeds have a higher speed than those captured in real-time. This is normal because these are theoretical feeds and it is very complicated to adopt these speeds in real life. The goal is obviously for the transport companies to get as close as possible to the theoretical curve. We also see that the "peaks" of speed are however respected, the vehicles know how to accelerate when it is necessary. There is no notable difference between one day of the week and another, or even between one hour of the day and another. This makes sense because trains are on railways, so they are cut off from general traffic. We can therefore ask ourselves what this kind of graph would look like on a feed dedicated to buses, which are much more impacted by traffic, and this is what Figure 5.11 shows.

Like the trains, the buses run slower than on the theoretical curves. The hourly curve already seems less regular and shows peaks in the evening, when there is less traffic, so the buses logically drive faster. On the daily curve, we see a slight speed peak on April 8 and 9, 2023, which correspond to the weekend. On average, buses run at an average speed between 10 and 18 km/h, while trains run at an average speed of around 70 km/h. We can see that

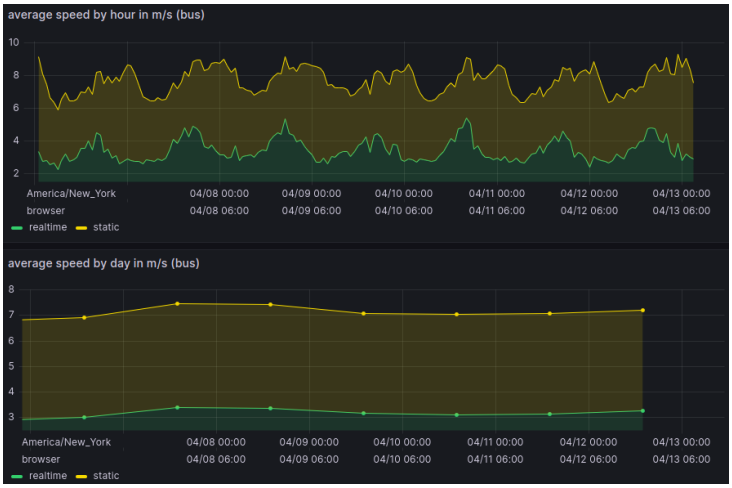


Figure 5.11: New York Buses average speed

MobilityDB is more interesting than PostGIS in terms of writing, thanks to its compact way of writing queries, but also in terms of performance, because of the more compact table than PostGIS uses to compute the speed.

5.2.3 Compute Delay between Static and Realtime Feed

We have seen that public transportation in New York is generally quite slow. So let us calculate the average delay of the trains. To do this, we will use the function `timespan()` of MobilityDB. It allows to calculate the total duration of a trip. It is thus enough for the same `Trip_Id`, to calculate the duration of the theoretical trip, and the real-time, and to simply subtract these two times. Listing 5.4 show the corresponding query.

```
SELECT bl.route_long_name AS metric,
    AVG(EXTRACT(EPOCH FROM timespan(rt.trip))/60 -
    EXTRACT(EPOCH FROM timespan(st.trip))/60) AS value
FROM trips_mdbrt AS rt
  JOIN trips_mdb AS st ON rt.trip_id = st.trip_id
  JOIN trainlines as tl on rt.trip_id = tl.trip_id
GROUP BY route_long_name
ORDER BY value DESC;
```

Listing 5.4: Compute average delay between Static and Realtime Feed using MobilityDB

We can see that the query is still compact as well as we have seen for Use Case 2. The `timespan` function computes easily the duration of a trip. Listing 5.5 shows how to compute the same query using PostGIS/PostgreSQL.

```
WITH static_times AS (
    SELECT trip_id, MAX(EXTRACT(EPOCH FROM t)) -
    MIN(EXTRACT(EPOCH FROM t)) AS static_dif
    FROM trips_input
    GROUP BY trip_id, date
), rt_times AS (
    SELECT tripid, MAX(timestamp) - MIN(timestamp) AS rt_dif
    FROM proto_vals
    GROUP BY tripid, TO_DATE(startdate, 'YYYYMMDD')
), average_dif AS (
    SELECT tl.route_long_name, AVG(static_dif/60 - rt_dif/60)
    AS dif
    FROM static_times
    JOIN rt_times ON tripid = static_times.trip_id
    JOIN trainlines AS tl ON tl.trip_id = static_times.
   trip_id
    GROUP BY tl.route_long_name
)
SELECT route_long_name, dif
FROM average_dif;
```

Listing 5.5: Compute average delay between Static and Realtime Feed without MobilityDB

In this case, we have to manually compute the duration of a trip. The query takes longer to write than using MobilityDB functions. In terms of performance, MobilityDB takes 2.408 seconds on LIRR data while PostGIS takes 11.339 seconds.

Figure 5.12 shows the delay observed on each of the Long Island Rail Road train lines in minutes. The results coincide with the results observed with the average vehicle speed. Indeed, the vehicles were very slow compared to the theoretical average speeds.

Among these results, Oyster Bay and Montauk branches stand out, in fact, they are very late compared to the others. These two branches are known to be quite slow to run. Oyster Bay is very slow, while Montauk is the longest line in the network, which also explains the potential delays. Also, the sample of real-time trips is limited to one week of service. It may have been an exceptionally bad week in New York City, that rail traffic exploded that week. Analyzing other such samples might be interesting. It is

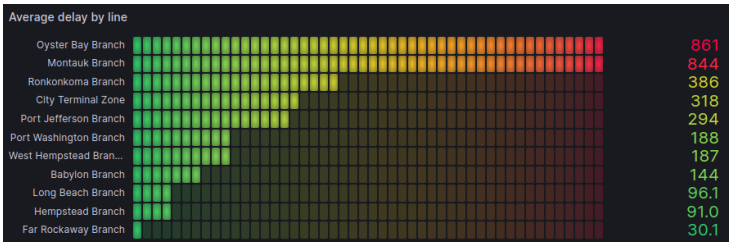


Figure 5.12: Long Island Rail Road average delay by line

also possible that the New York companies do not take external factors into account enough and overestimate the average speed of their transport, which would explain this gap between the theoretical and the real-time. MobilityDB is an interesting performer for delays computation. The `timespan` function is efficient and easy to use.

