Public Transport (2022) 14:287–321
https://doi.org/10.1007/s12469-022-00290-8

CASE STUDY AND APPLICATION

Analyzing public transport in the city of Buenos Aires with MobilityDB

Juan Godfrid[1] · Pablo Radnic[1] · Alejandro Vaisman[1] · Esteban Zimányi[2]

Accepted: 10 January 2022 / Published online: 1 March 2022

Abstract
The General Transit Feed Specification (GTFS) is a data format widely used to share data about public transportation schedules and associated geographic information. GTFS comes in two versions: GTFS Static describing the planned itineraries and GTFS Realtime describing the actual ones. MobilityDB is a novel and free open-source moving object database, developed as a PostgreSQL and PostGIS extension, that adds spatial and temporal data types along with a large number of functions, that facilitate the analysis of mobility data. Loading GTFS data into MobilityDB is a quite complex task that, nevertheless, must be done in an ad-hoc fashion. This work describes how MobilityDB is used to analyze public transport mobility in the city of Buenos Aires, using both, static and real-time GTFS data for the Buenos Aires public transportation system. Visualizations are also produced to enhance the analysis. To the authors' knowledge, this is the first attempt to analyze GTFS data with a moving object database.

Keywords Moving objects · MobilityDB · Public transport · Mobility analysis · Big data

✉ Alejandro Vaisman
avaisman@itba.edu.ar

Juan Godfrid
jgodfrid@itba.edu.ar

Pablo Radnic
pradnic@itba.edu.ar

Esteban Zimányi
ezimanyi@ulb.ac.be

[1] Department of Information Engineering, Instituto Tecnológico de Buenos Aires, Lavardén 315, C1437FBG Buenos Aires, Argentina

[2] Department of Computer and Decision Engineering (CoDE), Universite Libre de Bruxelles, Avenue F. D. Roosevelt 50, 1050 Brussels, Belgium

1 Introduction

The ubiquity of GPS tracking devices and Internet of Things (IoT) technologies has resulted in a collection of massive amounts of data that describe the temporal evolution of moving objects, like cars, trucks, and pedestrians, for government agencies and private companies as well. Many applications exist for finding the best route between multiple points of interest (PoIs), estimating the arrival time to some destination, and even to predict the traffic on a certain route at a certain point in the future (e.g., Google Maps[1]). However, most of these solutions are proprietary and not available to the general public. For example, Google Maps APIs allow computing travel times and distances between locations, and determining the roads that a certain vehicle is traveling. These functionalities are limited by the lack of a query language, and, being proprietary, cannot be extended by the GIS community. This is addressed by MobilityDB (Zimányi et al. 2020), an open-source PostgreSQL and PostGIS extension that provides a set of functions and spatio-temporal datatypes that, together with PostGIS capabilities, form a flexible tool set for querying moving objects. Besides, the General Transit Feed Specification (GTFS) is a common format used to share public transportation schedules and real-time data, along with associated geographic information. GTFS defines two standards, namely GTFS Static, which is used to define schedules, and GTFS Realtime, which is used to communicate live positions of vehicles.

The present work shows how GTFS data can be used to analyze the public transportation system of Buenos Aires using MobilityDB. It describes the software developed to acquire, import, and load the data, and then to query and visualize the results. The solutions described in this work cannot be generalized for importing any GTFS dataset into MobilityDB, because of the different options that the GTFS standard allows. However, many datasets of GTFS transit information may share the same difficulties for importing the data, therefore this work can serve as a guide for many other similar cases. All software developed for this project is publicly available.[2] More precisely, the contributions of this work are as follows:

- A description of the data acquisition processes for GTFS Static and GTFS Realtime data for the City of Buenos Aires, in Argentina.
- A description of the process of importing the data of both GTFS standards into MobilityDB.
- A study on how MobilityDB can be used to analyze data about the public transport system. This study is carried out using public transport data for Buenos Aires, and includes, among others, a comparison of planned and actual schedules, average delay per bus line, and average speed per line. The goal is to show how MobilityDB makes these queries not only computationally efficient, but also easy to express.

[1] https://www.google.com/maps.

[2] https://github.com/pabloito/MDB-Importer.

The remainder of this paper is organized as follows: Sect. 2 discusses related work and provides the necessary background to make the paper self-contained. Section 3 describes MobilityDB and compares it against PostGIS as a solution for the study of mobility data. Section 4 presents the case study, and also describes the data acquisition, preprocessing, and post-processing tasks. Section 5 presents the analysis tasks and discusses the results. The paper concludes in Sect. 6.

2 Background and related work

This section presents basic concepts concerning moving object databases, and also their use in traffic analysis. Since the work in the present paper is based on the use of MobilityDB, this database is presented in detail in Sect. 3.

2.1 Moving object databases

Moving objects (MOs) are objects (e.g., cars, trucks, pedestrians) whose spatial features change continuously in time (Güting et al. 2005). Moving objects are represented as sequences of spatio-temporal points, of the form (x, y, t). As defined in Spaccapietra et al. (2013), MO data are typically divided into parts, called *trajectories*, defined within an interval $[t_s, t_f]$, where t_s and t_f represent the start and the end time instants of the trajectory, respectively. Over these trajectories, different kinds of analyses can be performed (e.g., pattern matching, semantic analysis). Although raw data come in the form of discrete points, for many applications that need to simulate and study the real movement of an object, a continuous representation of a trajectory is needed, and this requires appropriate interpolation functions. When these functions are provided, trajectories are called *continuous*, otherwise they are denoted *discrete*. *Moving object databases* (MODs) are databases that store the positions of MOs at any point in time, in other words, to represent a continuous function from an instant to a point with signature $f : instant \rightarrow (x, y)$. To represent MOs, the definition of appropriate data types is needed. As explained in Vaisman and Zimányi (2019); Zimányi et al. (2020), *temporal types* capture the evolution over time of base types and spatial types. For instance, temporal floats may be used to represent how the salary of a person evolves across time. Analogously, a temporal point may represent the evolution in time of the position of a vehicle or a pedestrian, reported by a GPS device, yielding a temporal geometry of type *point*. In the sequel, following Zimányi et al. (2020), a *route* denotes a certain spatial trajectory that a moving object can take, without a specific date or time associated to it, and a *trip* denotes a route repeatedly traversed at a certain time.

The first proposed MOD was SECONDO (Xu and Güting 2013), a MOD developed at the Fern Universität in Hagen, based on the model proposed by Güting et al. (2005). SECONDO provides an extensible architecture that can support spatial and spatio-temporal applications. This architecture has three main components, namely the kernel, the optimizer, and a graphical user interface (GUI). The kernel, being extensible, can implement a wide array of data models, through different algebra modules that provide a collection of type constructors and operators. Hermes is a MOD introduced by Pelekis et al. (2006); Pelekis and Theodoridis (2014), developed at the University of Piraeus, in Greece. In addition of being a MOD, Hermes can also be used as a pure temporal or a pure spatial system. Its functionality is achieved through a collection of abstract data types (ADT) and their corresponding operations, developed and provided as a data cartridge that extends SQL with MO semantics. One of the main features of SECONDO and Hermes is that they provide support to different kinds of MOs that go beyond the basic types and the geometric `point` type (for example, moving polygons are supported). However, SECONDO and Hermes present several drawbacks to be used as a real-world analytical tool. First, both prototypes are not easily integrated with relational databases. For example, the Hermes cartridges mentioned above, that encapsulate the MO functionality, extend the Oracle DBMS. Therefore, to build an application on top of the database, the application developer must embed PL/SQL scripts into a source program (e.g., a Java program). These scripts are the ones that actually call the Hermes type constructors. Integrating SECONDO with existing DMBSs is even more complicated, given that the former is a packed system.

The problems expressed above are overridden by MobilityDB, a database management system for moving object geospatial trajectories, such as GPS traces. Built on top of PostgreSQL and its spatial extension PostGIS, MobilityDB adds support for temporal and spatio-temporal objects to the PostgreSQL database. MobilityDB can be run in distributed environments, as described in Bakli et al. (2019), therefore it is appropriate to support high data volumes. MobilityDB, like PostgreSQL, is coded in the C programming language; therefore, it seamlessly extends the PostGIS library with temporal data types, which is appropriate for the purpose of running analytic indicators over relational databases. A current limitation of MobilityDB is that it only supports moving points. However, for traffic analysis, moving points are appropriate enough, as it will be shown in the next sections.

