Instituto Tecnológico de Buenos Aires

An Analysis of Public Transport in the City of Buenos Aires with MobilityDB

School of Engineering & Management

Presented to obtain the degree:
Informatics Engineer

Authors
Juan Godfrid - 56609
Pablo Radnic - 57013

Tutors
Alejandro Ariel Vaisman
Esteban Zimanyi

January 25th, 2020
Buenos Aires, Argentina

Abstract

The General Transit Feed Specification (GTFS) is a data format used to express public transportation schedules and associated geographic information created in 2006 by Google. It is now widely used all around the world in both of its versions, GTFS Real-Time and GTFS Static. So far, tools to analyze these data are not widely available in the open source community. Companies that need to perform in-depth analysis must create proprietary solutions which the public has no access to. *MobilityDB* is a free and open source PostgreSQL and PostGIS extension that adds spatial and temporal datatypes along with convenient functions, to facilitate the analysis of mobility data. However, loading GTFS data into *MobilityDB* must still be done in an ad-hoc fashion. Furthermore, obtaining GTFS real-time data is an even more involved problem. This work describes how MobilityDB is used to analyze public transport mobility in the city of Buenos Aires. To this end, static and real-time GTFS data for the Buenos Aires public transportation system were acquired, imported into *MobilityDB*, and analyzed. Visualizations were also produced to enhance the analysis.

Contents

1 Introduction

The ubiquity of GPS tracking devices and Internet of Thing (IoT) technologies has resulted in a collection of massive amounts of data that describe the temporal evolution of such objects. This data abundance has made analytics tools such as specialized databases become of high importance to the industry. Working solutions exist for querying spatial and temporal evolution of objects, for example in Google Maps [4], to find the best route between multiple points of interest, estimate the arrival time, and even predict traffic of a certain route in a certain point in the future. However, these solutions are proprietary and not available to the general public. Google Maps contains a series of APIs [5] that can provide directions for transit, biking, driving, or walking between multiple locations, calculate travel times and distances, and determine the exact roads that a certain vehicle is traveling. Unfortunately, these functionalities are limited since they do not provide a querying language in which new use cases could be developed, and the evolution of these are tied to Google's interest in developing them. *MobilityDB* is an open-source PostgreSQL and PostGIS extension designed to tackle this problem, providing a set of functions and spatio-temporal datatypes that, together with PostGIS's functionalities, create a flexible tool set to query spatio-temporal objects.

The General Transit Feed Specification (GTFS) is a common format used to outline public transportation schedules and real-time data with associated geographic information. GTFS defines two standards, *GTFS Static*, which is used to define schedules, and *GTFS Realtime* which is used to communicate live positions of vehicles. This work shows how GTFS data can be used to analyze the public transportation system of Buenos Aires with *MobilityDB*. It describes the software used to acquire, import, and load the data, and then query and visualize the results. The solutions described in this work cannot be generalized to importing any GTFS dataset to *MobilityDB* because of the wide spectrum of variations that the GTFS standard allows. However, many datasets of GTFS transit information may share the same difficulties importing the data with the datasets used; therefore this work can serve as a guide on what to expect and what could be done to use a certain dataset with *MobilityDB*.
The goals of this work are:

- Obtaining GTFS-Static and GTFS-Realtime data from the City of Buenos Aires.
- Importing the data of both GTFS standards into *MobilityDB*.
- Querying the data using *MobilityDB's* functionality for traffic analysis.

2 State of the art

This section studies the state of the art, the techniques and technologies used. First, the basic terms will be defined, then **GTFS-Static** and **GTFS-Realtime** will be explained in detail. This will be followed by an overview and comparison of **MobilityDB** and **SECONDO**, another well known Moving Object Database.

2.1 Preliminary Definitions

Definition 1 (Spatio-Temporal Objects) *A **Spatio-Temporal Object** is a uniquely identifiable entity that has a value in space and time for as long as it is active.*

Definition 2 (Moving Object) ***Moving objects** are bodies (e.g., cars, trucks, pedestrians) whose spatial features change continuously in time. Spatio-temporal Data can be extracted from **Moving objects** by attaching devices which produce GPS signals such as smart phones.*

Definition 3 (Moving Object Database) *A **Moving Object Database** is a database with capabilities to store and query **Spatio-Temporal Objects***

Definition 4 (Route) *A **route**, is a certain spatial trajectory that a moving object can take, without a specific date or time associated to it.*

Definition 5 (Trip) *A **trip** is a route repeatedly traversed at a certain time.*

Definition 6 (Trajectory) *A **trajectory**, is the continuous line that is drawn by a spatio-temporal object during it's active time.*

Definition 7 (SRID) *A Spatial Reference System Identifier (**SRID**) is a unique value used to unambiguously identify projected, unprojected, and local spatial coordinate system definitions. These coordinate systems are at the heart of all GIS applications. [24]*

2.2 The GTFS Specification

The General Transit Feed Specification (GTFS) is a data format used to define public transportation schedules and realtime data with associated geographic information. GTFS started out as a side project of a Google employee named Chris Harrelson in 2005. Chris was acquainted with IT managers from the transit agency at Portland, Oregon, who provided Google CSV exports of the agencies schedule data. That was the beginning of Google's "Transit Trip Planner", which expanded to other US cities the following year, and the format used was released as *Google Transit Feed*

Specification. The specification became widely used since there was no public standard for transit timetables and due to its convenience and flexible format. As a consequence of its wide adoption, the data standard was renamed to *General Transit Feed Specification* in 2009.

The GTFS specification has two versions, Static and Realtime, the former being the most widely used. GTFS-Static is used to predefine trip schedules, while GTFS-Realtime is a feed of real time data with the positions and timestamps of the data points within trips and routes.

2.2.1 GTFS-Static

A GTFS-Static feed is composed of a series of text files with a CSV format that are stored in a ZIP file. Each file determines a specific aspect about the public transportation schedules, such as stops, routes and trips. [11] GTFS-Static contains the following files: [12]

- **agency.txt**
  This is a required file. It lists the different transit agencies that will operate the transport routes. These have an agency identification number, name, a timezone and contact information, such as email, website and phone number.

- **stops.txt**
  This is a required file. It lists the stops that will be used on the scheduled trips. It contains a stop code that identifies the location for riders, a stop name, latitude and longitude coordinates. There are several other optional fields, such as whether the stop is accessible to wheelchair boarding.

- **routes.txt**
  This is also a required file. It lists the routes that are contained in the public transport schedule. It contains the identification number of the agency that operates the route, a readable route short name, a long name and a route type which is a number defined by the GTFS standard. The number **0** represents a tram, streetcar or light rail, **1** represents a subway or metro, etc. There is a total of 12 different route types. There are also several other optional fields, such as a route color for displaying the route graphically, and a route url with a link to a webpage about that particular route.

- **trips.txt**
  This is a required file. It lists the trips contained in the schedule. The trips have an identification number, a route identification number and a service identification number. The service contains a set of dates for the trips.

- **stop_times.txt**
  This is a required file. It links trips with stops and adds a date and time with arrival time and departure time fields. It also contains several optional fields such as pickup and dropoff types (regular pickups, phone arranged pickups, driver coordinated pickups, etc.)

- **calendar.txt and calendar_dates.txt**
  These files are conditionally required, either **calendar.txt** or **calendar_dates.txt** must exist, both can be present too. **calendar.txt** contains a service identification number, and a field for each day of the week, representing if the service is available that day. It also contains a start date and an end date for the service. **calendar_dates.txt** adds service exceptions. These service exceptions can be addition exceptions or removal exceptions. The fields are a service identification number, a date for the exception, and the exception type (addition or removal).

- **Optional files**
  There are several optional files, such as **fare_attributes.txt** that includes trip prices and payment methods, **fare_rules.txt**, **shapes.txt** that includes data points that determine the trajectory taken between stops, **frequencies.txt**, **transfers.txt**, **pathways.txt**, **levels.txt**, **feed_info.txt**, **translations.txt** and **attributions.txt**.

2.2.2 GTFS-Realtime

GTFS Realtime is defined in a looser manner. A GTFS Realtime feed is served via the HTTP protocol, and should provide frequent updates, although there are no constraints on how frequently these updates should be served nor on the exact manner in which the feed is updated or retrieved. Any web server can host and serve the data, and any transport protocols can be used as well. A GTFS Realtime feed can support the following types of information [7]:

- **Trip updates** - delays, cancellations, changed routes.
- **Service alerts** - events affecting a station, route or trip.
- **Vehicle positions** - information about the vehicles currently in service, with their locations and other data such as congestion level.

GTFS Realtime has two distinct feed elements, **messages** and **enums**. **Messages** represent complex types and **enums** represent a list of fixed values, these are generally used to communicate certain events. The feed elements are used when the web server communication method is the Protocol Buffer, an efficient protocol used to transmit a GTFS Realtime feed. However, as mentioned before, the communication method for GTFS realtime is not

strictly defined. For the case of the real time data feed for the city of Buenos Aires, the API implements the Protocol Buffer and also a JSON format body within an HTTP response. The API sends nested **messages** as fields in the JSON body. Only the messages that are used will be described: [8]

- **FeedEntity**
  This is the message that is sent on all HTTP requests, it provides an update of an entity in the transit feed. It contains an identification field, a **TripUpdate** message, a **VehiclePosition** message and an **Alert** message. However, the **Alert** message is conditionally required and not implemented in this case.

- **TripUpdate**
  This message provides an update on the progress of a vehicle along a trip. It contains a trip descriptor, a vehicle descriptor and fields to represent the delay and new stop time that is being alerted.

- **VehiclePosition**
  This message provides real time position information of a given vehicle. It contains a trip descriptor, vehicle descriptor, a position described in latitude and longitude coordinates, a stop identificator of the current stop and a timestamp in **POSIX** time.

2.3 MobilityDB

MobilityDB [18] is a database management system for moving object geospatial trajectories, such as GPS traces.

Built on top of PostgreSQL and its Geo extension PostGIS; *MobilityDB* adds support for temporal and spatio-temporal objects to the open source Database environment.

The long established research in moving object databases has until now resulted in research prototypes such as *SECONDO* [23] and *HERMES* [15]. MobilityDB is envisioned to become the first open-source, industry-scale implementation that can be used in real-world applications.

2.3.1 Type System

2.3.1.1 Temporal Types

Temporal types is the *MobilityDB* solution for handling objects whose value changes over time. Representing values that evolve in time is essential in many applications; examples of this include stock prices, temperature and of course Spatio-Temporal Objects.

MobilityDB provides the following temporal types: **tbool**, **tint**, **tfloat**, **ttext**, **tgeompoint**, and **tgeogpoint**.

These temporal types are based, respectively, on the **bool**, **int**, **float**, and **text** base types provided by *PostgreSQL*, and on the **geometry** and **geography** base types provided by *PostGIS*.

Temporal types are initially constructed from a discrete set of values. The object represents the evolution of the value during a sequence of time instants, the values between these instants are interpolated using either a stepwise or a linear function according to the type.