5.2.4 Visualisation of Delays by Line

After having identified which lines were subject to delays, it would be interesting to identify on which sections of the line delays are observed. A heatmap would be easily computable with the information we have. We can establish all the points where the real-time vehicle is within a certain distance of the static vehicle. The function `tdwithin()` allows us to retrieve the sections where the two trips are close to a certain distance. By running the query with several distances as parameters, and storing the results in multiple tables, we can visualise the segments as a heatmap. Listing 5.6 shows the computation of a single heatmap table.

```
1  -- Segments where real-time is less than 100 meters from
       static
2  DROP TABLE if exists heatmap1;
3  CREATE TABLE heatmap1 (
4    trip_id text,
5    seg_geom geometry
6  );
7  INSERT INTO heatmap1(
8      trip_id,
9      seg_geom)
10 SELECT ST.trip_id,
11         getvalues(atPeriodSet(ST.Trip, getTime(atValue(
12             tdwithin(ST.Trip, RT.Trip, 100), TRUE))))
13 FROM trips_mdb ST,
```

```
    trips_mdbrt RT
WHERE ST.trip_id = RT.trip_id
    AND ST.Trip && expandSpatial(RT.Trip, 100)
    AND atPeriodSet(ST.Trip, getTime(atValue(
        tdwithin(ST.Trip, RT.Trip, 100), TRUE))) IS NOT NULL
ORDER BY ST.trip_id;
```

Listing 5.6: Select segments where realtime are less than 100 meters from static query

Writing the same query using PostGIS alone is quite complicated. Linking temporal data to trajectories requires exceptional temporality management. Indeed, the timestamps contained in real-time data are very rarely at the same instants as the data contained in static feeds. MobilityDB makes it easy to manage this.

Figure 5.13 shows the segments of a train on West Hempstead Branch. The visualisation coincides with what has been seen previously. Indeed, on average the theoretical trains run faster than the real-time trains. The few moments when the trains are close theoretically and in real-time take place near the stops. Moreover, when visualising with MOVE, we can observe that the real-time trains take time to stop at the stops while the theoretical trains do not necessarily stop. This supports the hypothesis that New York may be overestimating its network and resources to some extent.

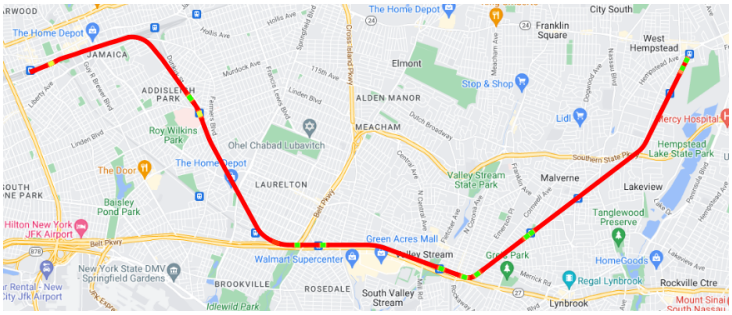


Figure 5.13: Long Island Rail Road West Hempstead Branch delay heatmap

5.2.5 Dynamic Visualisation of the Delay on a Line

Using a visualisation tool such as MOVE on QGIS, we can isolate the routes of interest to us. In the same way as the previous use case, we are going to visualise the same line of trains. Long Island Rail Road West Hempstead

Branch. The dynamic visualisation is available on the official Github repository for this work. As expected, we can see a blatant delay between the real-time train and the theoretical train over a large majority of the trip. Figure 5.14 shows the positions of the two trains during the journey. In red is the realtime train, in blue the theoretical train based on static flows.

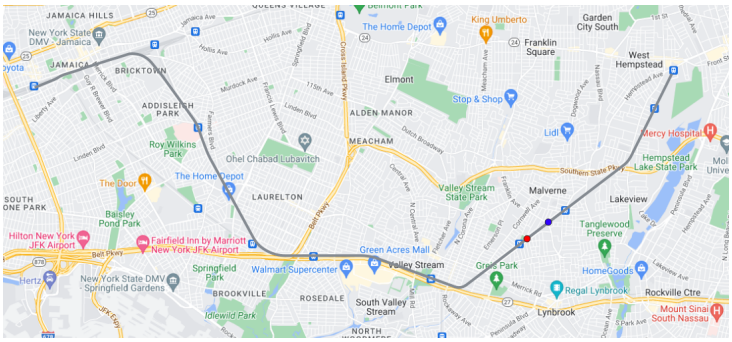


Figure 5.14: Long Island Rail Road West Hempstead Branch delay visualisation

5.2.6 Compute Static and Realtime Arrival and Departure Times

The two visualisations we have provided for delays work on a macro level. Indeed, we have an idea of the delay on a line thanks to the heatmaps but also thanks to the dynamic visualisation, but from a transport company's profitability and efficiency point of view, knowing whether the means of transport is late at the stops is more valuable. That is what we are going to try and calculate in this use case. First, we will try to isolate the trip we are interested in. In this case, we will retrieve the July 4, 2023 trip with `trip_id 'GO102_22_V4_1706'`. Next, of course, we need to list all the stops on the observed line. This is done by making a join between the `stops` table and the `stop_times` table. We also take the opportunity to retrieve the arrival and departure times at each of these stops. To calculate our train's arrival and departure times in real time, we need to define its position when stationary. As shown in Figure 5.15, we can't retrieve the times at the exact position when stopped, because the vehicle's position is not exact. Red is the observed position of the vehicle in realtime at the WestWood stop, while gray is the position extracted from the static.