2.2 Using moving object data for traffic analysis

There is a wide array of works showing how moving object data can be used to analyze traffic in road networks. A comprehensive study of mobility data analysis problems is presented in Renso et al. (2013). Krogh et al. (2012) use GPS trajectories to estimate speeds on road segments, and propose several indicators. Meng et al. (2017) use loop detectors and taxi GPS trajectories for traffic analysis. Further, Hohmann et al. also study traffic from a user's point of view (Hohmann and Geistefeldt 2016). Data warehouses (Vaisman and Zimányi 2014) have been also proposed for moving object analysis in general, and for traffic analysis in particular. Andersen et al. (2014) propose the use of a data warehouse for analysing speeds, fuel consumption, among others. Formal frameworks for trajectory analysis using data warehouses are discussed in Leonardi et al. (2014) and da Silva et al. (2015). Recently, Vaisman and Zimányi (2019) proposed the use of MobilityDB for building trajectory data warehouses.

Although sometimes overlooked, the problem of preprocessing GPS data is crucial to guarantee that analysis results are correct. For instance, Krogh et al. (2012) only select trajectories that follow certain paths, and Meng et al. (2017) use map matching just for inferring average speeds. To give an idea of the impact of this step, this paper shows that a large portion of the data sets used here is cleaned out for several reasons. Parent et al. (2013) propose a three-step trajectory preprocessing methodology, consisting in a cleaning phase, followed by a map-matching task, and finally a compression step. Data cleaning is studied in Yan et al. (2013), although much of the task is generally performed manually, like discussed by Fu et al. (2016), usually limiting to clean GPS signal errors. This paper discusses also other kinds of errors.

As mentioned above, map matching is usually part of the data preprocessing tasks. Map matching consists in transforming absolute GPS coordinates into a sequence of road segments, matching raw GPS observations to the road network, accounting for constraints like speed limits and traffic directions. Map matching can be performed online or offline (Wei et al. 2013).

2.3 The GTFS specification

The General Transit Feed Specification (GTFS) is a data format used to define public transportation schedules and real-time data with associated geographic information. The GTFS has two versions, Static and Realtime, the former being the most widely used. GTFS Static[3] is used to predefine trip schedules, while GTFS Realtime[4] is a feed of real-time data with the positions and timestamps of the data points within trips and routes.

GTFS Static 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. The GTFS Static reference contains the following files:

- `agency.txt`. Lists the transit agencies that operate the transport routes.
- `stops.txt`. Lists the stops that compose the scheduled trips.
- `routes.txt`. Lists the routes in the public transport schedule.
- `trips.txt`. Lists the trips contained in the schedule.
- `stop_times.txt`. Links trips with stops, and adds the arrival time and departure time fields for each stop.
- `calendar.txt` and `calendar_dates.txt`. The file `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, which can be additions or removals.

[3] https://developers.google.com/transit/gtfs/reference.

[4] https://developers.google.com/transit/gtfs-realtime/reference.

- In addition to the above, there are several optional files. Some of these are `fare_attributes.txt` (which includes trip prices and payment methods), `fare_rules.txt`, `shapes.txt` (which includes data points that determine the trajectory taken between stops), `frequencies.txt`, `transfers.txt`, `pathways.txt`, `levels.txt`, `feed_info.txt`, etc.

GTFS Realtime is defined in a looser manner, compared against the static option. 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 all transport protocols can be used as well. A GTFS Realtime feed can support the following types of information:

- 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 the congestion level.

GTFS Realtime has two feed elements, *messages* and *enums*. The former represent complex types and the latter represent a list of fixed values generally used to communicate certain events. The feed elements are used when the web server communication method is the Protocol Buffer, used by the real-time data feed API for the city of Buenos Aires, which also uses a JSON format body within an HTTP response. The API sends nested messages as fields in the JSON body. The ones used for Buenos Aires are:

- `FeedEntity`. This message is sent on all HTTP requests, and 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 optional, and it is 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, a vehicle descriptor, a position described in latitude and longitude coordinates, a stop identifier of the current stop, and a timestamp in POSIX time.

There is limited scientific literature around GTFS. Vuurstaek et al. (2020) describe a bus stop mapping technique that combines the OpenStreetMap and GTFS open data sources. Kaeoruean et al. (2020) present two approaches for measuring the difference between the demand and supply for public transit. de Queiroz et al. (2019) analyze the conformity of GTFS routes and the actual bus trajectories in four cities in Brazil. Wessel and Widener (2017) study the problem of schedule padding, which is the extra time added to transit schedules to reduce the risk of delay. Regarding visualization, Kunama et al. (2017) present a tool called GTFS-Viz for preprocessing and visualizing GTFS data, and Bast et al. (2014) introduce a tool that shows a worldwide live map of real-time public transit data based on freely available GTFS timetable data and real-time delay information. Finally, Braga et al. (2014) describe a web-based application aiming to simplify the creation and editing of public transportation data. None of the works above includes the notion of moving objects in the analysis of transportation networks. To the authors' knowledge, this is the first attempt to analyze GTFS data with a moving object database.

3 MobilityDB

This section first presents a brief overview of MobilityDB to make the paper self-contained. Further details about MobilityDB can be found in Zimányi et al. (2020) and in the system's documentation.[5] The second part of the section compares the MOD solution based on MobilityDB, against the classic solution based simply in PostGIS.

3.1 MobilityDB Data types and functions

MobilityDB defines *temporal types* for handling objects whose value changes over time, for example stock prices or temperature, among others. Temporal values are initially built from a discrete set of values and associated timestamps (i.e., observations) and represent the evolution in time of the value. Since MO databases represent a continuous function, values between discrete time instants are interpolated using either a stepwise or a linear function.

Temporal types are based on four *time types*: the `timestamptz` type provided by PostgreSQL, and three new types, namely `period`, `timestampset`, and `periodset`. The `period` type represents a set of timestamps between a lower and an upper bound. The `timestampset` type is a collection of one or more `timestamptz` values. The `periodset` is a non-empty collection of ordered and non-overlapping `period` values.

MobilityDB provides four *temporal alphanumeric types*, namely `tbool` (temporal Boolean), `tint` (temporal integer), `tfloat` (temporal float), and `ttext` (temporal text). Such temporal types are typically used to represent dynamic properties of a moving object. A temporal Boolean can represent, for example, whether a car is driving below the speed limit of the road segment on which it is located. A temporal integer can be used to represent the gear of the car while a temporal float can be used to represent its speed. Finally, a temporal text can be used to represent for example the transportation mode of a moving person, such as walk, car, bicycle, etc.

MobilityDB also provides two *spatio-temporal types*, namely `tgeompoint` (temporal geometry point) and `tgeogpoint` (temporal geography point), which correspond to PostGIS types `geometry` and `geography`. The difference

[5] https://docs.mobilitydb.com/MobilityDB/master/mobilitydb-manual.pdf.

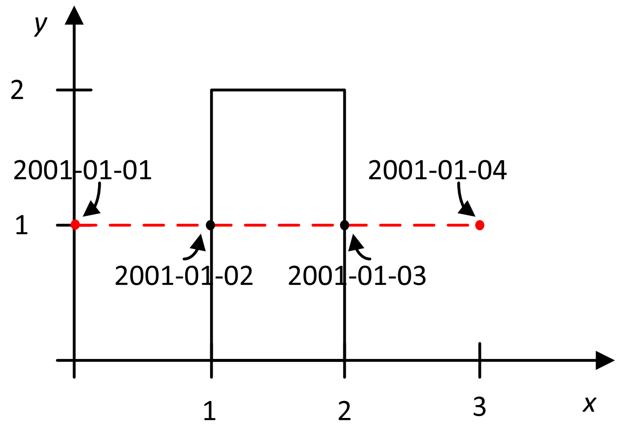


Fig. 1 Intersection between a `tgeompoint` and a polygon

between the two is the reference system: `geometry` points use a Cartesian reference system and allow calculation of speed and other distance-related metrics, while `geography` points use a geodesic reference system, which implies more complex operations.

MobilityDB includes a vast number functions to access and manage temporal types. Examples of these functions are `startTimestamp`, `endTimestamp`, `timespan`, `speed`, `direction`, `cumulativeLength`, `nearestApproachDistance` and many more. Some examples illustrating how MobilityDB's temporal types work are given next. For clarity, the results are given after each query in all the examples.

Example 1 The query below constructs two `tint` values and applies temporal addition to them. The resulting value is a `tint` as well.

```
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)
```

The result is obtained as follows. Since the value '1' exists between 2001-01-01 and 2001-01-02, and '1' exists between 2001-01-02 and 2001-01-04 (note the open and closed intervals), '1' and '2' are added in the intersection of the intervals, that is, 2001-01-02 and 2001-01-03.