2.3.1.2 Time Types

All temporal types are based on four time types: the **timestamptz** type provided by *PostgreSQL* and three new types which are **period**, **timestampset**, and **periodset**.

The **period** type is more efficient implementation of the **tstzrange** type provided by *PostgreSQL*. The most relevant differences are its fixed length, and disallowing empty periods while the **tstzrange** type is of variable length and allows any value.

The **timestampset** is a collection of one or more **timestamptz** values whilst the **periodset** is a non-empty collection of ordered and non-overlapping, **period** values.

2.3.1.3 Range Types

Two Range types were also introduced in the system. Namely **intrange** and **floatrange**. These ranges are pairs of upper and lower bounds which can be interacted with out of the box operators such as < @ (Contained in) and others.

2.3.2 MobilityDB functions

The type system defines only the building blocks of *MobilityDB's* functionality. The *MobilityDB* documentation [18] goes into great detail on the length of functionality implemented on the platform. The functions used in the making of this analysis can be separated in the following categories:

- **Functions and Operators for Time Types and Range Types**
  Perform different operations on Time types and Ranges. Generally polymorphic functions that receive any type and perform different calculations. Ex:

```
1 startTimestamp({timestampset, periodset}): timestamptz
2 endTimestamp({timestampset, periodset}): timestamptz
3 timespan({timestampset, period, periodset}): interval
```

- **Functions and Operators for Temporal Types**
  Perform different operations on Temporal types. These operations can

mostly be thought of as applying the traditional operations at each instant, which yields a temporal value as result. E.g:

```
cumulativeLength(tpoint): tfloatseq
speed(tpoint): tfloats
nearestApproachDistance({geo, tpoint}, {geo, tpoint}): float
```

2.3.3 Spatio-Temporal Object Implementation

Figure 1: Mobility DB Trajectory phases
Original (top), Interpolation (middle), Normalization (bottom)

A *Spatio-Temporal Object* is generally built from a series of discrete timestamped location points. However moving object's *trajectories* are continous; so in order to properly query the state of the object *MobilityDB* interpolates the points with a linear function, generating a continuous approximation. This interpolation allows estimative queries for the state of the object at any instant in the reported interval.

The process of *trajectory* creation also allows for redundant data points to be identified and removed from the dataset. This process is called Normalization and in practice significantly reduces the trajectory's storage size.

MobilityDB provides two different *Spatio-Temporal Object* types, Temporal **Geometry** Point and Temporal **Geography** Point, these correspond to *PostGIS'* data types. The difference between the two is the reference system, **geography** points use a geodesic reference system and offset accuracy for complexity whilst **geometry** points use a cartesian reference system and allow calculation of speed and other distance related metrics.

Once the object's evolution is stored in the system, MobilityDB allows multiple functions to access and manage its values. Some of these functions such as Speed and Direction return temporal values themselves allowing the user to take full advantage of the database's type system.

2.3.4 Example Queries

```
-- Temporal addition
SELECT tint '[1@2001-01-01, 1@2001-01-03)' +
       tint '[2@2001-01-02, 2@2001-01-05)';
--"[3@2001-01-02, 3@2001-01-03)"
--Temporal intersects
SELECT tintersects(tgeompoint '[Point(0 1)@2001-01-01,
       Point(3 1)@2001-01-04)',
       geometry 'Polygon((1 0,1 2,2 2,2 0,1 0))');
--"{[f@2001-01-01, t@2001-01-02, t@2001-01-03],
-- (f@2001-01-03, f@2001-01-04]}"
```

The query above shows how *MobilityDB's* temporal types work. The first **SELECT** statement constructs two **tint**s and adds them. The resulting value is a **tint** as well.

The second example illustrates how different Temporal types can work together. In this case when querying **tintersects** for a temporal point and a given geometry the resulting value is a **tbool**.

The last query can be thought of as the times a car entered a static region. The car is the spatio temporal point, the region is described as a polygon, and the result is a list containing all the times a car was in or out of the region.

```
CREATE TABLE Trips(CarId integer, TripId integer,
                   Trip tgeompoint);
INSERT INTO Trips VALUES
(10, 1, tgeompoint '{[Point(0 0)@2012-01-01 08:00:00,
Point(2 0)@2012-01-01 08:10:00, Point(2 1)@2012-01-01
     08:15:00)}'),
(20, 1, tgeompoint '{[Point(0 0)@2012-01-01 08:05:00,
Point(1 1)@2012-01-01 08:10:00, Point(3 3)@2012-01-01
     08:20:00)}');
-- Value at a given timestamp
```

```
11 SELECT CarId, ST_AsText(valueAtTimestamp(Trip, timestamptz '2012-01-01 08:10:00'))
12 FROM Trips;
13 -- 10;"POINT(2 0)"
14 -- 20;"POINT(1 1)"
```

The script above illustrates a more practical example where values are stored in a table. The **SELECT** statement queries the value of the cars at a specific timestamp resulting in a point.

2.4 The SECONDO Moving Object Database

Among the many attempts at producing Moving Object Databases *SECONDO* is one of the few that is still being maintained to this day [17]. *SECONDO* is an extensible database system supporting non-standard applications. *SECONDO* provides an out of the box GUI which supports spatio-temporal data allowing stress-free visualization.

The Secondo system is extensible by algebra modules, using a well defined interface. New data models and data structures together with their operations are integrated into the system in this way. Some modules, such as the Spatial and Temporal algebras are included out of the box.

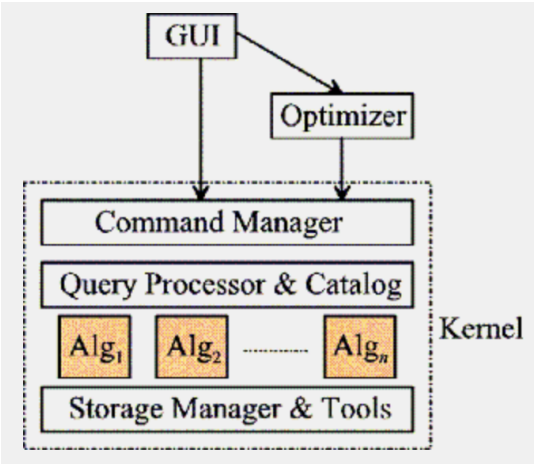

Figure 2: Secondo architecture

Though *SECONDO's* capabilities are virtually limitless, since it can always be expanded by new algebraic modules, it is mostly used for spatio-temporal or moving object analysis.

2.4.1 SECONDO queries

SECONDO queries can be formulated in two different ways: directly against *SECONDO*'s executable language or via a SQL-like language which

is translated by the query planner; however the latter is, in some cases, sub optimal and presents restrictions.

The following examples show how to query for streets whose name starts with A for both types of languages.

```
# Executable language
query streets feed filter[.Name starts "A"] project[Name] consume;
# SQL-like language
select Name from streets where [Name starts "A"]
```

2.4.2 Parallel SECONDO

SECONDO's capabilities are restricted by it's underlying hardware resources since the installation is done on a single computer. Parallel *SECONDO* intends to scale up the *SECONDO* from one computer to a cluster the workflow is controlled by the popular processing platform *Hadoop*.

2.5 MobilityDB vs SECONDO

2.5.1 Language Style

As mentioned, *SECONDO* queries can be formulated in a *SQL-like* format, or directly on *SECONDO*'s executable language. In this regard *MobilityDB* provides the advantage of being built on top *PostgreSQL* making the learning curve much easier for users already familiarized with SQL Databases.

2.5.2 BerlinMOD comparison

BerlinMOD is a benchmark for spatio-temporal databases. It is intended as a tool for comparing the performance of different implementations.

BerlinMOD primarily measures the performance on queries employing moving point data. Moving point data are sampled from simulated cars driving in the street network of the German capital Berlin in a representative way.

MobilityDB has been compared with *SECONDO* using this benchmark. In summary, *MobilityDB* came faster in 63% of the queries. The total run time of all queries in *MobilityDB* required 13% of the time required in *SECONDO*. [19] [20]

2.5.3 Decentralized Queries

With the addition of Parallel *SECONDO* queries on this database can be scaled horizontally as far nodes can be added to the cluster. In the case of *MobilityDB* clusterization is both benefited and limited by the capabilities of *PostgreSQL* and *PostGIS*.

2.5.4 Visualize query results

SECONDO can leverage its out-of-the-box GUI to visualize query results and input data. On the other hand, *MobilityDB* lacks this feature out of the box and has to be integrated with *QGIS* or other visualization applications.

2.6 Map Matching

Map Matching is the process of equating a series of discrete locations to a logical route in a network. Typically used to reduce inaccuracies from GPS signals, map matching algorithms take serial point locations and relate them to streets on a map.

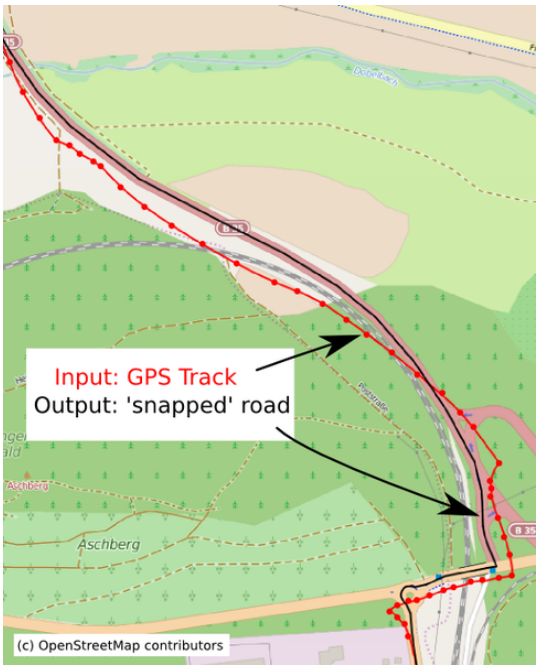

Figure 3: Map Matching

2.6.1 Map Matching algorithms

Map Matching algorithms can be separated into two categories

- **Real-Time Matching**
  Real time map matching compromises accuracy for increased speed and processing. Used to correct routes during the location sampling.

- **Offline Matching**
  Offline map matching compromises speed for increased accuracy. Used once all data is recorded, offline map matching can theoretically traverse the entire road network looking for the perfect match. However

in practice limits have to be established to reduce computational complexity.

2.6.2 Barefoot

One of the most broadly used map matching implementations is provided by **bmwcarit** by name of **Barefoot**. [1]
Barefoot is a Java map matching library with state-of-the-art online and offline map matching that can be used stand-alone or in the cloud.

3 Case Study

3.1 Introduction

In order to validate the effectiveness of *MobilityDB* in consuming GTFS data, studying the transport system of *Buenos Aires* was considered. The studied area includes the *City of Buenos Aires* and its outskirts; collectively known as the *Metropolitan Area of Buenos Aires* or **AMBA**.

The **AMBA** transport system consists of three main branches. The subway system, contained in the city itself, the metropolitan railway system, which connects *Buenos Aires* with other cities of the region such as *Zarate* and *La Plata* and finally the Bus system, which is an extremely complex system of hundreds of municipal and provincial bus lines from all across the urban area.