Figure 5.15: Position difference between Static and Realtime

The two positions are very close, but makes MobilityDB's `atGeometry` function unusable. So we are going to calculate the observed position closest to the stop. To do this, we will retrieve all the points along the train's path, and calculate the distance between each stop. Then group them by stop and select the minimum for each. This gives us the minimum distance between the stops and our observed points. Listing 5.7 shows the code needed to perform these steps.

```
WITH train AS (
    SELECT *
    FROM trips_mdbrt
    WHERE trip_id = 'GO102_22_V4_1706' AND
        starttime = '2023-04-07 05:53:42'
    LIMIT 1
),
line_stops AS (
    SELECT DISTINCT stops.stop_id,
        stop_name,
        st.stop_sequence,
        st.arrival_time,
        st.departure_time,
        stops.stop_geom
    FROM train
    JOIN stop_times st ON st.trip_id = train.trip_id
    JOIN stops ON st.stop_id = stops.stop_id
    ORDER BY st.stop_sequence
),
train_points AS (
    SELECT
        (st_dumppoints(trip::geometry)).geom
    FROM train
```

```
),
distances AS (
    SELECT stop_id,
        stop_name,
        arrival_time,
        departure_time,
        st_distance(train_points.geom, line_stops.stop_geom),
        train_points.geom
    FROM train_points, line_stops
),
closest AS (
    SELECT d.stop_id,
        d.stop_name,
        d.geom,
        d.st_distance,
        d.arrival_time,
        d.departure_time
    FROM distances d
    JOIN (
        SELECT stop_id,
            MIN(st_distance) AS min_distance
        FROM distances
        GROUP BY stop_id
    ) b ON d.stop_id = b.stop_id
        AND d.st_distance = b.min_distance
)
SELECT DISTINCT stop_id,
    stop_name,
    startTimestamp(atGeometry(trip, c.geom)) AS real_arrival,
    endTimestamp(atGeometry(trip, c.geom)) AS real_departure,
    arrival_time,
    departure_time
FROM closest c, train
ORDER BY arrival_time;
```

Listing 5.7: Compute arrival and departure times for a trip

Finally, we have all the information we need to visualise our delays. The MobilityDB function `atGeometry()` allows us to retrieve the `tgeompoint` at the points we've calculated. The `startTimestamp` and `endTimestamp` functions then calculate the start and finish times at the position. Figure 5.16 shows the arrival and departure times for our line. We can see that this line is quite on time, usually even slightly ahead of schedule. We can also see that in the GTFS dataset, arrival and departure times are the same, which can generally be a source of observable delay in real time. Leaving on time

means not stopping to let passengers on and off. This principle can even lead to delays, as it overestimates the efficiency of services.

Arrival and departure for realtime and static

stop_id	stop_name	real_arrival	real_departure	arrival_time	departure_time
102	Jamaica	2023-04-07 05:53:42	2023-04-07 06:40:22	06:39:00	06:39:00
184	St. Albans	2023-04-07 06:45:45	2023-04-07 06:46:58	06:46:00	06:46:00
219	Westwood	2023-04-07 06:55:57	2023-04-07 06:56:20	06:56:00	06:56:00
142	Malverne	2023-04-07 06:58:25	2023-04-07 06:58:46	06:58:00	06:58:00
124	Lakeview	2023-04-07 07:01:11	2023-04-07 07:02:14	07:00:00	07:00:00
85	Hempstead Garde...	2023-04-07 07:03:27	2023-04-07 07:04:19	07:02:00	07:02:00
216	West Hempstead	2023-04-07 07:05:52	2023-04-07 07:06:13	07:07:00	07:07:00

Figure 5.16: Arrival and departure times for LIRR train

This table also shows us that even if theoretical delays can occur during the journey, as visualised in Use Case 6, we need to be more pragmatic and also check arrival times at stops. Be careful, however, to avoid any speeding that might follow a delay.

5.2.7 Export Stop Times from MobilityDB

This use case is in fact an extension of the previous one. The previous one allowed us to select arrival and departure times in real time and compare them with the static feed. It would be interesting, for example, if someone decided to publish their own GTFS feed, to be able to do so with MobilityDB. The most important thing at this stage is to have a table containing `tgeompoint` as well as a table listing the various service stops. By adding the sequence number to these stops, we actually have the complete contents of a `stop_times` table. This table can then be exported as a CSV file. Listing 5.8 shows the final query allowing to select a full GTFS `stop_times.txt` file. The absence of the CTE train allows us to select all trips present in our initial table, without filtering.

```
1 CREATE TABLE custom_stop_times AS (
2 -- Previous CTEs
3
4     SELECT DISTINCT c.trip_id,
5         stop_id,
```

```
6           stop_name,
7           startTimestamp(atGeometry(trip, c.geom)) AS
    arrival_time,
8           endTimestamp(atGeometry(trip, c.geom)) AS
    departure_time
9       FROM closest c JOIN (SELECT trip_id, trip from
    trips_mdbrt) t ON t.trip_id = c.trip_id
10      ORDER BY c.trip_id, arrival_time;
11 );
12
13 COPY custom_stop_times TO 'path/to/own/gtfs/stop_times.txt'
    DELIMITER ',' CSV HEADER;
```

Listing 5.8: Select Stop Times from trips

5.2.8 City Area Coverage

This case is inspired by the article by Nishino et al [20]. In this paper, they try to calculate the percentage of the population that could be alerted if a calamity occurred at a given time `t` in the city of Brisbane. The means of alerting the population relies on loudspeakers installed on all the buses of the city. The study focuses on the percentage of the population based also on the demographics of the city. Here we will simply try to calculate the percentage of Brooklyn's surface that could be covered under similar conditions. A circular area of 7 km radius has been developed around the center of Brooklyn. The geometry is stored in a table as well as its computed area. We assume that the buses can cover a circular area of a radius of 400 meters around them. Figure 5.17 shows the audible area that the buses could cover in real time on April 8, 2023, at 09:50:00.