Example 2 Now we illustrate the temporal intersection (`tintersects`) between a temporal point and a geometry. The resulting value is a `tbool`.

```
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]}
```

Figure 1 depicts the result given as text above. We can see that the initial and final positions of the temporal point do not intersect the polygon. However, performing a linear interpolation, the moving point intersects the polygon at two locations at instants `2001-01-02` and `2001-01-03`. The latter are indicated above as `t@2001-01-02` and `t@2001-01-03`. The former as `f@2001-01-01` and `f@2001-01-04`.

Example 3 We now show how we can create a table `Trips` containing a temporal column, add data to it, and query the table using MobilityDB.

```
(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)}');
```

Now, we write a query that uses the table `Trips` defined above, and retrieves the value of the temporal points at a specific timestamp, returning two points (note that points (2,0) and (1,1) exist in the table at the timestamp mentioned in the WHERE clause.

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

3.2 MobilityDB vs. PostGIS

A question that immediately arises is the following: Why do we need a MOD when existing tools, e.g., PostGIS, can also handle this problem? There are two key advantages for the MOD approach. On the one hand, queries over moving object data are more concise, easier to understand, and efficient. On the other hand, moving object data can dramatically reduce the storage space. Below we elaborate on these two issues.

3.2.1 Expressing trajectory queries

Consider a table `gpsPoint(tripID, pointID, t, geom)`, storing trajectories, represented using PostGIS.[6] In this table, `tripID` and `pointID` are, respectively, identifiers of the trip and the observation, `t` is a timestamp, and `geom` is the geometry of each point. There is also another table `pointOfInterest(poIID, poIName, geom)`, containing points of interest (PoI). Using these tables, a query asking for the points in the trajectory that are within 30 m from a point of interest (PoI) reads:

```
SELECT tripID, pointID, poIID
FROM gpsPoint a, pointOfInterest p
WHERE ST_DWithin(a.geom, p.geom, 30);
```

Note however, that the query does not account for the time interval of the situation, that is, for how long the trajectory was within 30 m from the PoI. The PostGIS query that solves this problem is quite more involved than the one above, as it can be seen below.

```
WITH pointPair AS (
  SELECT tripID, pointID AS p1, t AS t1, geom AS geom1,
   lead(pointID, 1) OVER (PARTITION BY tripID ORDER BY pointID) p2,
   lead(t, 1) OVER (PARTITION BY tripID ORDER BY pointID) t2,
   lead(geom, 1) OVER (PARTITION BY tripID ORDER BY pointID) geom2
  FROM gpsPoint ),
segment AS (
  SELECT tripID, p1, p2, t1, t2, ST_MakeLine(geom1, geom2) geom
  FROM pointPair
  WHERE p2 IS NOT NULL ),
approach AS (
  SELECT tripID, p1, p2, t1, t2, a.geom,
   ST_Intersection(a.geom, ST_ExteriorRing(ST_Buffer(b.geom, 30)))
   AS visibilityTogglePoint
  FROM segment a, pointOfInterest p
  WHERE ST_DWithin(a.geom, p.geom, 30) )
SELECT tripID, p1, p2, t1, t2, geom, visibilityTogglePoint,
  (ST_LineLocatePoint(geom, visibilityTogglePoint) * (t2 - t1))
   + t1 AS visibilityToggleTime
FROM approach;
```

The query above requires a deep understanding of SQL. It uses Common Table Expressions (CTE) for defining temporary tables to incrementally compute the final result of the query. Table `pointPair` stores every pair of consecutive points that belong to the same bus trip into one tuple. For this computation, it uses window functions, another advanced SQL feature. Table `segment` connects these pairs of consecutive points with a line segment, thus performing a linear interpolation between them. The locations where the bus starts/ends within 30 m from the PoI

[6] This example is based on the post https://techcommunity.microsoft.com/t5/azure-database-for-postgresql/analyzing-gps-trajectories-at-scale-with-postgres-mobilitydb-amp/ba-p/1859278.

are computed in the `approach` table. The final query lists such points and computes the time elapsed between them assuming constant speed. The complexity of the query above arises from the problem it addresses: it attempts to represent a *continuous movement* by means of reconstructing discrete GPS data. On the other hand, using a system that naturally handles continuous data may be a better option.

Consider now the solution using MobilityDB and the functions explained in Sect. 3.1. We first create a table `busTrip(tripID, trip)`, that stores the continuous trajectory. Attribute `tripID` is the trajectory identifier, while the attribute `trip` is of type `tgeompoint`, which is the MobilityDB type for storing a complete trajectory. The query above reads in MobilityDB as follows:

```
SELECT asText(atPeriodSet(trip,
  getTime(atValue(tdwithin(a.trip, b.geom, 30), TRUE))))
FROM busTrip a, pointOfInterest p
WHERE dwithin(a.trip, p.geom, 30)
```

The nesting of the functions `getTime`, `atValue`, and `tdwithin` returns the time periods during which a bus trip has been within a distance of 30 m from a PoI. The function `atPeriodSet` restricts the bus trip to only these time periods. The function `asText` converts the coordinates in the output to textual format.

As a conclusion, it clearly follows that a database that stores continuous trajectories will allow more natural and simple queries than a spatial database based on discrete data types. Furthermore, these queries will be more efficient, in particular since a full trip will be brought to memory with a single database access, while as illustrated above, multiple database accesses are required for the equivalent PostGIS query. Furthermore, more efficient algorithms can be defined for manipulating the continuous trajectories. For example, since the points and associated timestamps are stored in ascending order of time, MobilityDB uses binary search to efficiently locate the position of a moving object at a given timestamp.

3.2.2 Trajectory representation

As seen above, PostGIS represents the trajectory as a sequence of GPS points, each one stored on a single row. On the contrary, MobilityDB makes a more efficient use of the storage space taking advantage of the continuous trajectory notion. That is, for example, if an object moves in straight line at constant speed during a portion of the trajectory, only the starting and ending points of this segment need to be stored. All the intermediate points can be discarded. This can be seen in Fig. 2,[7] which compares both options. Note that the MobilityDB representation only used the points highlighted in green. As another example, when a moving object does not move, e.g., when it is stopped due to a traffic light or a traffic jam, MobilityDB will remove redundant observations and will only keep two of them when the stopped and when the object started to move again.

[7] Figure borrowed, with kind permission of the authors, from http://docs.mobilitydb.com/pub/MobilityDB_PGVision2021.pdf.

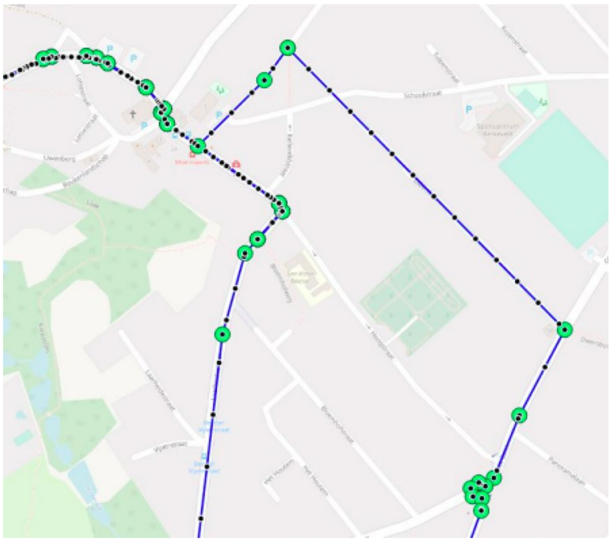


Fig. 2 Trajectory compression

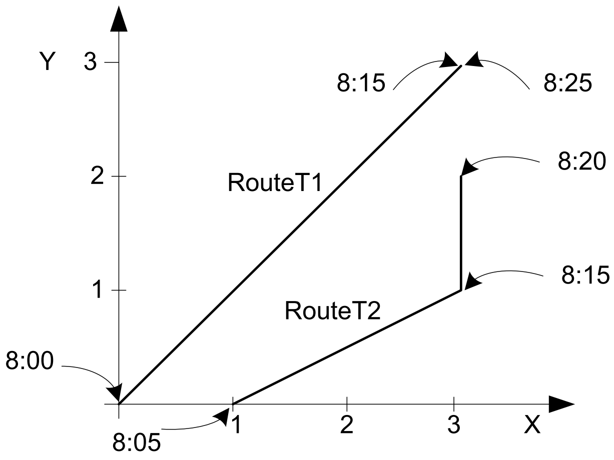


Fig. 3 Graphical representation of the trajectories of two buses

Experiments performed over the Moscow public transportation system showed a dramatic reduction in the storage space required to store transportation data: 10 billion rows a day (around 500 MB per day), are represented in MobilityDB by 15,000 rows (around 5 MB per day).