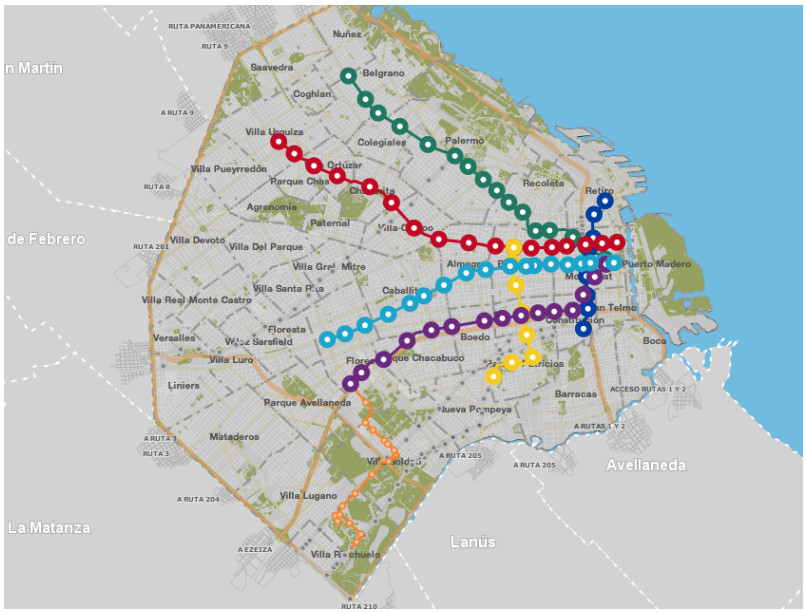

Figure 4: Buenos Aires Subway Network

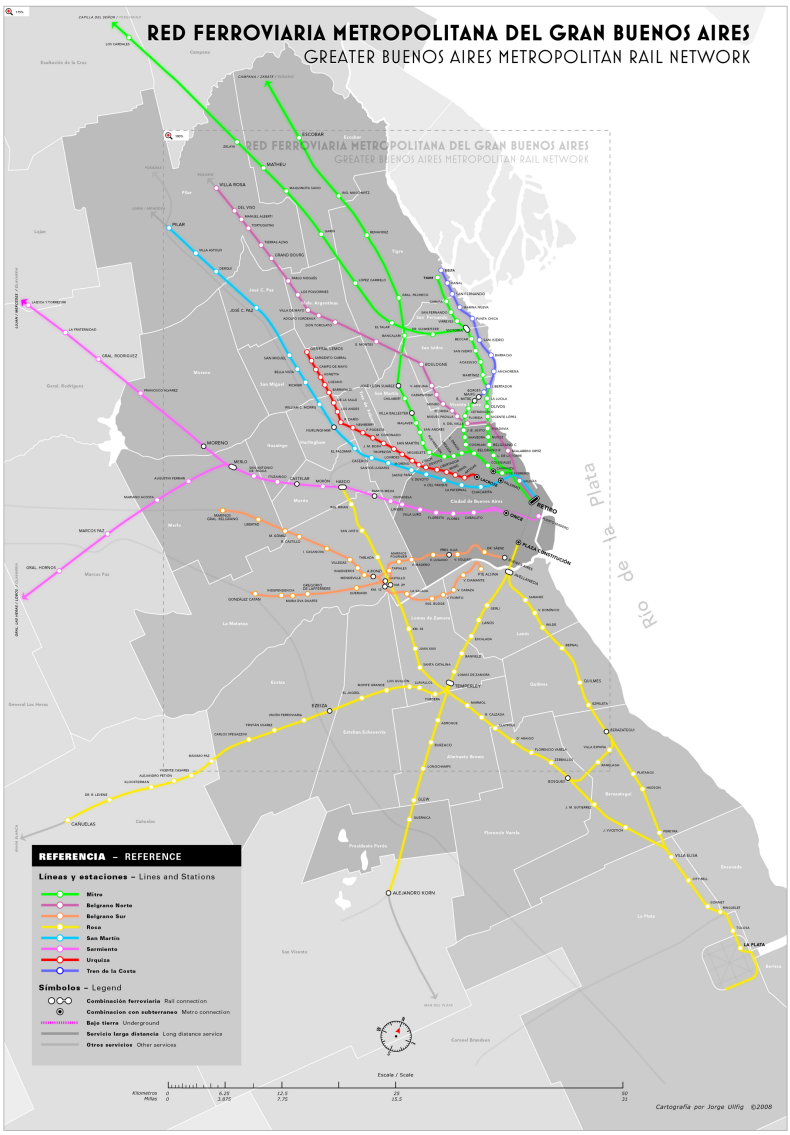

Figure 5: Metropolitan Railway Network

Chief amongst the multiple providers of information on the **AMBA** transport system is the *City of Buenos Aires* open data site. [2] This portal supplies both the itineraries for all branches of the **AMBA** transport system as well as real time information on the vehicles themselves.

Having access to this information allowed the leverage of *MobilityDB*'s capabilities to execute complex queries which would provide data on: Average speed depending on the hour or the day of the week, transports whose trajectory passes close to a point of interest, average delay for each transport and finally transport delay heat maps.

The following section will explore how the transport system data was obtained, the different systems used to import this data and finally the results obtained.

3.2 Desired Schema

In order to import spatio-temporal data into *MobilityDB*, it is necessary to be able to represent the given material with native MobilityDB spatial-temporal types. The desired output of the import process is to have an SQL table named **trips_mdb** with the following fields:

- **trip_id**
  An identification number that describes the particular trip.
- **vehicle_id**
  An identification number that describes the particular vehicle in service.
- **startdate**
  A date field that represents the starting date of the trip.
- **starttime**
  A time field that represents the starting time of the trip.
- **trip**
  This is the field that contains the time and location data of all of the trip. This is achieved by using MobilityDB's **tgeompoint** data type.
- **traj**
  This field represents the trajectory of the trip with only spatial data, no time data is included. This field uses the **geometry** data type included in PostGIS.

trip contains all of the trips spatial-temporal data, and it is the field used in the majority of the queries, since all of the MobilityDB convenience functions use these types. **traj** will be used mainly for representing the trajectories of the trips graphically, since QGIS is able to graph **geometry** types without any additional configuration or processing. It is worth mentioning that **tgeompoints** are not a discrete set of points, these simulate continuous spatial-temporal data, hence their utility.

3.3 GTFS Static

3.3.1 Obtaining the Data

GTFS Static data sets were obtained for trains [14], subways [13] and buses [10]. The train data set spans from February to April 2020 and the subways dataset spans from January 2014 to October 2019. The latest dataset

on buses that can be found on the government website is from August 2019, so a more recent dataset, from OpenMobilityData that span from 20 April 2020 to 20 October 2020, was used.

3.3.2 Data Structure

3.3.2.1 Bus System

The downloaded transit feeds for the Buenos Aires Bus system includes the following files: **agency.txt**, **calendar_dates.txt**, **routes.txt**, **shapes.txt**, **stop_times.txt**, **stops.txt**, **trips_txt**

3.3.2.2 Subway and Railway System

Regarding the data structure for the railway and subway system, the files found were similar. However, one remarkable difference is the inclusion of the file *calendar.txt*. This is a GTFS supported alternative for providing recurring information in a more succint manner.

3.3.2.3 Bus Interpolation Problem

The Buenos Aires bus system data presented an abnormality which caused issues during interpolation. For some lines where two distinct stops were very close to each other the feed specified the arrival time to be the same for both stops.

```
trip_id,arrival_time,departure_time,stop_id,stop_sequence,timepoint,shape_dist_traveled
10803-1,08:31:52,08:31:52,6441112744,100,0,18927
10803-1,08:32:12,08:32:12,6441112685,101,0,19139
10803-1,08:32:12,08:32:12,6441112778,102,0,19161
10803-1,08:32:34,08:32:34,6441112773,103,0,19408
```

Figure 6: Bus System Abnormality

3.3.3 Data Pre-processing and Importing

The GTFS Static Pipeline was separated into two steps; Pre-Processing and Data Importing.

3.3.3.1 Pre-Processing

The Pre-Processing Pipeline includes two scripts:

- **Data Pruner**
  Python script responsible for removing unused columns from *GTFS* data.
- **Data Wrangler**
  Go script responsible for finding abnormalities in stop arrival times and modifying values to allow interpolation.

3.3.3.2 Data Importing

Once the data is preprocessed the Data Importing phase takes place. This step includes three scripts:

- **GTFS Importer**
  SQL script responsible for loading preprocessed data into Auxiliary tables.

- **Dates Importer**
  SQL script responsible for loading service dates into auxiliary tables according to the availabiliy of *calendar_dates.txt*, *calendar.txt* or both files.

- **MDB Importer**
  SQL script responsible for populating the *trips_mdb* table. Takes care of generating the geometry of every trips route, calculating arrival times to every trips stops and finally generating the *tgeompoint* from the GTFS information.

All agencies reported in the transit feed use *UTC-3* Timezone, so timestamps were loaded with *America/Argentina/BuenosAires* timezone.

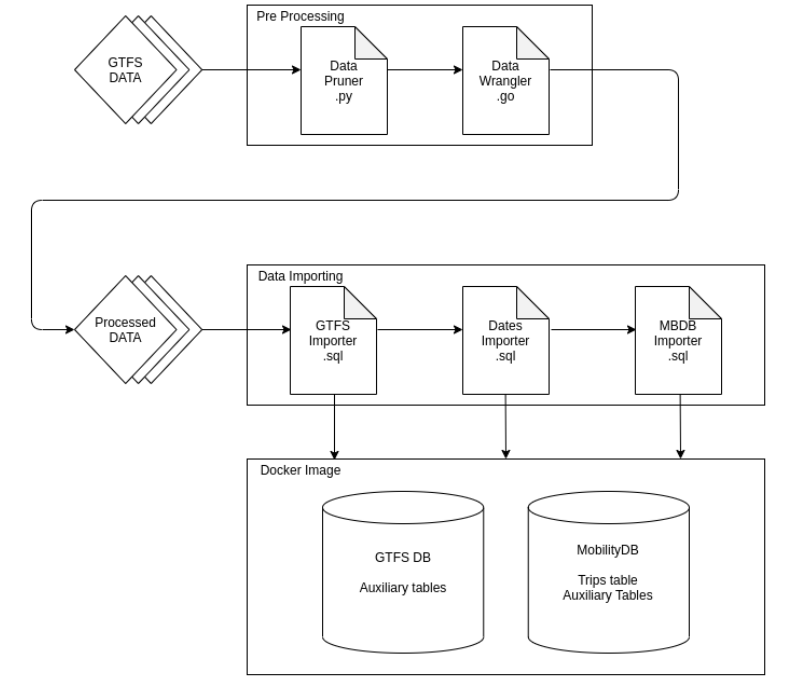

Figure 7: GTFS-Static importing pipeline

3.4 GTFS Realtime

3.4.1 Obtaining the Data

Upon some analysis it was discovered that the Transportation Ministry of Buenos Aires provides a GTFS Real Time API to allow users to track the state of the public transportation vehicles [9].