The internal area covered represents 64.91% of the total area delimited of Brooklyn. The result is obtained by retrieving the points at a time `t`. Using MobilityDB function `atTime()` allows us to get the geometry of trips at a given instant. From there, the points that intersect the entire area are kept, and the PostGIS function `ST_Buffer()` is used to represent the points as having a radius of 400 meters. In order to avoid double counting the overlapping areas, we will use `ST_Union()`, which will represent all the points as a single geometry. PostGIS then finally has the function `ST_Area()` which calculates the area of a geometry. The ratio of the two surfaces obtained gives us the percentage of surface covered by the buses. Listing 5.9 shows the query to get this result. Temporary table `filtered_table` stores trips

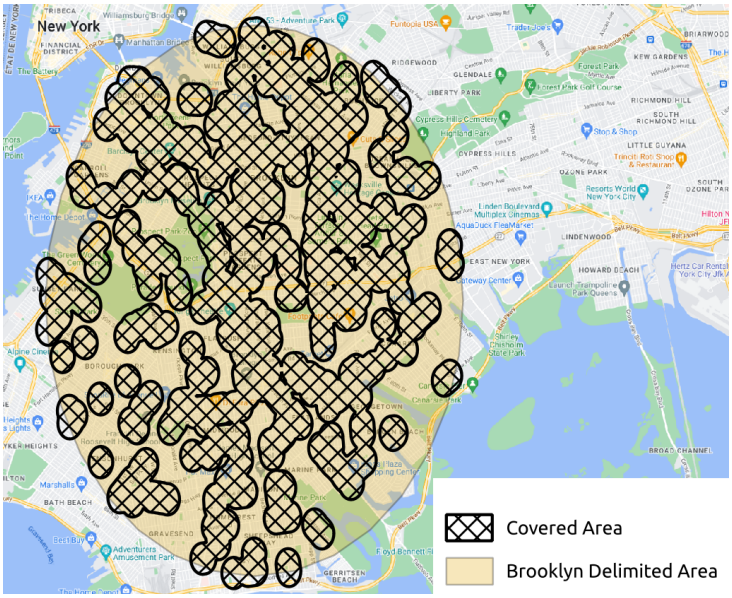


Figure 5.17: Audible area of Brooklyn on April 8, 2023, at 09:50:00

that are present at the current time and delete the others. It also stores the trip geometry used to perform the calculations.

```
-- Creating a table to store circle geometry
DROP TABLE IF EXISTS circle_geometry;
CREATE TABLE circle_geometry (
    geom GEOMETRY(Polygon, 2831),
  surface float
);

-- Inserting circle geometry and area
INSERT INTO circle_geometry (geom, surface)
VALUES (
    ST_Buffer(ST_MakePoint(-73.949997, 40.650002), 7000),
    ST_Area(ST_Buffer(ST_MakePoint(-73.949997, 40.650002),
   7000))
);

-- Filter trips by timestamp
DROP TABLE IF EXISTS filtered_table;
CREATE TABLE filtered_table AS (
  SELECT trip_id, atTime(trip, timestamp '2023-04-08 09:50:00
   ') AS subtrip FROM trips_mdb
);
```

```
20 DELETE FROM filtered_table WHERE subtrip IS null;
21 ALTER TABLE filtered_table ADD column shape geometry;
22 UPDATE filtered_table SET shape=subtrip::geometry;
23
24 -- Calculation of the total area covered by points without
   overlap
25 WITH buffered_geoms AS (
26     SELECT ST_Buffer(shape, 400) AS surface
27     FROM filtered_table
28     WHERE ST_Intersects(shape,
29         (SELECT geom FROM circle_geometry))
30 )
31 SELECT ST_Area(ST_Union(buffered_geoms.surface))
32     AS surface_totale,
33     ST_Area(ST_Union(buffered_geoms.surface))/circle_geometry
   .surface
34 FROM buffered_geoms, circle_geometry
35 GROUP BY circle_geometry.surface;
```

Listing 5.9: Compute audible area from realtime buses positions

The case can be extended to another application. Instead of checking to a given timestamp, we can also check into a timestamp range. Indeed, `atTime()` function has multiple signatures and also takes `tstzspan` type. The query is therefore easily adjustable. Figure 5.18 shows the result if the audible area was computed during 2 complete minutes, between 09:49:00 and 09:51:00. With 2 minutes elapsed, the covered area goes from 64.91% to 80.53%. The real difference between MobilityDB and native PostGIS lies in data filtering and the use of the `atTime()` function. MobilityDB makes it easy to extract a subtrip from a complete trip. Where PostGIS, which has to rely on a table full of points, must be able to locate time t, and if the position doesn't exist at time t, must at least be able to retrieve the last recorded position, or calculate an estimate of the position on the complete trip, which MobilityDB itself already does. MobilityDB is not only easy to write, but also easy to compute, thanks to its `atTime()` function, optimized for extraction at one instant or over one period, and its `tgeompoint` converters for simple geometries. We also see that the use of MobilityDB is interesting for moving objects, but that it is then up to PostGIS to perform operations linked to geometries, such as calculating areas, intersecting geometries, etc.

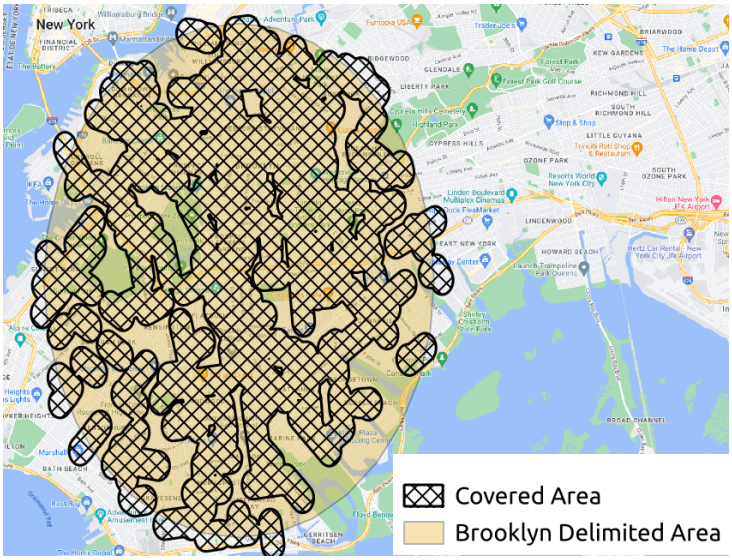


Figure 5.18: Audible area of Brooklyn on April 8, 2023, between 09:49:00 and 09:51:00

5.2.9 City Public Transport Trafic Visualisation

With these ways of importing and displaying GTFS Static and Realtime data, it would be interesting to display all the public transport services passing through a city on a map. In this use case, we are going to visualise the different public transport services available in Brussels, the STIB-MIVB, the Flemish transport services, De Lijn, and the Walloon services, TEC, and finally the SNCB, the national rail transport company. There are various ways of importing this grouped information, but here we will use a database for each of the data sources.