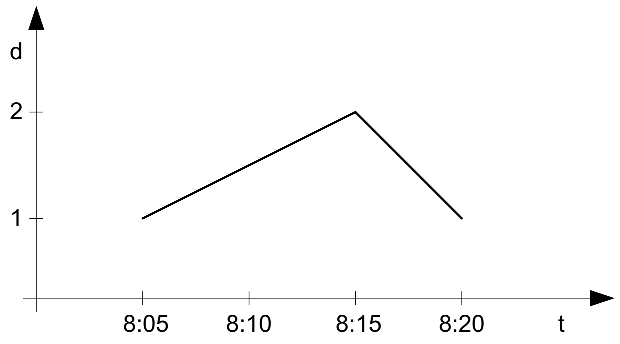


Fig. 4 Distance between the trajectories of the two vehicles in Fig. 3

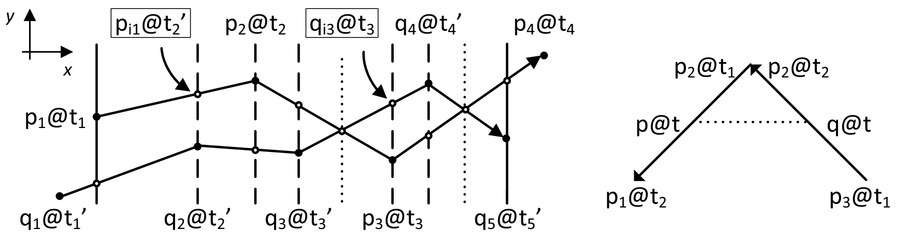


Fig. 5 Computing the temporal distance between two moving objects. Left: synchronization of the two trajectories. Right: computing the turning point for each segment of the synchronized trajectories

3.3 Other MobilityDB applications

Consider now two buses moving in a city, as shown in Fig. 3, call them `RouteT1` and `RouteT2`, respectively. We can see, for instance, that it takes 15 minutes to the first bus to go from point (0,0) to point (3 3). Then it stopped for 10 minutes at that point. We assume a constant speed between consecutive pairs of points. Thus, `RouteT1` travelled a distance of $\sqrt{18} = 4.24$ in 15 minutes, while `RouteT2` travelled a distance of $\sqrt{5} = 2.23$ in the first 10 minutes and a distance of 1 in the following 5 minutes.

In MobilityDB, the operation `trajectory` projects moving geometries into the spatial plane. The projection of a temporal point into the plane may consist of points and lines, the projection of a temporal line into the plane may consist of lines and regions, and the projection of a temporal region into the plane consists in a region. In our example, `trajectory(RouteT1)` would result in the leftmost line in Fig. 3, without any temporal information.

This raises an interesting question and illustrates another key feature of using MOD for mobility analysis. If we want to study how close to each other are two bus lines at any time, the spatial information would not be enough. We must compute the distance between the two lines at any time instant. Thus, the `ST_Distance` function in PostGIS would not be enough. We would need the MobilityDB function, `distance(RouteT1, RouteT2)`, which returns a temporal real value, shown in Fig. 4. It can be seen, for instance, that this function has value 1.5 at 8:10 and 1.41 at 8:15.[8]

We now briefly explain why this feature is not easily performed in PostGIS (or any spatial database). Consider the two moving points `p` and `q` on the left-hand side of Fig. 5. In order to compute the distance between these objects, we first need to temporally synchronize them, as the figure shows. This synchronization is performed internally by MobilityDB, restricting the two trajectories to their common time span (from t_1 to t'_5), and adding the intermediate synchronization points represented by the dashed vertical lines. In the figure, the solid circles represent the observations while the hollow circles represent the interpolated points added for synchronization. Two interpolated points are highlighted in the figure, shown within a box. Then, the computation of the temporal distance is performed for each synchronized segment. For this, we compute the distance at the beginning and the end of the segment but in addition we need to determine whether there is a *turning point*, which is the timestamp at which the distance between the two trajectories is minimal. These are represented in the figure by the dotted lines. In this figure, we have two turning points where the two objects are at the same point at the same time, and thus the distance between them is zero.

On the right-hand side of Fig. 5 we illustrate the general case of two synchronized segments, where the two objects are at the same point `p2` at two different timestamps. In the figure, `p` moves from `p2@t1` to `p1@t2`, while `q` moves from `p3@t1` to `p2@t2`. The turning point is indicated with the dotted line and in this case, opposite to the case above, the distance between the trajectories at the turning point is not zero. Therefore, the result of the temporal distance for this segment would be composed of three values: `ST_Distance(p2,p3)@t1`, `ST_Distance(p,q)@t`, and `ST_Distance(p2,p1)@t2`. This computation must be performed for every pair of synchronized segments. Therefore, the reader could understand intuitively that computing the temporal distance in PostGIS (i.e., without the MobilityDB temporal point data type) would be very complex and inefficient.

4 Case study

This section describes the use of MobilityDB for analyzing GTFS data, both Static and Realtime. For this, data of the public transportation system in Buenos Aires, Argentina are used. The area under study includes the city of Buenos Aires and its outskirts, known as the Metropolitan Area of Buenos Aires (AMBA).

The AMBA public transportation system consists of three main branches. The subway system, contained in the city itself, the metropolitan railway system, which connects Buenos Aires with its suburbs, and the bus system, composed of hundreds of municipal and provincial bus lines from all across the urban area. The open data site of the city of Buenos Aires[9] provides both, the itineraries for all branches of the AMBA transport system as well as real-time information

[8] Notice that the distance is a quadratic function and MobilityDB approximated it with a linear function.

[9] https://data.buenosaires.gob.ar/dataset?groups=movilidad.

on the moving vehicles. For the present work, the information referred above is loaded and analyzed using MobilityDB to obtain, for example, the average speed for each hour, for each day of the week, vehicles passing close to a PoI, and average delay for each bus. Visualizations like transport delay heat maps are also produced.

The data acquisition and preprocessing tasks are described next. The use of these data for analysis is described in Sect. 5.

4.1 Data model

The input data must be converted into MobilityDB's native spatio-temporal types. Thus, the output of the import process is a relational table called `trips_mdb` with the following columns:

- `trip_id`: The identifier of a particular trip.
- `vehicle_id`: The identifier of a particular vehicle in service.
- `startdate`: The starting date of the trip.
- `starttime`: The starting time of the trip.
- `trip`: The location and time information for the whole trip represented using MobilityDB `tgeompoint` data type. This is the column used in most of the queries below.
- `traj`: The spatial trajectory of the trip represented using PostGIS `geometry` data type. In other words, this is the spatial projection of the trip. This column is used for representing the trajectories of the trips graphically, e.g., using QGIS.

As discussed in Sect. 3.2, the MobilityDB type `tgeompoint` keeps the discrete set of points and associated timestamps (i.e., the observations) for a complete trip in a single value. All these observations would be stored in multiple rows in PostGIS. By collecting all the points and timestamps of a trip in a single value, MobilityDB is able to simulate continuous spatio-temporal data assuming a linear interpolation between subsequent observations. Therefore, in this work, GTFS data are turned into spatio-temporal continuous trajectories, such that, for GTFS Static they represent the planned itineraries while for GTFS Realtime they represent the actual mobility data in real time. Therefore, this case study shows the advantages of using moving object (i.e., continuous) data, instead of the classic solution that uses discrete data.

4.2 Data acquisition and preprocessing

This section reports the work required to acquire, import, process, and load data into MobilityDB structures. The study is divided into two sections: GTFS static and GTFS Realtime data.

```
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
```

Fig. 6 Bus system anomaly

4.2.1 GTFS static

Although GTFS Static data sets for the city of Buenos Aires can be obtained for trains, metro, and buses, the latest data set that can be found on the government's official website is from August 2019. Thus, we decided to use a more recent dataset from OpenMobilityData[10] that spans from April 20 to October 20, 2020, for the city's *bus system*. The downloaded data for the *bus system* in Buenos Aires, includes the following files: `agency.txt`, `calendar_dates.txt`, `routes.txt`, `shapes.txt`, `stop_times.txt`, `stops.txt`, and `trips_txt`. These files are described in Sect. 2.3 above. We remark that, although train and metro static data are available, since this is not the case for real-time data (see below), only bus data are considered for the present study.

The bus system data present an anomaly which raises issues during interpolation: there are cases where two distinct stops are very close to each other, and the feed specified the arrival time to be the same for both stops. This is illustrated in Fig. 6, where we can see two rows with the same `arrival_time` and different `stop_id`.