The API provides endpoints to request the status of both Trains and Buses of the city and its outskirts. Unfortunately, at the moment of writing there is no endpoint to query Subway lines in real time.

According to the API documentation the endpoints are updated at an interval of thirty seconds, they support both Protocol Buffers and JSON responses.

A scraper, denoted *BA Catcher*, that polls the transportation system at the update interval, was developed. To simplify development the scraper was run against the JSON endpoint starting August 18th and ending August 25th.

During the week of polling the Train endpoint did not return any values, so data extraction was limited to the Bus system.

3.4.1.1 BA Catcher Architecture

The Architecture for the BA Catcher is fairly simple. All resources can be found in the *GitHub Repository*. [3]

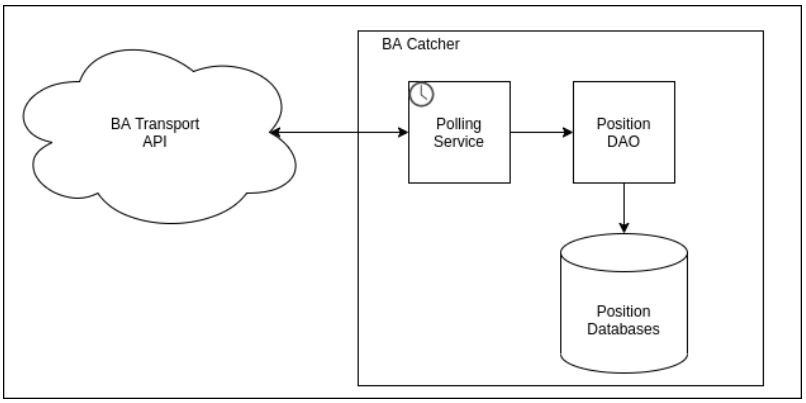

Figure 8: BA Catcher Architecture Diagram

- **BA Transport API**
  Transport API provided by the Buenos Aires Transport Ministry.
- **Polling Service**
  Service responsible for setting up the recurring JSON requests. Parsing the response and validating the values.

- **Position DAO**
  Data Abstraction. Responsible for persisting values to the database and assuring no duplicate values are persisted.

- **Position Databases**
  PostgresSQL database. Contains a single table *Positions* where all relevant information of the timestamped locations is persisted.

3.4.1.2 Checking the results

Taking advantage of the range types introduced in *MobilityDB* simple queries were executed to display a barchart which allowed the visualization of the number of timestamped locations reported for each trip.

It was noticed that for a large proportion of the trips less than 10 locations had been reported.

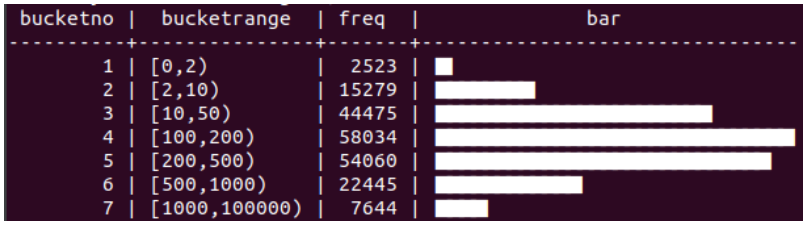

Figure 9: Timestamped location frequency Barchart

```
WITH buckets (bucketNo, bucketRange) AS (
        SELECT 1, intrange '[0, 2)' UNION
        SELECT 2, intrange '[2, 10)' UNION
        SELECT 3, intrange '[10, 50)' UNION
        SELECT 4, intrange '[100, 200)' UNION
        SELECT 5, intrange '[200, 500)' UNION
        SELECT 6, intrange '[500, 1000)' UNION
        SELECT 7, intrange '[1000, 100000)'),
vals (trip_id, amount) AS (
    SELECT trip_id, count(trip_id) AS amount
    FROM positions
    GROUP BY trip_id
),
valswithBucket (bucketNo, bucketRange, trip_id) AS (
        SELECT bucketNo, bucketRange, trip_id
        FROM buckets LEFT OUTER JOIN vals ON amount::int <@ bucketRange
),
histogram (bucketNo, bucketRange, freq) AS (
    SELECT bucketNo, bucketRange, count(*) AS freq
    FROM valswithBucket
        GROUP BY bucketNo, bucketRange
        ORDER BY bucketNo, bucketRange
)
SELECT bucketNo, bucketRange, freq,
    repeat('□', ( freq::float / max(freq) OVER () * 30 )::int ) AS bar
FROM histogram;
```

Figure 10: Timestamped location frequency Barchart Query

The query above makes use of *MobilityDB's* range types.

The query begins by defining a set of buckets, the bucket bounds were picked after executing some exploratory queries on the data. For different data sets other buckets may provide better results.

An intermediary table is populated which stores the **trip_id** followed by the **amount** of appearances in the data set. Taking advantage of *MobilityDB's* operations on ranges it is possible to join the previous table with it's respective bucket, according to the **amount** value.

At this point the final intermediary is populated. This table stores the bucket information alongside the number of **trip_id**'s with the corresponding **amount**.

The final **SELECT** statement outputs this information to the terminal alongside a simple ASCII bar chart to improve readability.

3.4.2 Data Structure

The fields stored from the HTTP requests for obtaining the real time data are the following:

- **trip_id**
  The trip identificator number that is used to reference the trip that the moving object is performing. This trip identificator coincides with the id used in the static data.
- **vehicle_id**
  An identificator number used to reference the particular vehicle in service.
- **instant**
  A timestamp for the data being sent. This timestamp is in **POSIX** time.
- **latitude**
  The latitude coordinate of the vehicle at the informed instant in the Universal Transverse Mercator coordinate system. [25]
- **longitude**
  The longitude coordinate of the vehicle at the informed instant in the Universal Transverse Mercator coordinate system. [25]
- **startdate**
  The date in which the trip started, in *YYYMMDD* format.
- **starttime**
  The time in which the trip started, in 24h format.
- **direction_id**
  Indicates the direction of travel for a trip, can be a **0** or **1**, e.g. outbound or inbound.

There were several rows which had positive latitude and longitude values, which are not coherent with the geographical location of the City of Buenos Aires. These values had coordinates for points somewhere in the Atlantic Ocean, clearly erroneous. When these points were plotted with the same modulus, but negative values, they matched the current trip.

3.4.3 Data Pre-processing and Importing

The following diagram illustrates the pipeline created to import the GTFS Realtime data:

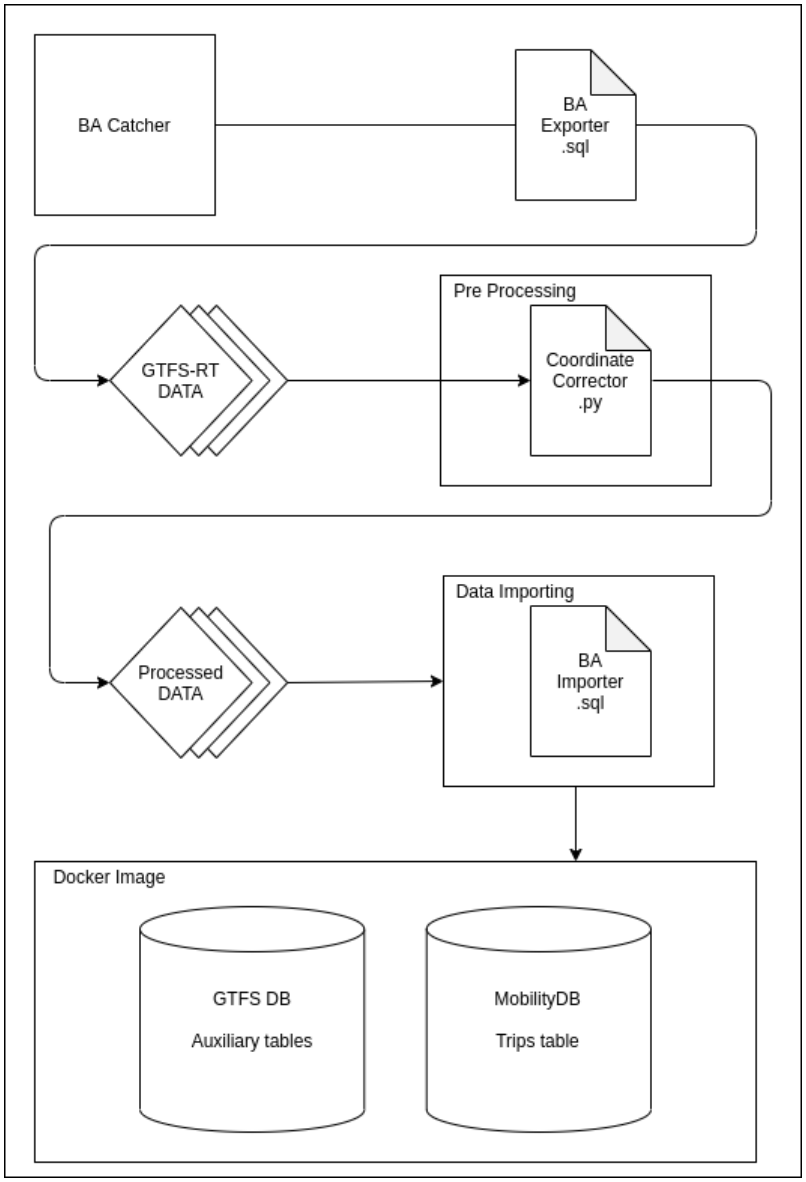

Figure 11: GTFS-Realtime importing pipeline

The **BA Catcher** component has been previously explained. **BA Exporter.sql** uses the data that **BA Catcher** stored and creates a **CSV** file that used for pre-processing. **Coordinate Corrector.py** fixes the positive coordinate values by making them negative. **BA Importer.sql** creates a table called **positions** with the direct import of the fields mentioned in **section 3.4.2**. With this table, the script creates points of **geometry** type using the **ST_MakePoint()** function provided by PostGIS, the **SRID** of the **geometry** is set to 4326 (the standard longitude and latitude coordinates on

the Earths surface), since that is the format the data is obtained in. Then, the table described in **section 3.2** is created, and the data from the table **positions** is imported. The **tgeompoints** are created with the following line of code:

```
tgeompointseq(array_agg(
    tgeompointinst(
        point,
        (to_timestamp(instant) at time zone
        'America/Argentina/Buenos_Aires')) ORDER BY instant
    )
)
```

Figure 12: Converting a series of timestamps and points to tgeompoints

tgeompointinst() creates a **tgeompoint** with a single point at a certain instant, and by aggregating these with **array_agg()**, they can be passed to **tgeompointseq()** to create a **tgeompoint** that represents the whole trip, to be added to the **trips_mdb** table. After that, the **SRID** is converted to 5345 (the standard for defining geodesic coordinates within the Argentinean territory), and the additional field **traj** is filled by using the MobilityDB function **trajectory()**. This function returns the PostGIS trajectory that a **tgeompoint** contains.