Figure 5.19 shows the theoretical GTFS data for 21 July 2023 at 17:53 for the four transport companies mentioned. The different coloured dots represent the companies' means of transport. STIB-MIVB, the main transport company in Brussels, is represented by the blue dots, De Lijn in yellow, TEC in red, and SNCB in green.

Unsurprisingly, the blue dots representing STIB-MIVB are dominant. Given the position of Brussels, which is surrounded by Flanders, De Lijn buses are relatively present on all sides of the city, but mainly in the north. As Wallonia is located to the south of Brussels, TEC buses are very few and far between. Finally, SNCB trains are spread throughout the city on the

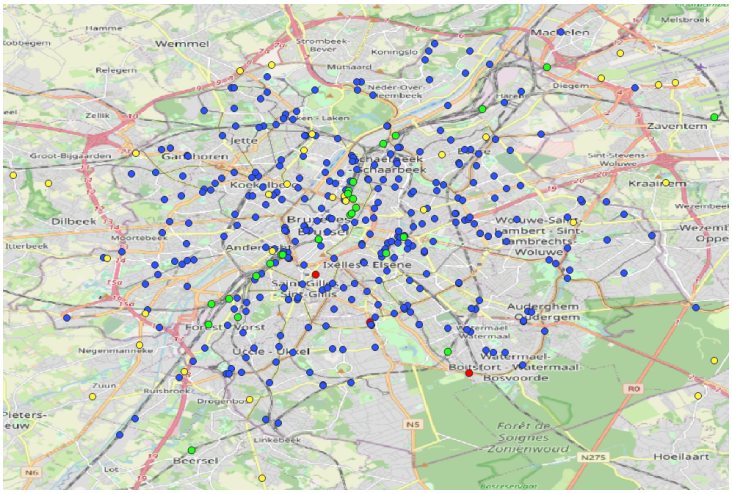


Figure 5.19: Theoretical presence of public transports in Brussels

railway lines.

Chapter 6

Conclusion and Future Work

6.1 Discussion

6.1.1 Import

This section will focus on evaluating the success or failure of dataset import operations. We have carried out an in-depth analysis of the performance of these imports.

We have foremost shown that we can import any public transport feed regarding some conditions, the presence of positions and their corresponding timestamps, but also the presence of a geometry describing the trajectory for an accurate matching. These conditions can lead us to the elaboration of the table `trip_positions`, allowing us to begin the transformation to MobilityDB.

In the context of GTFS Static, our observations reveal a successful implementation of the import functions. Data is integrated accurately and efficiently into MobilityDB, testifying to the robustness of our procedures. Indeed, the integration of GTFS Static data runs smoothly, reflecting the maturity and reliability of the conversion and pre-processing tools we have implemented.

However, importing NeTEx data requires a more nuanced approach. At this stage, we need to introduce an additional conversion step to transform NeTEx data into a GTFS-compatible format. Although this adds complexity to the import chain, the conversion broadens the scope of our integration process by including NeTEx data sources. This strategic adaptation reinforces the versatility of our system and extends its application potential. Neverthe-

less, we have shown that it was possible to import NeTEx feeds despite the fact it was not on a nordic specification. This requires another parsing step.

Regarding realtime feeds, GTFS Realtime and SIRI, the designed servers work relatively well and store received inputs correctly. Go seems to be a very good language for handling large quantities of data. Once the information is in the database, we can start the process of converting it into MobilityDB trips. By coupling this information with the Static feeds, in particular the trajectory identifiers, we can build the perfect trajectory thanks to its geometric representation.

In reviewing the current pre-processing steps, it is important to note that we have designed our process to successfully accommodate any static GTFS dataset in MobilityDB. This ability to accommodate diverse and varied data underlines the flexibility of our approach and MobilityDB's ability to adapt to the specifics of different datasets.

In addition, it should be noted that MobilityDB is distinguished by its compactness in terms of data storage. To evaluate MobilityDB's performance in more detail, we set up a final test to measure the size of the data in the tables. To do this, we use an SQL query, as illustrated in Listing 6.1.

```
1 SELECT table_name,
2 pg_size_pretty(pg_total_relation_size(table_schema || '.' ||
    table_name)) As total_size
3 FROM information_schema.tables WHERE table_name IN ('
    trips_input', 'trips_mdb')
```

Listing 6.1: Compute size of tables in database

Comparative table size analysis between GTFS Static and MobilityDB datasets reveals some enlightening findings. The results, presented in Table refcomparisonsize, demonstrate that MobilityDB excels at storing journeys efficiently and compactly. This efficiency is reflected in a significant reduction in the size of the tables compared to the input datasets. On average, tables using MobilityDB data types take up more than five times less space than input tables. However, it is important to note that this reduction can vary depending on the specific nature of the dataset, the number of trips, and the average size of each trip.