The GTFS Static Pipeline is split in two steps, namely, preprocessing and data importing. The preprocessing pipeline includes two scripts:

- Data pruner: a Python script that removes unused columns from GTFS data.
- Data wrangler: a Go script that finds anomalies in arrival times and modifies values to allow interpolation.

Once the data are preprocessed, the data importing phase takes place. This is depicted in Fig. 7, and includes three scripts:

- GTFS Importer: an SQL script that loads preprocessed data into auxiliary tables.
- Dates Importer: an SQL script that loads service dates into auxiliary tables depending on the availability of the files `calendar_dates.txt` and/or `calendar.txt`.
- MDB Importer: an SQL script that populates the `trips_mdb` table. It takes care of generating the geometry of every trip's route, calculating arrival times of all trip stops, and finally generating the `tgeompoint` from the GTFS information.

[10] https://openmobilitydata.org/l/401-buenos-aires-autonomous-city-of-buenos-aires-argentina.

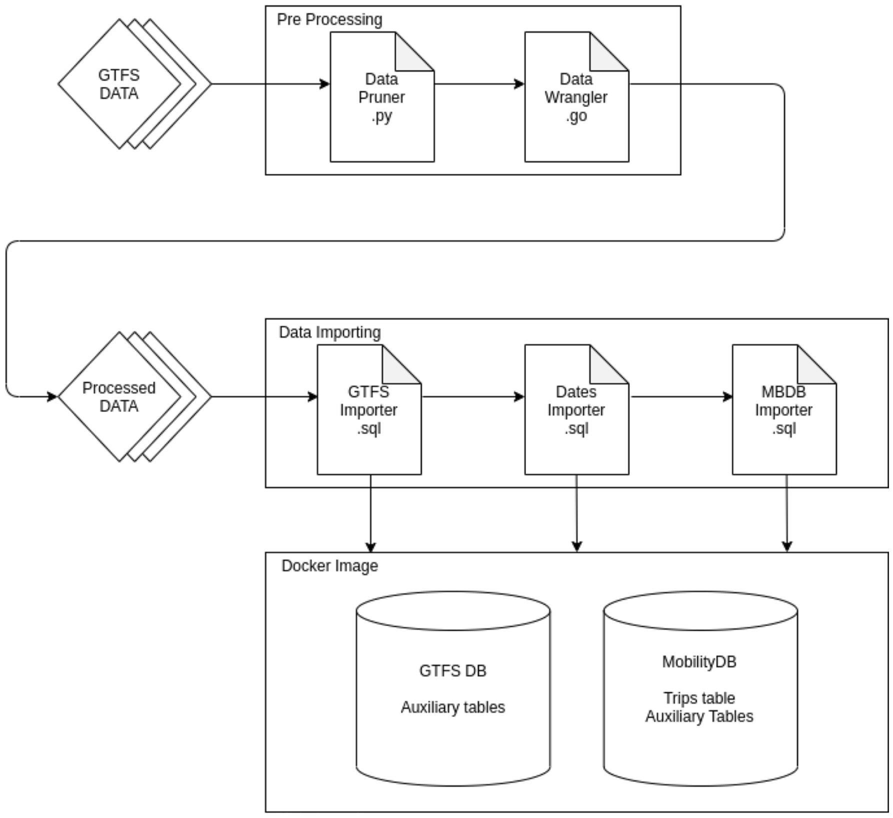


Fig. 7 GTFS-Static importing pipeline

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

4.2.2 GTFS Realtime

The Transportation Ministry of the city of Buenos Aires provides a GTFS Realtime API that allows users to track the state of the public transportation vehicles.[11] This API provides endpoints that expose the status of both, trains and buses in the city and its outskirts. The endpoints are updated at an interval of thirty seconds, and support both Protocol Buffers and JSON responses. At the moment of writing this work, there is no endpoint for querying subway lines in real time, and during the week of polling, the train endpoint did not return any values. Therefore, the data extraction is limited to the bus system. However, for the goals of this work, the bus data suffices.

BA Catcher A scraper, called `BA Catcher`, that polls the transportation system at the update interval, has been developed. The scraper is run against the

[11] https://www.buenosaires.gob.ar/desarrollourbano/transporte/apitransporte/api-doc.

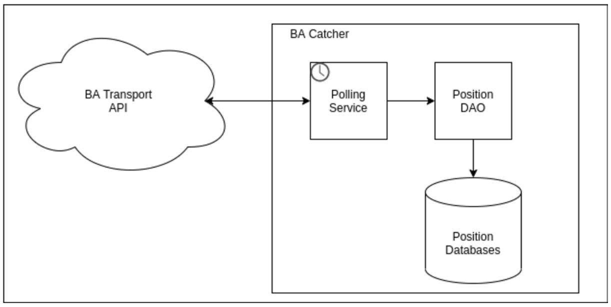
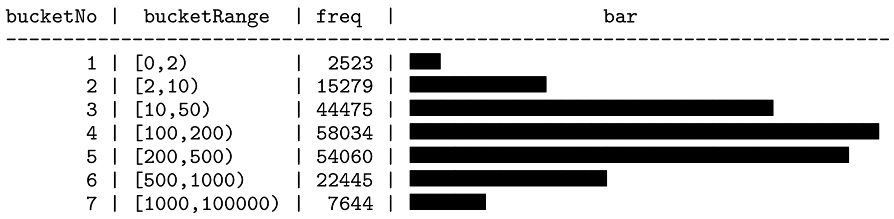


Fig. 8 `BA Catcher` Architecture Diagram



```
bucketNo |  bucketRange   | freq  |                  bar
--------------------------------------------------------------------------------
       1 | [0,2)          |  2523 |
       2 | [2,10)         | 15279 |
       3 | [10,50)        | 44475 |
       4 | [100,200)      | 58034 |
       5 | [200,500)      | 54060 |
       6 | [500,1000)     | 22445 |
       7 | [1000,100000)  |  7644 |
```

Fig. 9 Timestamped location frequency barchart

JSON endpoint starting August 18 and ending August 25, 2020. Figure 8 shows the architecture of the `BA Catcher`, composed of the following services:

- `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`: It is responsible for persisting values to the database, removing duplicate values.
- `Position Database`: a PostgreSQL database that contains a single table `Positions` where all relevant information of the observations is persisted.

For exploratory data analysis, the query below, expressed using PostgreSQL, produces a bar chart showing the number of timestamped locations reported for each trip. Figure 9 depicts the barchart, where it can be seen that there is a large portion of trips where less than 10 locations have been reported.

```
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)'),
tripCount (trip_id, no_observ) AS (
  SELECT trip_id, COUNT(*)
  FROM positions
  GROUP BY trip_id
),
bucketTrip (bucketNo, bucketRange, trip_id) AS (
  SELECT bucketNo, bucketRange, trip_id
  FROM buckets LEFT OUTER JOIN tripCount ON no_observ::int <@ bucketRange
),
histogram (bucketNo, bucketRange, freq) AS (
  SELECT bucketNo, bucketRange, COUNT(*)
  FROM bucketTrip
  GROUP BY bucketNo, bucketRange
  ORDER BY bucketNo, bucketRange
)
SELECT bucketNo, bucketRange, freq,
  repeat('■', ( freq::float / max(freq) OVER () * 30 )::int ) AS bar
FROM histogram;
```

Table buckets defines a set of buckets, whose bounds were selected after executing some exploratory queries on the data. Table tripCount stores for each trip_id the number of observations in the data set. This table is joined with its corresponding bucket in table bucketTrip, according to the no_observ value. Then, the histogram table associates each bucket with the number of trips in the bucket. The final SELECT statement outputs this information to the terminal alongside a simple ASCII bar chart to improve readability.

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

- trip_id: The identifier of the trip that the moving object is performing. This identifier coincides with the identifier used in the static data.
- vehicle_id: The identifier of the vehicle in service.
- instant: A timestamp for the data being sent, in POSIX time.
- latitude: The latitude of the vehicle at the instant, in the UTM coordinate system.
- longitude: The longitude of the vehicle at the instant, in the UTM coordinate system.
- startdate: The date in which the trip started, in YYYYMMDD format.
- starttime: The time in which the trip started, in 24h format.
- direction_id: The direction of travel for a trip, can be a 0 or 1, e.g., outbound or inbound.

There are several rows with positive latitude and longitude values, which are not coherent with the geographical location of the city of Buenos Aires. These values correspond to points somewhere in the Atlantic Ocean. When these points are plotted with negative values, they match the current trip.