3.5 Data Post-Processing

3.5.1 SRID modifications

Both Static and Real Time values were loaded using SRID 4326. This allowed the Latitude and Longitude values provided to be used to immediately identify the desired point on the map. SRID 4326 is convenient in that it provides great accuracy in a geodesic system such as the planet, however to be able to calculate distance, speed and other metrics a different reference system must be used.

The final decision was to utilize SRID 5345 which is a Cartesian system that encompasses all of Argentina and its surroundings. This reference system is maintained by the IGN which is the principal authority on geography in the country. [16] To actually make the reference system change a utility function provided by *MobilityDB* was executed once the data was already loaded.

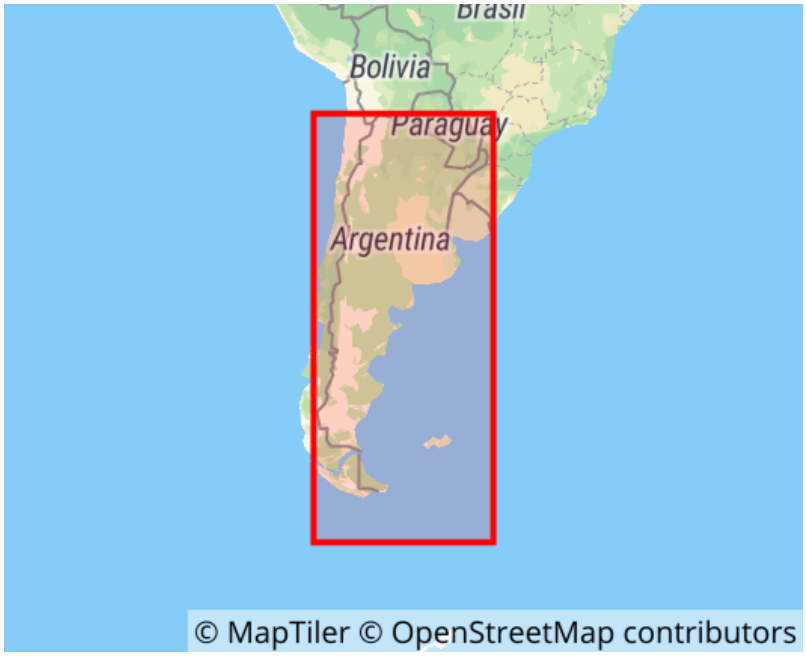

Figure 13: SRID 5345

3.5.2 Lines with low sampling rate

From the observation in **section 3.4.1.2** it was deemed necessary to remove lines which did not have enough information. A minimum of eleven real time, timestamped locations per bus line was established to ensure the trajectory provided enough value. All bus lines that did not pass this benchmark were removed.

3.5.3 Bus System Intersection

To be able to accurately compare the Real time and static Bus System Feeds it was deemed necessary to ensure that the compared bus lines were the same. In order to do so all bus lines not present in the real time feed were removed from the static feed, and all bus lines not present in the static feed were removed from the real time feed. Effectively only the intersection of both feeds was kept.

3.6 Map Matching

3.6.1 Experience with Barefoot

To improve the results the use offline map matching was attempted on the sampled data. *bmw-carit/barefoot* was set up and brought some issues.

- The offline map matching provided by *barefoot* can only correct one route per thread. At the scale of thousands of trips like in the city of Buenos Aires this restriction demands the development of a complex parallel pipeline which would handle the workload.
- Even though the BA Transport API documentation states that the response is updated every thirty seconds this statement does not seem to be true for every line.

  In practice the average interval between two adjacent samples for a given line was of 130 seconds. This alone would not be as worrying were it not for the fact that the interval variance is very high. In the worst cases the interval can be as long as 300 to 600 seconds.

As discussed above, as uncertainty increases so does the temporal complexity of the matching algorithm. In this specific case were both the uncertainty (given by the interval) and the scale (given by the size of the transport network) were very high; the decision was made to explore alternative solutions for the map matching problem.

3.6.2 Proof Of Concept Map Matching Algorithm

Though the realtime data suffered from inaccuracies due to both gps signal errors and sampling frequency; the static data had none of these problems.
The static input data was provided already map-matched. It was therefore noticed that the problem of matching the realtime trajectories to the physical route network was equal to matching the realtime trajectories to the static ones.
In order to test this hypothesis, an example of a trajectory that clearly displays the problem was found. The *bus line 152* is a great example, because during its trajectory it goes by the presidential house, along a large roundabout which, with the current real time frequency, MobilityDB will be unable to create an adequate route for:

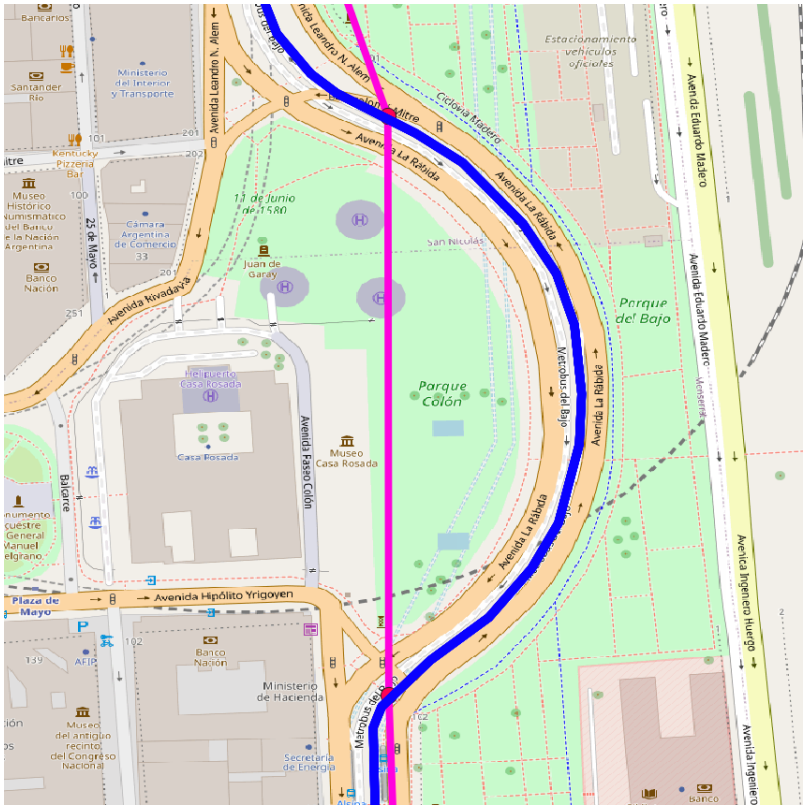

Figure 14: Bus Line 152, static route (blue), real time data points (red), generated route (pink).

The trajectory generated from the data points crosses the park through the middle, because the sampling frequency is not high enough to create a route that matches the static one. This example was deemed good enough to attempt a proof of concept algorithm of *Manual Map Matching*.

The following code was developed:

```
select tgeompointseq(array_agg(
        tgeompointinst(
                (dump).geom,
                startTimestamp(
                        nearestApproachInstant(transform(trip, 4326),
                        ST_SetSRID((dump).geom,4326)))
                )
        )) from (
select
        ST_LineSubstring(
                ST_SetSRID(traj, 4326),
                ST_LineLocatePoint(
                        ST_SetSRID(traj, 4326),
                        ST_ClosestPoint(ST_SetSRID(traj,4326),
                        ST_SetSRID(
                            ST_MakePoint('-58.3693348', '-34.6093502'), 4326)
                        )
                ),
                ST_LineLocatePoint(
                        ST_SetSRID(traj, 4326),
                        ST_ClosestPoint(ST_SetSRID(traj,4326),
                        ST_SetSRID(
                            ST_MakePoint('-58.369340001', '-34.606379996'), 4326)
                        )
                )
        ) as line,
        ST_DumpPoints(ST_LineSubstring(
                ST_SetSRID(traj, 4326),
                ST_LineLocatePoint(
                        ST_SetSRID(traj, 4326),
                        ST_ClosestPoint(ST_SetSRID(traj,4326),
                        ST_SetSRID(
                            ST_MakePoint('-58.3693348', '-34.6093502'), 4326)
                        )
                ),
                ST_LineLocatePoint(
                        ST_SetSRID(traj, 4326),
                        ST_ClosestPoint(ST_SetSRID(traj,4326),
                        ST_SetSRID(
                            ST_MakePoint('-58.369340001', '-34.606379996'), 4326)
                        ))
        )) as dump, trip
from trips_mdb_static
where starttime::DATE = '2020-08-25' and trip_id = '10000-1') as subquery;
```

The script will be explained in parts, first the subquery, from line 9 onwards. The subquery returns the line of the static trajectory that is contained between the two points closest to the red points, which are the real points obtained with the API. This trajectory will be referred as **line**. It also returns **dump**, which is an array of points that are contained in the previously mentioned trajectory. The *where* conditions (line 45) specify the exact trip that is shown on figure 14.

To obtain **line** and **dump**, **ST_ClosestPoint** was utilized with the trajectory and the red point shown on the figure. This returns the closest point to the red one, within the trajectory. With the function **ST_LineLocatePoint** the percentage of the trajectory in which the mentioned point is found is obtained. By calling **ST_LineSubstring** with both obtained points, and the trajectory, the trajectory contained within both datapoints is retrieved. With **line** and **dump**, the MobilityDB function **nearestApproachInstant** is used to generate the **tgeompoint**, and thus obtain the map matched route:

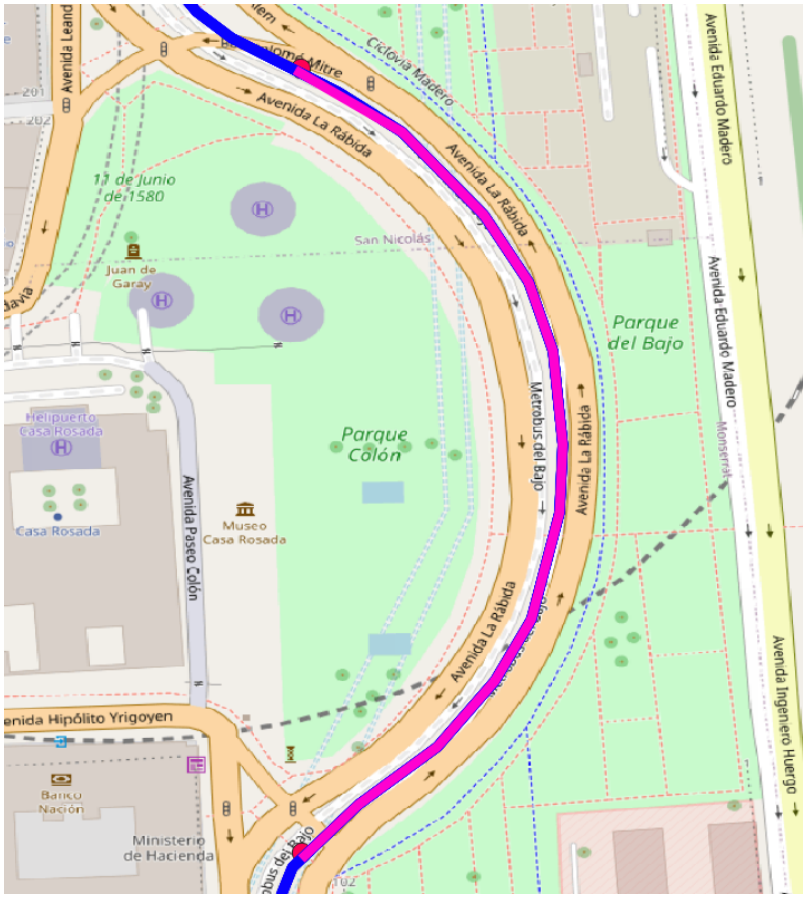

Figure 15: Corrected trajectory with map matching algorithm

The pink route can be observed to respect the original route spatially, and the time values associated to the new points can be checked too to ensure reasonable results:

```
mobilitydb=# select unnest(timestamps(trip)) as timestamps from map_matched_tgeom;
          timestamps
-------------------------------
 2020-08-25 12:55:39.493533+00
 2020-08-25 12:55:46.995048+00
 2020-08-25 12:55:56.89431+00
 2020-08-25 12:56:05.764556+00
 2020-08-25 12:56:12.986334+00
 2020-08-25 12:56:24.17368+00
 2020-08-25 12:56:31.057966+00
 2020-08-25 12:56:40.551818+00
 2020-08-25 12:56:47.947588+00
 2020-08-25 12:56:56.129312+00
 2020-08-25 12:57:07.897554+00
 2020-08-25 12:57:19.910053+00
(12 rows)
```