The space savings achieved by MobilityDB are having a significant impact, particularly on large transport systems such as the Long Island Rail Road and Metro de Madrid. In these cases, MobilityDB can compress data to ten to fifteen times its original size. This optimisation of storage space contributes to improved overall performance, as it reduces the processing

GTFS Feed	trips_input	trips_mdb	#trips_input	#trips_mdb
SNCB	48Mb	15Mb	229657	871
De Lijn	92 Mb	31 Mb	648230	1512
LIRR	4373 Mb	466 Mb	44626600	42962
NY Bus	320 Mb	121 Mb	2658620	10752
Kobe Subway	22 Mb	9.5Mb	129769	617
Metro de Madrid	148 Mb	10Mb	811748	420
Oslo (NeTEx)	480 Mb	74 Mb	2151864	10260

Table 6.1: Size and line count of spatiotemporal tables

load and facilitates access to data.

In addition, we find that the consolidation of data by path, illustrated by the presence of a single row per path in the `trips_mdb` table, results in a significant reduction in the total number of rows compared to the `trips_input` table. This structural simplification improves the efficiency of calculations and subsequent operations, reinforcing MobilityDB's performance advantage.

In summary, this detailed evaluation of data import operations highlights the robustness of MobilityDB in processing and managing spatiotemporal datasets. The results attest to the effectiveness of our approach and the relevance of MobilityDB as a storage and analysis tool for mobility data.

6.1.2 Use Cases

Through the exploration of these different use cases, we were able to unveil the polyvalence and power of MobilityDB in concrete applications. The results highlighted a series of key functionalities that confirm MobilityDB's role as an essential tool for the manipulation and analysis of temporal geospatial data in the context of mobility standards.

The first of the use cases presented is MobilityDB's ability to quickly calculate the nearest line to a point of interest. This functionality is of significant importance in real-life scenarios such as urban navigation or public transport planning. The efficiency of this calculation using MobilityDB demonstrates its ability to provide fast and accurate solutions to location problems.

Another crucial dimension explored is the measurement of vehicle speeds. Thanks to MobilityDB, this operation is proving to be not only feasible, but also highly efficient. MobilityDB's ability to manage temporal geospatial data enables significant results to be obtained in terms of speed estimation,

opening the way to more in-depth analyses of traffic flow and travel patterns.

One of MobilityDB's major strengths is its ability to combine static and real-time information to assess delays. This ability to merge data of different kinds opens up new prospects for real-time analysis and decision-making in the field of mobility. The concrete example of delay assessment demonstrates how MobilityDB can serve as a platform for the harmonious integration of heterogeneous data, improving the overall understanding of transport system performance.

Dynamic visualisation of data flows is another key feature of MobilityDB. The ability to graphically display temporal and geospatial information in realtime provides an interactive interface for monitoring and interpreting travel patterns. This functionality is invaluable to transport managers, urban planners and other stakeholders who require a realtime understanding of travel dynamics.

An innovative aspect of the use cases is the ability to export data of type `tgeompoint` to a GTFS directory. This feature further broadens MobilityDB's horizons as a data integration tool. The ability to convert temporal and geospatial data into a standardized format such as GTFS facilitates data exchange and enhances interoperability between different platforms.

Comparative results between MobilityDB and PostGIS/PostgreSQL, as illustrated in Table 6.2, confirm MobilityDB's undeniable performance advantages. The speed of query execution, as well as the overall efficiency of operations, position MobilityDB as an interesting choice for processing complex temporal geospatial data. The analysis clearly shows how MobilityDB outperforms traditional solutions in terms of speed and efficiency.

Use case	MobilityDB	PostGIS/PostgreSQL
1	1.22	2.091
2	2.401	23.476
3	2.408	11.339

Table 6.2: Queries average executions times in seconds

In addition to its high performance, MobilityDB also shines in terms of ease of writing. The manipulation of temporal geospatial data is greatly simplified thanks to specific functions such as `atTime()` and `twavg()`. This intuitive syntax enables complex spatiotemporal operations to be expressed concisely and elegantly. Compared to native solutions such as PostGIS, Mo-

bilityDB clearly emerges as a more user-friendly choice for writing sophisticated queries.

In summary, the in-depth study of the use cases highlighted the significant impact and relevance of MobilityDB in the context of mobility standards. Practical applications confirmed MobilityDB's ability to meet the complex challenges of handling and analyzing temporal geospatial data. As a robust, high-performance and user-friendly tool, MobilityDB has established itself as the ally of choice for advanced spatiotemporal modeling and analysis in various mobility-related fields.

6.2 Future Work

The present study has opened up exciting new perspectives for the optimal management and exploitation of spatiotemporal public transport data. However, there are still several areas to be explored and deepened to further enrich research in this field and maximize the impact of the results obtained.

- **Improved import and export methods :** Although methods for importing public transport data have been successfully developed and tested, ongoing improvements are needed to ensure greater flexibility and adaptability. For example, further exploration of the different variants of data formats such as NeTEx (Nordic, French, Italian,...) could ensure smooth and accurate import from a wider range of data sources. In addition, an export method for GTFS has been detailed, but it would be interesting to explore a method for other standards such as NeTEx. An XML export method should therefore be considered.

- **Implicit real-time position acquisition :** Exploring SIRI and GTFS Realtime messages and using them for further analysis of real-time transport data could bring greater precision and a finer understanding of mobility dynamics. Exploiting other fields in real-time messages, such as current status and stop identifier, could provide additional information for analysing transport system performance.

- **Improved dashboarding :** Grafana was a great help during this study. However, we are limited in the ways we can visualize the information. It would be interesting to see if it is possible to visualize

our moving objects on one of Grafana's existing plugins, or to use another dashboarding tool (Apache Superset, Metabase, ...). This would make it possible to centralize information on a single medium instead of viewing graphics on one medium and viewing a map on another.

In short, the perspectives for the future are vast, and encourage continued exploration of the potential offered by spatiotemporal public transport data. By deepening import methods, extending application to other mobility domains and further exploiting MobilityDB's capabilities, it is possible to contribute to better management of transport systems and an enhanced user experience. These future developments will be part of the global drive to continuously improve urban mobility.