Data acquistion and preprocessing Figure 10 illustrates the pipeline created to import the GTFS Realtime data used in this study. The `BA Catcher` component has been previously explained. The `BA Exporter.sql` program uses the data that `BA Catcher` stores, and creates a CSV file that is used for preprocessing. A Python program, `Coordinate Corrector.py` fixes the errors in the coordinate values mentioned above. The `BA Importer.sql` module creates a table called `positions` with the direct import of the fields mentioned above. With this table, the script creates points of the `geometry` type using the PostGIS function `ST_MakePoint`. The SRID of the `geometry` is set to 4326 (the WGS84 standard longitude and latitude coordinates on the Earth's surface), since that is the format in which data come. Then, the table described at the beginning of the section is created, and the data from the table `positions` is imported. The following code creates the `tgeompoint` of a trip.

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

The function `tgeompoint_inst` creates a `tgeompoint` with a single point at a certain instant, and by aggregating these with `array_agg`, they can be passed to `tgeompoint_seq` to create a `tgeompoint` that represents the whole trip, which is added to the `trips_mdb` table. The additional field `traj` is filled using the MobilityDB function `trajectory`, which returns the PostGIS `geometry` trajectory that a `tgeompoint` contains.

Both Static and Realtime values come in SRID 4326, which is a geodetic coordinate system. To calculate distance, speed, and other metrics, a plannar reference system is needed. Thus, the SRID 5345 is used, which is a Cartesian system that encompasses all of Argentina and its surroundings. To change the reference system, a MobilityDB function is executed over the loaded data. As another preprocessing action, bus lines with less than eleven real-time observations are removed to guarantee that they are useful for data analysis.

Finally, to compare the real-time and static bus feeds, all bus lines not present in the real-time feed are removed from the static feed, and, conversely, all bus lines not present in the static feed are removed from the real-time feed.

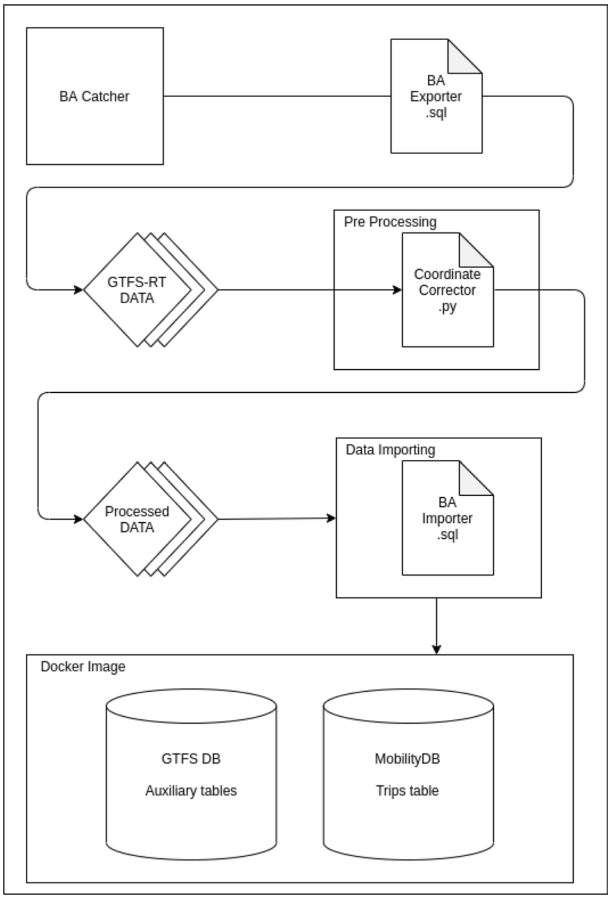


Fig. 10 GTFS Realtime importing pipeline

4.3 Map matching

To improve the results, offline map matching was applied to the data. For this, we used Barefoot,[12] an open-source Java library for online and offline map matching with OpenStreetMap. However, several problems were found, namely: (a) 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 constraint demands a parallel pipeline which would handle the workload; (b) 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

[12] https://github.com/bmwcarit/barefoot.

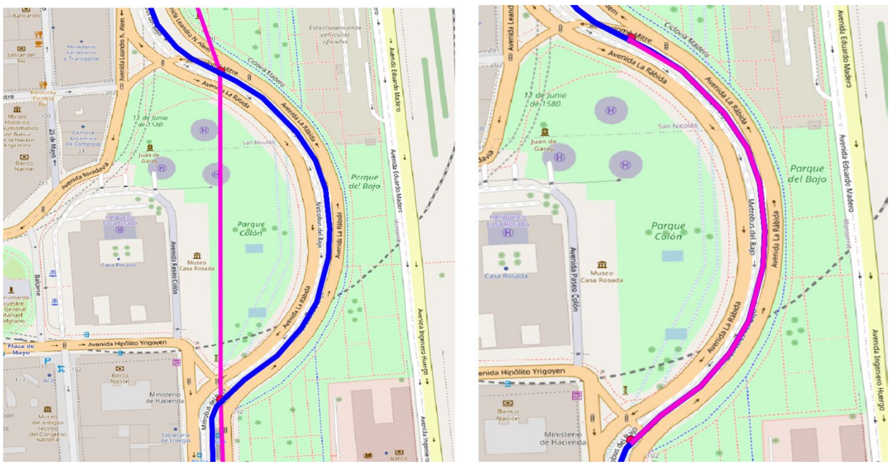


Fig. 11 Left: Bus line #152, static route (blue), real time data points (red), generated route (pink); Right: Trajectory corrected with the map-matching algorithm

line. In practice, the average interval between two adjacent samples for a given line was about 130 seconds, but the interval variance is very high, reaching a value up to 600 seconds. In this study, where both the uncertainty (given by the interval) and the scale (given by the size of the transport network) are very high, alternative solutions for the map-matching problem are explored, and described next.

Although real-time data suffer from inaccuracies due to both GPS signal errors and sampling frequency, these problems are not present in static data. The static input data are provided already map-matched. Therefore, the problem of matching the real-time trajectories to the physical route network equals matching the real time trajectories to the static ones. In order to test this hypothesis, an example of a trajectory that clearly displays the problem is shown next. The bus line #152 is an example of this issue, since during its trajectory it goes by the presidential house, along a large roundabout, and thus, with the current real-time frequency, MobilityDB is unable to create an adequate route, as shown on the left-hand side of Fig. 11.

The trajectory generated from the data points goes across the park, because the sampling frequency is not high enough to create a route that matches the static one. A manual map-matching algorithm is developed, and explained below. The code shown in Listing 1 produces such a match.

Listing 1 Map-matching algorithm

```
 1 SELECT tgeompoint_seq(array_agg(
 2   tgeompoint_inst(
 3     (dump).geom,
 4     startTimestamp(
 5       nearestApproachInstant(transform(trip, 4326),
 6       ST_SetSRID((dump).geom,4326)))
 7     )
 8   )) FROM (
 9 SELECT
10   ST_LineSubstring(
11     ST_SetSRID(traj, 4326),
12     ST_LineLocatePoint(
13       ST_SetSRID(traj, 4326),
14       ST_ClosestPoint(ST_SetSRID(traj,4326),
15       ST_SetSRID(
16           ST_MakePoint(-58.3693348, -34.6093502), 4326)
17       )
18     ),
19     ST_LineLocatePoint(
20       ST_SetSRID(traj, 4326),
21       ST_ClosestPoint(ST_SetSRID(traj,4326),
22       ST_SetSRID(
23           ST_MakePoint(-58.369340001, -34.606379996), 4326)
24       )
25     )
26   ) AS line,
27   STST_DumpPoints(ST_LineSubstring(
28     ST_SetSRID(traj, 4326),
29     ST_LineLocatePoint(
30       ST_SetSRID(traj, 4326),
31       ST_ClosestPoint(ST_SetSRID(traj,4326),
32       ST_SetSRID(
33           ST_MakePoint(-58.3693348, -34.6093502), 4326)
34       )
35     ),
36     ST_LineLocatePoint(
37       ST_SetSRID(traj, 4326),
38       ST_ClosestPoint(ST_SetSRID(traj,4326),
39       ST_SetSRID(
40           ST_MakePoint(-58.369340001, -34.606379996), 4326)
41       ))
42   )) AS dump, trip
43 FROM trips_mdb_static
44 WHERE starttime::date = '2020-08-25' and trip_id = '10000 - 1') AS subquery;
```

The subquery on Line 9 returns the line of the static trajectory that is contained between the two points closest to the red points (the extreme points, obtained with the API, that can be seen in the figure). This trajectory is referred as `line`. The query also returns `dump`, an array of points that are contained in the previously mentioned trajectory. The `WHERE` clause (Line 45) specifies the exact trip that is shown in Fig. 11. To obtain `line` and `dump`, the function `ST_ClosestPoint` is invoked with the trajectory and the red point shown in the figure. This returns the point within the trajectory, closest to the red one. 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 points obtained, and the trajectory, the trajectory contained within both data points is retrieved. With `line` and `dump`, the MobilityDB function `nearestApproachInstant` is used to generate the `tgeompoint`, and thus obtain the map matched route, shown on the right-hand side of Fig. 11. It can be noted that now the pink route matches the original route spatially.