Figure 16: Time values associated to the map matched route

In conclusion the results obtained with the *Manual Map Matching* solution were deemed reasonably good. In this particular case the method was better than traditional map matching because it takes advantage of the expected route data. Utilizing *MobilityDB* functions, a custom map matching algorithm was produced; the algorithm matches error-prone discrete points to a known route in the system. However due to time constraints the results which follow are not map matched.

3.7 Analysis and Results

In the following section, the imported data will be visualized with various types of graphs and some samples of spatio temporal queries will be shown, with the objective of understanding the imported data sets and showcasing the capabilities of *MobilityDB*. First, the tools used to create the visualizations will be listed and explained, then visualizations for **GTFS-Static** data will be shown, followed by **GTFS-Realtime** visualizations and lastly, visualizations that use both datasets with some observations about the data displayed.

3.7.1 Tools

To create the following results Grafana was used, a free software with the Apache 2.0 license [6]. Grafana is a web application that is able to create dashboards with data from multiple databases. For the visualizations that contain maps, QGIS [22] was used, a very useful tool for geolocation data, because of its seamless integration with PostGIS data types. The map used was provided by OpenStreetMap [21].

3.7.2 GTFS-Static

3.7.2.1 Subways

The following visualization shows the imported GTFS Static data for the subway lines. It shows the stops overlapped with the trajectory created from the **tgeompoints**:

Figure 17: Subway stops overlapped with trajectories

3.7.2.2 Railways

The following visualization show the imported GTFS Static data for Railway and Subway Lines.

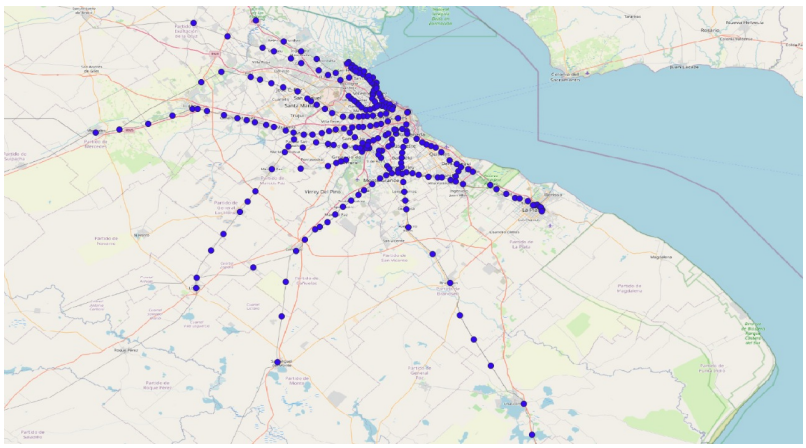

Figure 18: BA Railway System Stations

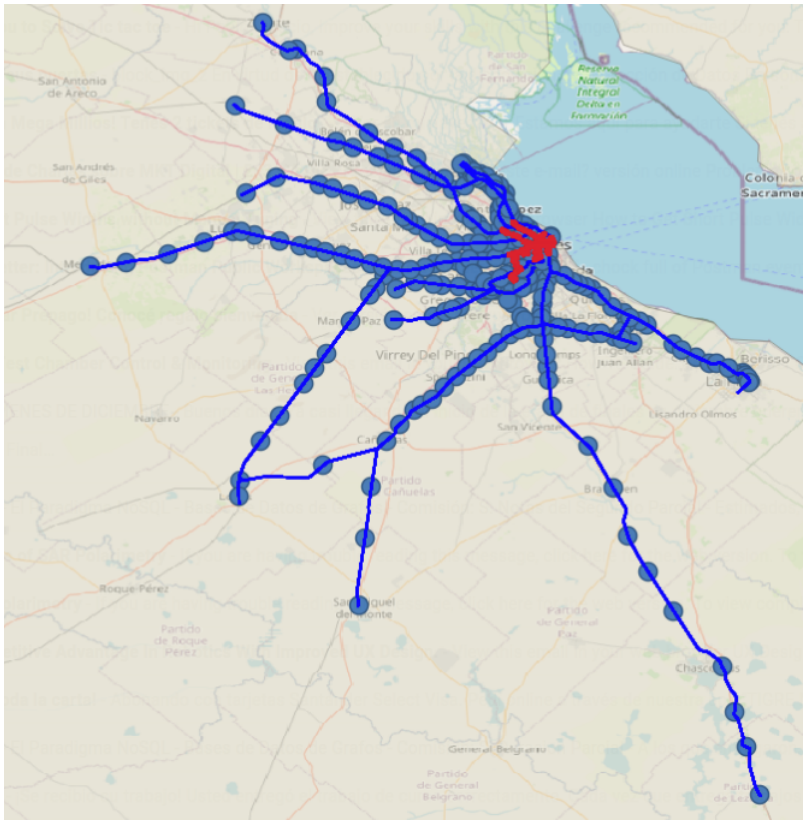

Figure 19: BA Subway (red) and Railway (blue) Systems

3.7.2.3 Buses

In the following Figure a sample of the the trajectories for the real time Bus feed are visualized.

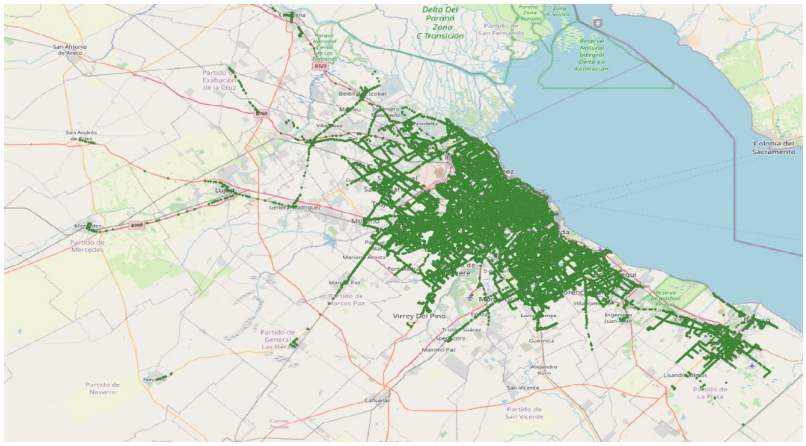

Figure 20: GTFS Static bus lines

3.7.2.4 Closeness to a Point of Interest

Taking advantage of *MobilityDB* utility functions it is possible to visualize the trajectories for all Buses that pass close to a Point of Interest. In this case trajectories and identifiers for buses that pass less than 200 meters away from *Teatro Colón* were queried.

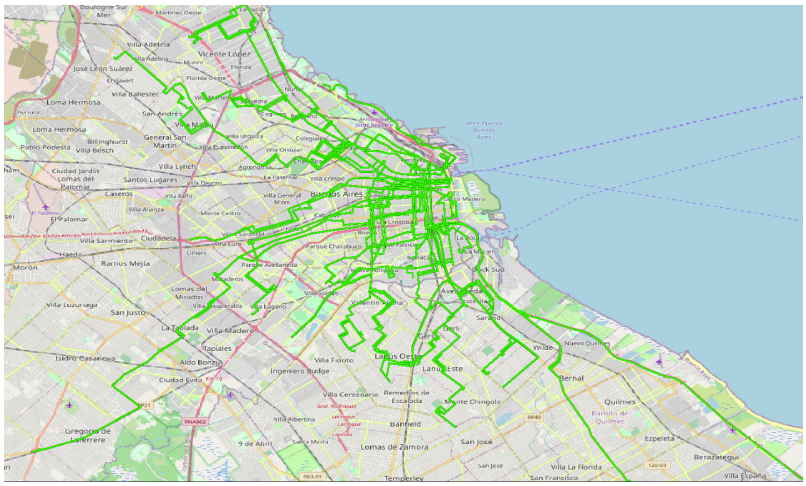

Figure 21: Bus Trajectories close to Teatro Colón

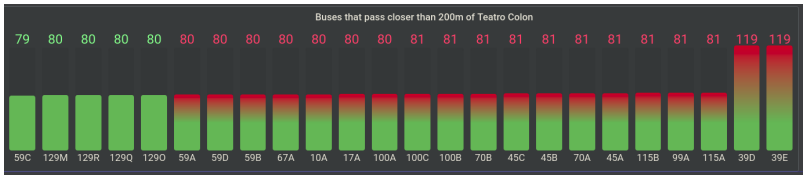

Figure 22: ID's of Bus lines that pass less than 200m of Teatro Colón

```sql
-- Lines that pass less than 200 m of Teatro Colon
WITH trip_distances as (
 SELECT bl.route_short_name as line,
      ST_Length(
        shortestLine(
            trip,
            ST_SetSRID(
              ST_MakePoint(4199468.71, 6145133.6),5345))
            ) as distance
    FROM trips_mdb as st join buslines as bl on st.trip_id = bl.trip_id
    WHERE ST_Length(
       shortestLine(
          trip, ST_SetSRID(ST_MakePoint(4199468.71, 6145133.6),5345))
          ) < 200
)
SELECT NOW() AS "time", line AS metric, AVG(distance) as value
FROM trip_distances
GROUP BY line
ORDER BY value ASC;
```

Figure 23: Bus Trajectories close to Teatro Colón Query

The core of the query is in the definition of the auxiliary table **trip_distances** in which the bus' line and the shortest distance to the point of interest are stored.

In order to calculate the latter value *MobilityDB* functions are used, in particular **shortestLine** which takes in a **tgeompoint** and a **geompoint** and provides the shortest line which connects the two figures. Online web tools were used to find the coordinates of the Point of Interest (in SRID 5345). These coordinates were passed in the function **ST_MakePoint**.