6.3 Conclusion

In this work, we took the time to understand the issues related to mobility. As a result, one of the most primary solutions that emerged was the sharing and analysis of public transportation company data. We have seen that some existing tools [33, 32] could handle this kind of data, but were quickly limited by a lack of advanced analysis techniques. That is why we chose MobilityDB, an open-source Moving Object Database Management System, as the technology that would allow us to import and process public transportation data. The different most used mobility standards were explored: GTFS static, GTFS Realtime, NeTEx and SIRI seem to be promising candidates. These are the most widely adopted standards, and many studies are derived from data extracted from GTFS. NeTEx is still a good competitor, but more research is needed to establish `tgeompoints` from the source, especially for non-nordic implementation and for SIRI.

Next, we established means to import data from these two standards into a format that can be interpreted by MobilityDB. The raw data was inserted into a PostgreSQL database. Using the positions and timestamps extracted from the raw data, we were finally able to build MobilityDB trips of type `tgeompoint`. This resulted in a function in PostgreSQL that allows to import any static GTFS feed, as well as a minimal code in Golang that allows to catch a GTFS Realtime feed and to import it into MobilityDB. Golang was selected for its efficiency, its intelligent management of large amounts of data, and its speed of execution compared to other widely used languages.

The quality of the feeds is of paramount importance, so we had to find ways of formatting the data in case their formatting contained errors. The means of importing also involved pre-processing steps, which enabled the GTFS files to be reformatted for import into SQL, error handling, but also the generation of missing data such as vehicle trajectories. We also explored the possibility of converting NeTEx format into GTFS format for NeTEx files that comply with the format defined by Entur.

The tests of these import methods are very conclusive, although the success of the import depends on some conditions. Indeed, for GTFS static, it is necessary that the file `shapes.txt` is present and filled correctly in order to establish the geometries of the transport lines. And we have found some ways to generate it when it is missing. For GTFS Realtime, we only explored the avenue where the vehicle positions are present in the `VehiclePosition` message. Otherwise, as for the subways that are doing indoor, we might want to explore the avenue of using the other fields in the message, such as `StopId` and `CurrentStatus` that allow us to know at which stop the subway is, and whether it is moving or not. `TripUpdates` message can also be used to estimate the positions of vehicles.

A NeTEx pipeline can be explored, where the Part 1 and Part 2 of the specification could be parsed and extract all the datas. Meanwhile we have found a way to convert NeTEx feeds that respect nordic specification into a GTFS feed, allowing us to import the data in MobilityDB. This could also allow us more freedom for SIRI, to match its data to a real NeTEx feed. As we have seen that SIRI is easily importable to MobilityDB.

Once we tested the import methods, we used MobilityDB to analyze the imported data in different use cases on buses and trains in New York City. Traffic can be analyzed using MobilityDB's built-in functions. Speed, delay, and position of vehicles can be calculated easily. Road sections containing delays can be calculated easily as well and visualised as a heatmap. Other use cases have also been shown such as the audible coverage area. Moving transports can be observed with MOVE, a QGIS extension for MobilityDB. This resulted to also allow us to visualise the traffic in a city, as we did for Brussels, but also to visualise the delay by plotting two moving objects, static and realtime, on the same layer.

In conclusion, the exploitation of spatiotemporal public transport data is a key element in improving mobility. Mobility standards such as GTFS, GTFS-realtime, NeTEx, SIRI, etc. play a crucial role in facilitating data sharing between players in the sector. The use of MobilityDB as a tool for

managing and analysing this data offers significant advantages in terms of structuring, efficient processing and exploitation of the information.

This thesis has highlighted the importance of mobility standards in the context of public transport and demonstrated the relevance of using MobilityDB as a suitable DBMS. The results open up new prospects for optimising mobility services, transport planning and improving the user experience. Future research opportunities remain to further explore the functionalities of MobilityDB and to extend its use to other mobility domains.

In short, this thesis contributes to a better understanding of the issues involved in managing spatiotemporal public transport data and highlights the importance of using specialised tools such as MobilityDB for optimal exploitation of this data. These advances form a solid basis for the continuous improvement of mobility and for the implementation of public transport services that are more efficient, effective and adapted to users' needs.

Bibliography

[1] R. H. Güting and M. Schneider, *Moving objects databases*. Elsevier, 2005.

[2] E. Zimányi, M. Sakr, and A. Lesuisse, "MobilityDB: A mobility database based on PostgreSQL and PostGIS," *ACM Trans. Database Syst.*, vol. 45, Dec. 2020.

[3] M. Bakli, M. Sakr, and E. Zimanyi, "Distributed moving object data management in MobilityDB," in *Proceedings of the 8th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pp. 1–10, 2019.

[4] G. Rovinelli, S. Matwin, F. Pranovi, E. Russo, C. Silvestri, M. Simeoni, and A. Raffaetà, "Multiple aspect trajectories: a case study on fishing vessels in the Northern Adriatic sea.," in *Proceedings of the Workshops of the EDBT/ICDT 2021 Joint Conference*, 2021.

[5] S. Wu, E. Zimányi, M. Sakr, and K. Torp, "Semantic segmentation of AIS trajectories for detecting complete fishing activities," in *Proceedings of the 23rd IEEE International Conference on Mobile Data Management*, pp. 419–424, IEEE, 2022.

[6] M. Buchanan, *The benefits of public transport*. PhD thesis, Nature Publishing Group, 2019.

[7] M. Fadaei and O. Cats, "Evaluating the impacts and benefits of public transport design and operational measures," *Transport Policy*, vol. 48, pp. 105–116, 2016.

[8] A. Antrim, S. J. Barbeau, *et al.*, "The many uses of GTFS data–opening the door to transit and multimodal applications," *Location-Aware In-*

formation Systems Laboratory at the University of South Florida, vol. 4, 2013.

[9] L. Ge, M. Sarhani, S. Voß, and L. Xie, "Review of Transit Data Sources: Potentials, Challenges and Complementarity," *Sustainability*, vol. 13, no. 20, p. 11450, 2021.

[10] M. Roth, "How Google and Portland's TriMet Set the Standard for Open Transit Data," *SF. Streetsblog. org*, vol. 5, 2010.