In conclusion, the results obtained with the simple map-matching solution implemented in MobilityDB are reasonably good. Furthermore, this approach is considerably much more efficient than the one followed by Barefoot, which uses a Hidden Markov Model map-matching implementation. In this particular case, the method yields better results because it takes advantage of the expected route data. Obviously, this functionality can be implemented in PostGIS but this would require to follow an approach similar to the one sketched in Sect. 3.2 to connect subsequent observations, and rewrite in SQL all the functionality natively provided by MobilityDB for computing the turning points of the distance function (e.g., function `nearestApproachInstant` above) between the planned and the actual trajectories.

5 Analysis and results

This section shows the use of MobilityDB for analyzing static and real-time public public transport in the city of Buenos Aires. First, GTFS Static data are analyzed and the results displayed by means of visualizations. Then, analyses that use both GTFS Static and Realtime datasets together are carried out. The section concludes with a discussion of the results.

To create the visualizations reported below, Grafana,[13] a web application that allows creating dashboards with data from multiple databases, is used. For the visualizations that contain maps, QGIS[14] is used. The maps are obtained from OpenStreetMap.[15]

5.1 GTFS Static

Figure 12 shows a portion of the trajectories for the real time bus feed. The trajectories are computed using MobilityDB, as the spatial projection of the continuous spatio-temporal trajectories produced from the schedules and stops data. From these spatio-temporal trajectories, interesting analyses can be performed, as shown next.

Query 1 List the trajectories and their bus identifiers, that pass at a distance less than 200 m away from Colón Theater.

MobilityDB utility functions are used to compute and visualize the trajectories for all buses that pass close to a PoI, in this case, trajectories that pass at a distance

13 https://github.com/grafana/grafana.

14 http://qgis.org.

15 https://www.openstreetmap.org.

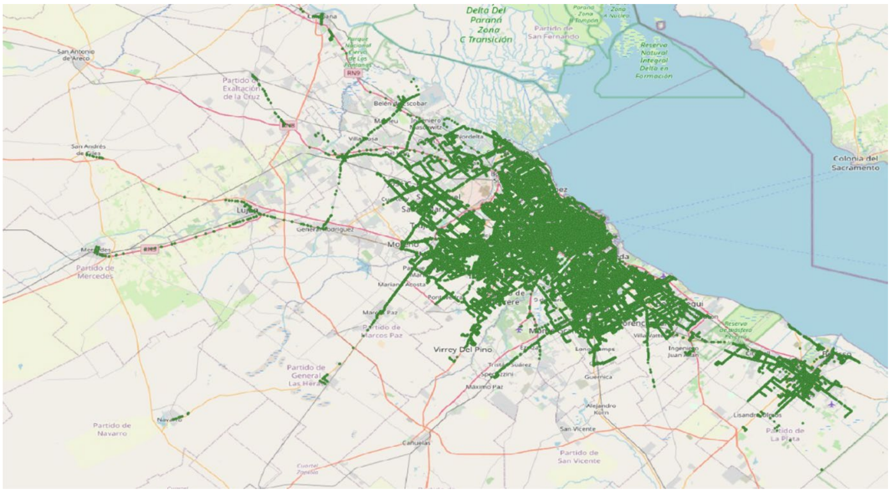


Fig. 12 GTFS Static bus lines trajectories

less than 200 m away from the Colón Theater, a world-famous opera venue. Results are displayed in Figs. 13 and 14. Comparing the density of the trajectories displayed in Fig. 12, and considering that the Colón Theater is located downtown in the city, the radial design of the public bus system becomes evident: a large part of the lines go from the suburbs to downtown Buenos Aires.

Listing 2 Lines that pass at a distance less than 200 m from Colón Theater

```
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;
```

Listing 2 depicts the query, which defines the table `trip_distances`, where 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 `shortest-Line`, which receives a `tgeompoint` and a `geompoint`, and returns the shortest line that connects the two figures. Online web tools are used to find the coordinates of the PoI (in SRID 5345). These coordinates are passed on to the function `ST_MakePoint`. The PostGIS function `ST_Length` retrieves 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. Figure 14 also shows, using a red gradient, the relative average distances between each bus line and the theater.

5.2 GTFS Static and Real-time comparison

Figure 15 shows a portion of the positions of buses registered in the week of 18–25 August. By identifying the individual bus lines with the *trip_id* it is possible to query both, real-time and static feeds, for the different trips of a particular bus line, as shown in Fig. 16. Figure 17 shows the comparison for Line #152A, between real time and planned trajectories. It can be seen that both are similar.

Queries on real-time data stored as continuous data can be efficiently and easily expressed using MobilityDB functions. An example is shown next.

Query 2 Compute the average speed of the vehicles by starting hour and day.

Listing 3 shows the MobilityDB queries that solve the problem.

Listing 3 Average speed of the vehicles by starting hour and 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;
```

With the `trip` field as a `tgeompoint`, the MobilityDB function `twavg()` computes the average speed of a trip. This function receives a list of numbers with a temporal value and computes the time-weighted average of these numbers. By using another MobilityDB function, `speed()`, in conjunction with `twavg()`, the average speed of the trips is obtained. Figure 18 displays the result, comparing real-time and static data. It can be noticed that the buses are moving considerably faster than in the planned itinerary. It also seems that the itinerary does not take into account the traffic congestion changes occurring during the day, while in the real-time data obtained, higher speeds are registered at midnight every day. The average speed of buses oscillates between 18km/h and 27km/h. When grouped by day, in both real-time and itinerary data, the average speed rises on weekends, and remains constant on weekdays. This is expected, since there are less commuters on weekends that may create traffic jams and congestion in the streets (see Agarwal 2004).

Another query, relevant for traffic analysis is shown next.

Query 3 Compute the average delay by bus line.

The query must compute the average delays, grouped by bus lines. This is shown in Listing 4 and results are depicted in Fig. 19.

Listing 4 Average delay by bus line