With the line built *PostGIS'* length function can be called to retrieve the desired metric. The final **SELECT** statement simply provides the data in a format readable by **Grafana**.

Since a single bus line may be associated to many **trip_ids** the final output averages the minimum distance from all trips for the given bus line.

3.7.3 GTFS-Realtime

3.7.3.1 Buses

The following visualization shows a sample of all the reported positions of buses registered in the week of 18 August - 25 August:

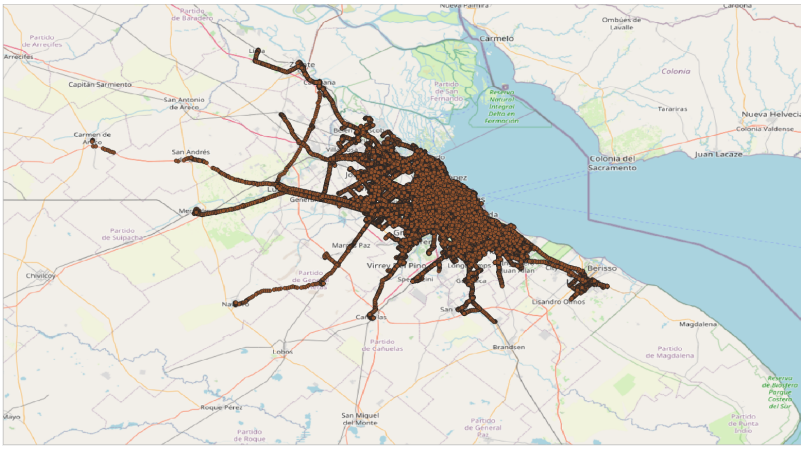

Figure 24: Bus Line 152A number of trips comparison

3.7.4 GTFS Static and Realtime comparison

By identifying the individual bus lines with the *trip_id* it is possible to query in both real time and static feeds for the different trips the particular bus line has done.

```
mobilitydb=# Select startTimestamp(trip), endTimestamp(Trip) from trips_mdb where trip_id = '10000-1'
;
     starttimestamp     |      endtimestamp
------------------------+------------------------
 2020-08-18 12:42:00+00 | 2020-08-18 14:24:56+00
 2020-08-19 12:42:00+00 | 2020-08-19 14:24:56+00
 2020-08-20 12:42:00+00 | 2020-08-20 14:24:56+00
 2020-08-21 12:42:00+00 | 2020-08-21 14:24:56+00
 2020-08-24 12:42:00+00 | 2020-08-24 14:24:56+00
 2020-08-25 12:42:00+00 | 2020-08-25 14:24:56+00
(6 rows)

mobilitydb=# Select startTimestamp(trip), endTimestamp(Trip) from trips_mdbrt where trip_id = '10000-
1';
     starttimestamp     |      endtimestamp
------------------------+------------------------
 2020-08-25 15:43:54+00 | 2020-08-25 17:08:24+00
 2020-08-20 15:43:24+00 | 2020-08-20 17:03:58+00
 2020-08-19 15:46:22+00 | 2020-08-19 17:12:56+00
 2020-08-21 17:24:54+00 | 2020-08-21 17:44:24+00
(4 rows)
```

Figure 25: GTFS-Static importing pipeline

The results show that in reality the busline has missed two of the trips it was expected to do during the week.

By applying this same technique the trajectory of the same bus line in both real time and static feeds can be visualized in *QGis* and confirm their similarity.

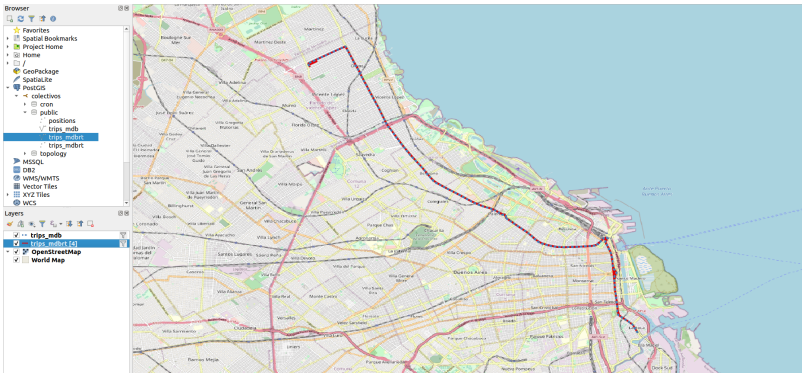

Figure 26: Bus Line 152A trajectory comparison

In the following code, the average speed of the vehicles is calculated grouped by starting hour, and grouped by starting day:

```
-- Average speed by starting hour (ST)
SELECT AVG(twavg(speed(Trip))) as static,
       date_trunc('hour',startTimestamp(Trip)) at time zone 'GMT-3' as time
FROM Trips_mdb
group by time
order by time;

--Average speed by starting hour (RT)
SELECT AVG(twavg(speed(Trip))) as realtime,
       date_trunc('hour', starttimefull) at time zone
       'America/Argentina/Buenos_Aires' as time
FROM Trips_mdbrt
WHERE starttimefull is not null
group by time
order by time;

-- Average speed by starting day (ST)
SELECT AVG(twavg(speed(Trip))) as static,
       date_trunc('day',startTimestamp(Trip)) at time zone 'GMT-3' as time
FROM Trips_mdb
group by time
order by time;

--Average speed by starting day (RT)
SELECT AVG(twavg(speed(Trip))) as realtime,
       date_trunc('day', starttimefull) at time zone
       'America/Argentina/Buenos_Aires' as time
FROM Trips_mdbrt
WHERE starttimefull is not null
group by time
order by time;
```

Figure 27: Code to calculate average speed grouped by hour and date

With the **trip** field as a **tgeompoint**, the *MobilityDB* function **twavg()** can be used to calculate the average speed of a trip. **twavg()** receives a list of numbers with a temporal value and calculates the time-weighted average of these numbers. By using another MobilityDB function, **speed()**, in conjunction with **twavg()** the average speed of various trips is obtained.

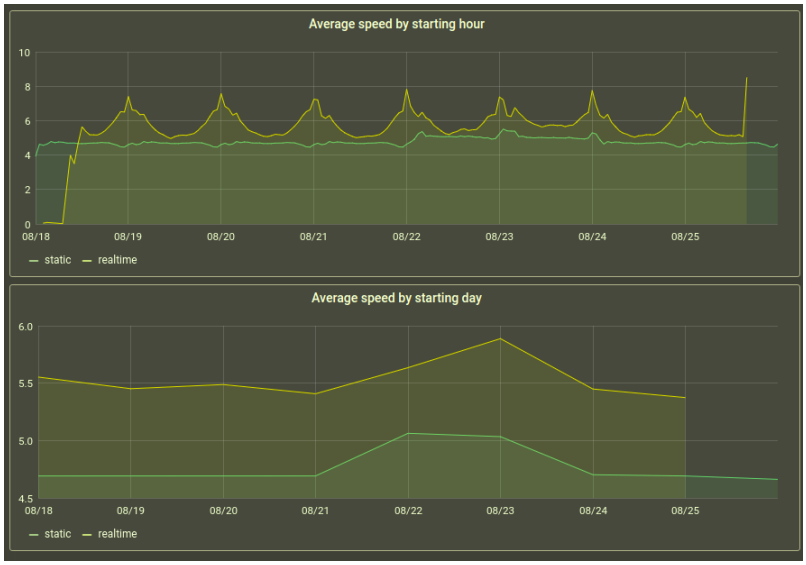

Figure 28: Average speed grouped by hour and date comparison

The main takeaway is that the buses are moving considerably faster than what the itinerary expects them to. It also looks like the itinerary does not take into account the traffic congestion changes throughout the day, while in the real time data obtained, Maximums in average speed can be observed at midnight every day. The average speed of buses oscillates between 5m/s and 7.5m/s (18km/h - 27km/h).

When grouped by day, in both Real time and itinerary data the average speed rises on weekends, and is constant on weekdays. This is expected, since there are less commuters on weekends to create traffic jams and congestion in the streets.

Next, the average delays, grouped by bus lines, is calculated:

```
1  -- Average delay by line
2  SELECT bl.route_short_name AS BusLine,
3    AVG(EXTRACT(EPOCH FROM timespan(rt.trip))/60
4    - EXTRACT(EPOCH FROM timespan(st.trip))/60) as Delay
5  FROM trips_mdb as rt
6    join trips_mdb_static as st on rt.trip_id = st.trip_id
7    join buslines as bl on rt.trip_id = bl.trip_id
8  GROUP BY route_short_name
9  ORDER BY value DESC LIMIT 25;
```

Figure 29: Code to calculate average delay grouped by bus line

BusLine	Delay
312R3	3.20468363784589
532A	2.55
299A	-1.04480389047468
299S	-1.64213011542494
59C	-2.06060606060607
15K	-2.08333333333334
514R4	-2.39485631219942
17A	-3.02439832468638
68B	-3.20222222222222
354B	-3.69773278560318
500DA	-3.72171972986006
299M	-3.90768947755683
500DI	-4.4161150259806
299C	-4.51627358490561
324T5F	-4.53900343642612
506R1	-4.74761137162968
524A	-5.26533639338713
461B	-5.34165852069052
506R2	-6.14219698453121
59A	-6.14947412081952
318D	-6.25016418287456
263BC	-6.45578938681213
634D	-6.74409590198573
276D	-7.64290448343078
526A	-7.8562386775354

Figure 30: Average delay by bus line

MobilityDB's function **timespan** allows querying the duration of the trip, after that the query above uses pure *PostgreSQL* syntax to extract seconds from it. By calculating the difference between the durations of both *tgeompoints* it is possible to retrieve the delay in completing the whole trip. It can be observed that only two lines have delays with respect to the itinerary. Even though this data is surprising, it is coherent with the speed data in the previous section. These results are expressed in minutes.

3.7.4.1 Bus lines Heatmap

With the real time and static *tgeompoints* created the *trip_id* was used to find both the theoretical and real trajectories of every bus in the system.

Taking advantage of *MobilityDB's* functions to find regions of *'closeness'* between *spatio-temporal objects* a delay heatmap for every bus in the system was built.

Segments of the route where both the realtime and static buses are close to each other are painted green, as the theoretical and real buses move away from each other the segment changes color towards red.

```
-- Segments where realtime was close to static.
-- Tolerance 100m
DROP TABLE heatmap1;
CREATE TABLE heatmap1 (
  trip_id text,
  seg_geom geometry
);
INSERT INTO heatmap1(
    trip_id,
    seg_geom)
SELECT ST.trip_id,
       getvalues(
           atPeriodSet(
               ST.Trip, getTime(atValue(tdwithin(ST.Trip, RT.Trip, 100), TRUE))
           )
        )
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;

-- Segments where realtime was close to static.
-- Tolerance 50m
DROP TABLE heatmap2;
CREATE TABLE heatmap2 (
  trip_id text,
  seg_geom geometry
);
INSERT INTO heatmap2(
    trip_id,
    seg_geom)
SELECT ST.trip_id,
       getvalues(
           atPeriodSet(
               ST.Trip, getTime(atValue(tdwithin(ST.Trip, RT.Trip, 50), TRUE))
          )
        )
FROM trips_mdb ST,
     trips_mdbrt RT
WHERE ST.trip_id = RT.trip_id
  AND ST.Trip && expandSpatial(RT.Trip, 50)
  AND atPeriodSet(ST.Trip, getTime(atValue(tdwithin(ST.Trip, RT.Trip, 50), TRUE)))
    IS NOT NULL
ORDER BY ST.trip_id;
.
.
.
```


Figure 31: HeatMap Calculation

The extract above show the queries for definition and population of the heatmap tables. Using MobilityDB syntax the segments for which the realtime and static buses were near each other up to a given tolerance were selected.