[11] G. Inc., "GTFS reference." `https://developers.google.com/transit/gtfs/reference`.

[12] S. Goliszek and M. Połom, "The use of general transit feed specification (GTFS) application to identify deviations in the operation of public transport at morning peak hours on the example of Szczecin," *Europa XXI*, vol. 31, pp. 51–60, 2016.

[13] K. Kaeoruean, S. Phithakkitnukoon, M. G. Demissie, L. Kattan, and C. Ratti, "Analysis of demand–supply gaps in public transit systems based on census and GTFS data: a case study of Calgary, Canada," *Public Transport*, vol. 12, pp. 483–516, 2020.

[14] E. Chondrodima, H. Georgiou, N. Pelekis, and Y. Theodoridis, "Public Transport Arrival Time Prediction Based on GTFS Data," in *Proceedings of the 7th International Conference on Machine Learning, Optimization, and Data Science*, pp. 481–495, Springer, 2022.

[15] E. Chondrodima, H. Georgiou, N. Pelekis, and Y. Theodoridis, "Particle swarm optimization and RBF neural networks for public transport arrival time prediction using GTFS data," *International Journal of Information Management Data Insights*, vol. 2, no. 2, p. 100086, 2022.

[16] C. Bannur, C. Bhat, G. Goutham, and H. Mamatha, "Traffic Congestion Prediction Based on Spatio-Temporal Graph Structure Learning," in *Proceedings of the International Conference on Intelligent and Innovative Technologies in Computing, Electrical and Electronics*, pp. 264–269, IEEE, 2023.

[17] C. Bannur, C. Bhat, G. Goutham, and H. Mamatha, "General Transit Feed Specification Assisted Effective Traffic Congestion Prediction Using

Decision Trees and Recurrent Neural Networks," in *Proceedings of the 1st International Conference on Data, Decision and Systems*, pp. 1–6, IEEE, 2022.

[18] S. A. H. Zahabi, A. Ajzachi, and Z. Patterson, "Transit trip itinerary inference with GTFS and smartphone data," *Transportation Research Record*, vol. 2652, no. 1, pp. 59–69, 2017.

[19] G. Inc., "GTFS realtime reference." `https://developers.google.com/transit/gtfs-realtime/reference`.

[20] A. Nishino, A. Kodaka, M. Nakajima, and N. Kohtake, "A Model for Calculating the Spatial Coverage of Audible Disaster Warnings Using GTFS Realtime Data," *Sustainability*, vol. 13, no. 23, p. 13471, 2021.

[21] Z. Aemmer, A. Ranjbari, and D. MacKenzie, "Measurement and classification of transit delays using GTFS-RT data," *Public Transport*, vol. 14, no. 2, pp. 263–285, 2022.

[22] S. Cortés Fernández *et al.*, "Detección automática de desvíos en rutas del sistema de transporte público mediante el uso de especificaciones GTFS y GTFS Realtime," 2019.

[23] A. R. M. Queiroz, V. B. Santos, D. C. Nascimento, and C. E. S. Pires, "Conformity analysis of GTFS routes and bus trajectories," in *Proceedings of the 34th Brazilian Symposium on Databases*, pp. 199–204, SBC, 2019.

[24] F. Effendy, B. Adhilaksono, *et al.*, "Performance Comparison of Web Backend and Database: A Case Study of Node. JS, Golang and MySQL, Mongo DB," *Recent Advances in Computer Science and Communications (Formerly: Recent Patents on Computer Science)*, vol. 14, no. 6, pp. 1955–1961, 2021.

[25] R. Cox, R. Griesemer, R. Pike, I. L. Taylor, and K. Thompson, "The Go programming language and environment," *Communications of the ACM*, vol. 65, no. 5, pp. 70–78, 2022.

[26] C. Keller, S. Brunk, and T. Schlegel, "Introducing the public transport domain to the web of data," in *Proceedings of the 15th International Conference on Web Information Systems Engineering*, pp. 521–530, Springer, 2014.

[27] L. Filina-Dawidowicz, A. Cernova-Bickova, I. Semenov, D. Możdrzeń, A. Wiktorowska-Jasik, and D. Bickovs, "Information support of cargo ferry transport: case study of Latvia," *Procedia Computer Science*, vol. 176, pp. 2192–2201, 2020.

[28] M. Braga, M. Y. Santos, and A. Moreira, "Integrating public transportation data: creation and editing of GTFS data," in *Proceedings of the New Perspectives in Information Systems and Technologies, Volume 2*, pp. 53–62, Springer, 2014.

[29] S. J. Barbeau, "Quality control-lessons learned from the deployment and evaluation of GTFS-realtime feeds," in *Proceedings of the 97th Annual Meeting of the Transportation Research Board*, 2018.

[30] P. Brosi, "A toolchain for generating transit maps from schedule data," in *Proceedings of Schematic Mapping Workshop*, vol. 10, 2022.

[31] H. Bast and P. Brosi, "Sparse map-matching in public transit networks with turn restrictions," in *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pp. 480–483, 2018.

[32] T. Zhang, M.-H. Chen, and C. Lawson, "General transit feed specification data visualization," in *Proceedings of the 22nd International Conference on Geoinformatics*, pp. 1–6, IEEE, 2014.

[33] N. Kunama, M. Worapan, S. Phithakkitnukoon, and M. Demissie, "GTFS-VIZ: Tool for preprocessing and visualizing GTFS data," in *Proceedings of the 2017 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2017 ACM International Symposium on Wearable Computers*, pp. 388–396, 2017.

[34] A. A. Pertence, R. A. Mini, and H. T. Marques-Neto, "Vulnerability Analysis of the Urban Transport System in the Context of Smart Cities," in *Proceedings of the IEEE International Smart Cities Conference (ISC2)*, pp. 1–8, IEEE, 2020.

[35] J. Godfrid, P. Radnic, A. Vaisman, and E. Zimányi, "Analyzing public transport in the city of Buenos Aires with MobilityDB," *Public Transport*, pp. 1–35, 2022.