The function **tdwithin** generates a continuous boolean temporal type which has value **TRUE** when the temporal points are within a given distance from each other. Combining this with the **atPeriodSet** function it is possible to discard all segments from the *tgeompoint* where the trips are farther away than the given tolerance.

The **WHERE** clause allows improvement of the performance of the query by taking advantage of the topological operator && (contains) which can be indexed.

Five tables were created each containg the trip segments for differing degrees of tolerance. Once visualized in an application such as QGIS the stepped-tolerance creates a heatmap like visualization. In reality each decrease in tolerance creates an ever-decreasing subset of the previous table.

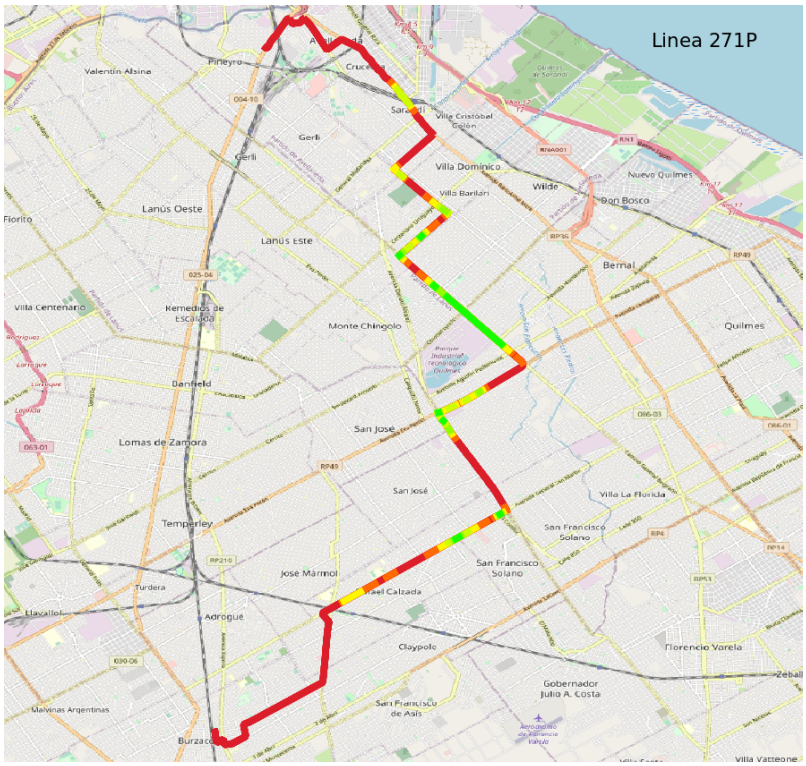

Figure 32: Bus Line Heatmap Line 271P

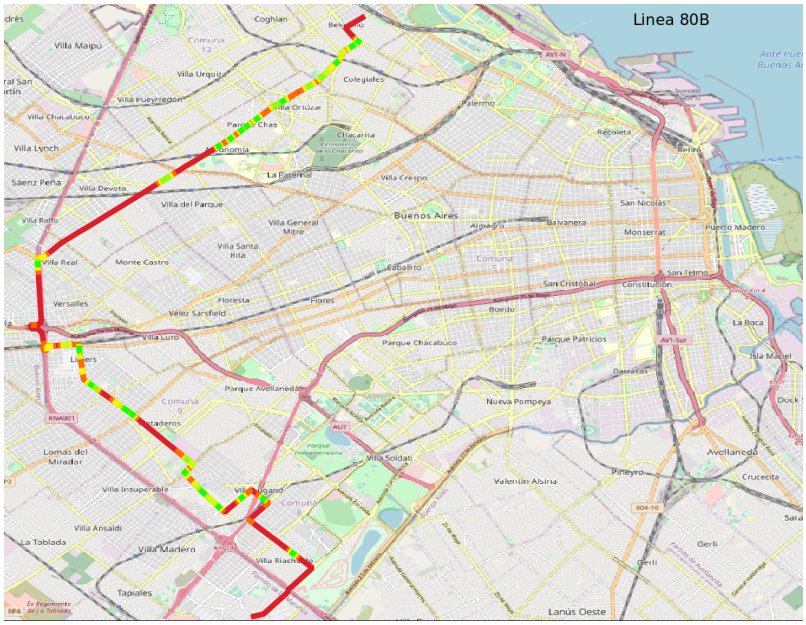

Figure 33: Bus Line Heatmap Line 80B

Figure 34: Bus Line Heatmap Line 152A

3.7.4.2 SARS-CoV-2 Lockdown Impact

Many visualizations confirm a significant difference in the location of the real bus when compared to its static itinerary. These differences are to be expected by any person who uses public transport in their daily life.

However the interesting observation is that these differences are not caused due to a delay but rather, in almost all cases, are due to a significant advance in the real bus with respect to its itinerary.

In any other year this observation would most likely lead one to conclude that the data is erroneous. However due to the extremely unusual events that have taken place in 2020 it is believed that the root cause for this observation is different.

The sampled data was taken during August in Argentina when the City of Buenos Aires and its outskirts were going through their fifth month in lockdown due to the effects of the SARS-CoV-2 Pandemic. Public transport was not stopped; however frequencies were reduced and there was strict restrictions on its usage. People were allowed to circulate independently only with a government emitted authorization.

Many studies have already been analyzing the effects of the lockdown on traffic and circulation. Amongst them those given by the *Google Mobility Report*.

Figure 35: Google Mobility Report Buenos Aires 25-Aug-2020

The root cause for the average increase in speed for bus lines throughout the sampled interval is believed to be best explained by the reduction in circulation and public transport usage. It will be very interesting to analyze the evolution of these metrics as the lockdown fizzles down and *normal* life begins again.

Be it on its effect on transit, retail, workplace or park usage, there is no doubt that the effects of this gargantuan lockdown will be analyzed for many years to come.

4 Conclusions

In this work the **GTFS** standard was discussed in-depth and the whole process of importing the data into *MobilityDB* was shown, from obtaining the dataset, to the structure of the data, pre-processing, loading it onto *MobilityDB* and post-processing to amend data errors and to format it in the most convenient way. The process that took place cannot be generalized to the importing of any **GTFS** dataset into *MobilityDB* because of the sheer breadth of different possibilities of formats that the **GTFS** standard allows;

however, a lot can be learned from this particular process used to load the *Buenos Aires* dataset. Many of the problems that were encountered in this work could very well be shared with the importing of another **GTFS** dataset onto *MobilityDB* and many of the solutions implemented can be re-used. The queries displayed in the analysis section of the work represent a useful guide to querying and displaying transit data in *MobilityDB* and are also a showcase of its power and flexibility. Even though successful results were not obtained using the proposed map matching tool, an alternate algorithm was implemented from scratch in **MobilityDB** and **SQL** code that gave promising results to fixing trajectories in certain complex points of trips. Finally, visualizations were interpreted and some conclusions were drawn in the context of the lockdown from the SARS-CoV-2 pandemic.

5 References

References

[1] *Barefoot*. URL: https://github.com/bmwcarit/barefoot. (accessed: 15.11.2020).

[2] *Buenos Aires Data Site*. URL: https://data.buenosaires.gob.ar/dataset?groups=movilidad. (accessed: 23.12.2020).

[3] *Github Repository*. URL: https://github.com/pabloito/MDB-Importer. (accessed: 15.11.2020).

[4] *Google Maps*. URL: https://www.google.com/maps. (accessed: 23.12.2020).

[5] *Google Maps APIs*. URL: https://developers.google.com/maps/documentation. (accessed: 23.12.2020).

[6] *Grafana License*. URL: https://github.com/grafana/grafana/blob/master/LICENSE.md. (accessed: 15.11.2020).

[7] *GTFS Real Time Overview*. URL: https://developers.google.com/transit/gtfs-realtime?hl=en. (accessed: 15.11.2020).

[8] *GTFS Real Time Reference*. URL: https://developers.google.com/transit/gtfs-realtime/reference. (accessed: 15.11.2020).

[9] *GTFS Realtime API*. URL: https://www.buenosaires.gob.ar/desarrollourbano/transporte/apitransporte/api-doc. (accessed: 15.11.2020).

[10] *GTFS Static Buses Dataset*. URL: https://openmobilitydata.org/p/colectivos-buenos-aires/1037/20200422. (accessed: 15.11.2020).

[11] *GTFS Static Overview*. URL: https://developers.google.com/transit/gtfs?hl=en. (accessed: 15.11.2020).

[12] *GTFS Static Reference*. URL: https://developers.google.com/transit/gtfs/reference. (accessed: 15.11.2020).

[13] *GTFS Static Subways Dataset*. URL: https://data.buenosaires.gob.ar/dataset/subte-gtfs. (accessed: 15.11.2020).

[14] *GTFS Static Trains Dataset*. URL: https://data.buenosaires.gob.ar/dataset/trenes-gtfs. (accessed: 15.11.2020).

[15] *Hermes*. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.2508&rep=rep1&type=pdf. (accessed: 15.11.2020).

[16] *IGN*. URL: https://www.ign.gob.ar/. (accessed: 15.11.2020).

[17] *MDBPaper*. URL: https://docs.mobilitydb.com/pub/DistMobilityDB_BigSpatial19.pdf. (accessed: 15.11.2020).

[18] *MobilityDB*. URL: https://docs.mobilitydb.com/MobilityDB/master/mobilitydb.pdf. (accessed: 15.11.2020).

[19] *MobilityDB Benchmark*. URL: https://docs.mobilitydb.com/pub/MobilityDBDemo_SSTD19.pdf. (accessed: 23.12.2020).

[20] *MobilityDB Benchmark infography*. URL: https://docs.mobilitydb.com/pub/MobilityDBDemo_SSTD19_Poster.pdf. (accessed: 23.12.2020).

[21] *OpenStreetMap*. URL: https://www.openstreetmap.org. (accessed: 15.11.2020).

[22] *QGIS*. URL: https://www.qgis.org/es/site/. (accessed: 15.11.2020).

[23] *Secondo*. URL: http://dna.fernuni-hagen.de/secondo/. (accessed: 15.11.2020).

[24] *SRID*. URL: https://en.wikipedia.org/wiki/Spatial_reference_system. (accessed: 15.11.2020).

[25] *Universal Transverse Mercator coordinate system*. URL: https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system. (accessed: 15.11.2020).