```
SELECT bl.route_short_name AS BusLine,
  AVG(EXTRACT(EPOCH FROM timespan(rt.trip))/60
  - EXTRACT(EPOCH FROM timespan(st.trip))/60) AS Delay
FROM trips_mdb AS rt
  JOIN trips_mdb_static AS st ON rt.trip_id = st.trip_id
  JOIN buslines AS bl ON rt.trip_id = bl.trip_id
GROUP BY route_short_name
ORDER BY value DESC LIMIT 25;
```

MobilityDB's function `timespan` computes the duration of the trip. Then, the query uses pure PostgreSQL syntax. By calculating the difference between the duration of both `tgeompoints` it is possible to compute the delay for the whole trip, which is depicted in Fig. 19). It can be observed that only two lines have delays with respect to the itinerary. This is coherent with the speed data computed above. Results are expressed in minutes.

Query 4 Compute a heatmap comparing real and planned trajectories for different bus lines.

With the real-time and static `tgeompoints`, the `trip_id` is 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 proximity between spatio-temporal objects, a delay heatmap for every bus in the system is built. Listing 5 shows a portion of the code for this, that is, the queries for the definition and population of the heatmap tables. The result is depicted in Fig. 20 for two different lines. Segments of

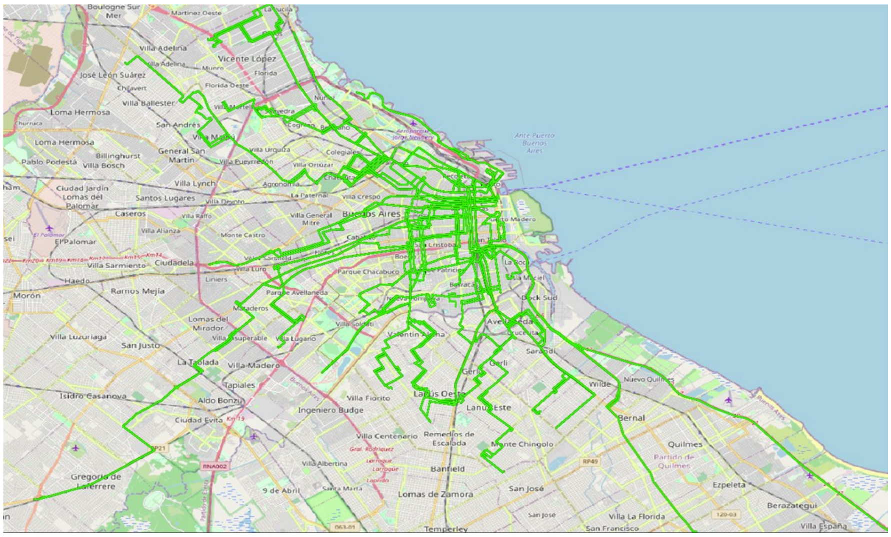


Fig. 13 Bus trajectories close to Colón Theater

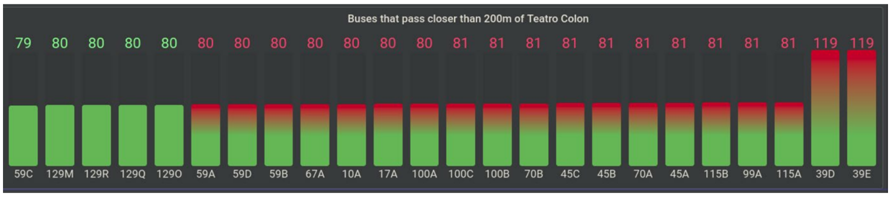


Fig. 14 IDs of bus lines that pass at a distance less than 200m from Colón Theater

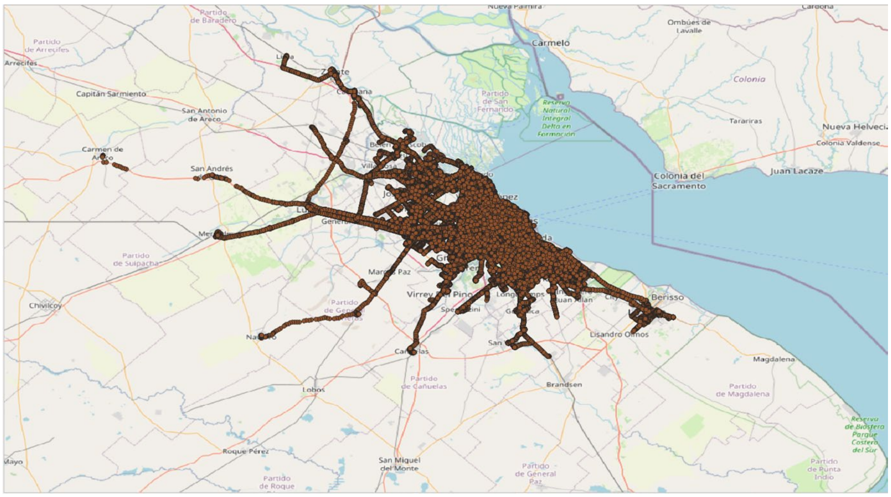


Fig. 15 Actual trajectory of buses

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. Using MobilityDB syntax the segments for which the real-time and static buses were near each other up to a given tolerance are 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 `&&` (overlaps) which launches an index search.

```
mobilitydb=# SELECT startTimestamp(trip), endTimestamp(trip) FROM trips_mdb
mobilitydb=# 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
mobilitydb=# 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)
```

Fig. 16 Comparing static and real-time feeds

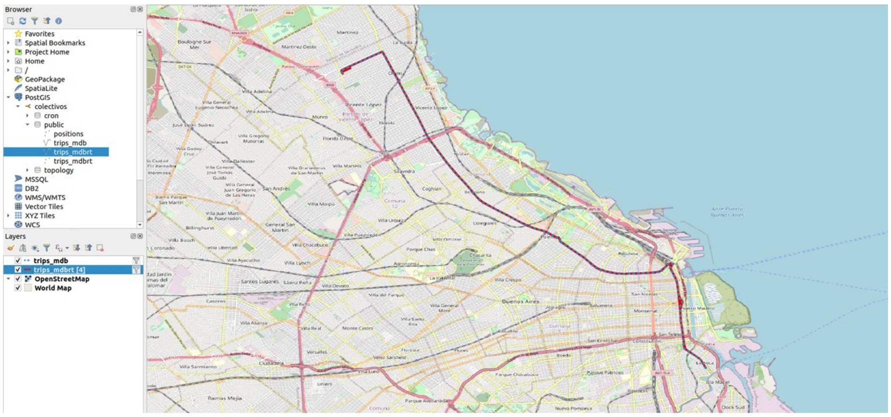


Fig. 17 Bus line #152A trajectory comparison

Five tables are created, each one containing the trip segments for different degrees of tolerance. Once visualized in an application such as QGIS, the stepped-tolerance creates a heatmap-like visualization. These visualizations are shown in Figs. 20 and 21.

Listing 5 Heatmap calculation buses delay

```
-- 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;
...
```

As a final discussion, many visualizations confirm significant differences between the real location of the buses and their static itineraries. These differences are to be expected by any person who uses public transport in her daily life.

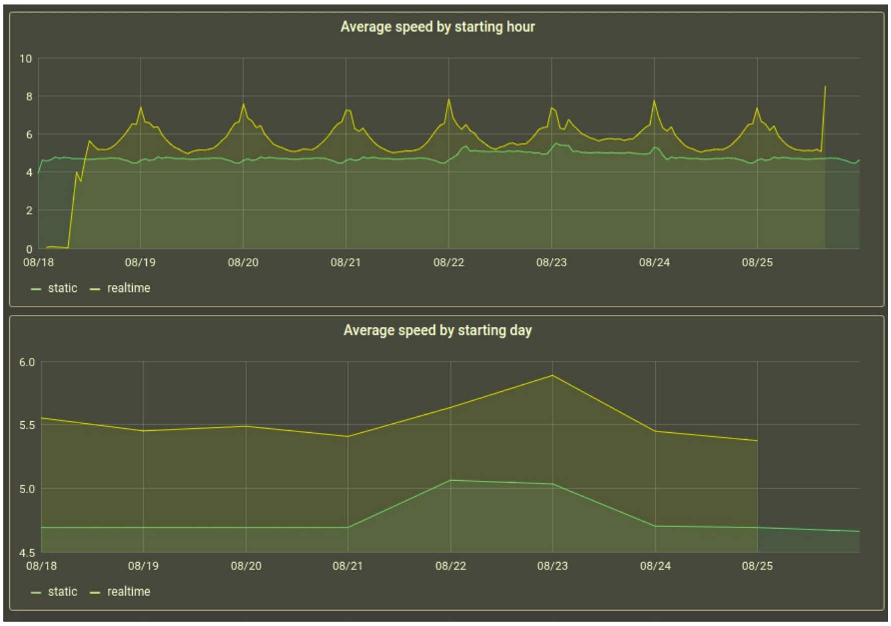


Fig. 18 Average speed grouped by hour and date comparison

However, it is interesting to observe that in the cases reported here, these differences are not delays but, on the contrary, the real bus time is ahead of its itinerary. In any other year this observation would most likely lead to conclude that the data are erroneous. However, due to the extremely unusual events that have taken place in 2020, it is believed that the cause for this result is different. The sampled data corresponds to August in Argentina, when the city of Buenos Aires and its outskirts were going through a strict lockdown due to the SARS-CoV-2 pandemic, except for public transportation. Many studies have already been analyzing the effects of the lockdown on traffic and circulation. Amongst them are those given by the Google Mobility Report, shown in Fig. 22. It can be observed there, that the results obtained in the use case reported here are explained by the traffic reduction in the streets of Buenos Aires.

6 Conclusion

This work shows how MobilityDB, an open-source moving object database, can be used to analyze mobility data, in particular, public transport data. For this, the city of Buenos Aires, Argentina, is used as a case study. Public data about bus, train, and subway schedules are available compliant with the GTFS Static standard. Further, real-time data for buses are also available. These data are captured, preprocessed and loaded into a PostgreSQL database provided with the PostGIS

Fig. 19 Average delay 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
```

and MobilityDB extensions, and used to analyze schedules, itineraries, delays, and other typical questions of interest for transport planners. The processes carried out and software developed for all the tasks involved, like capturing data, are also discussed (software is also publicly available). Further, a novel map matching method is also proposed, that uses MobilityDB to match real-time trajectories to roads using the planned trajectories, and considering that the latter represent the actual roads to which the former ones must be matched.

The processes described in this paper cannot be replicated and generalized for importing any GTFS dataset into MobilityDB, because of the wide spectrum of formats that the GTFS standard allows. However, this study is a valuable

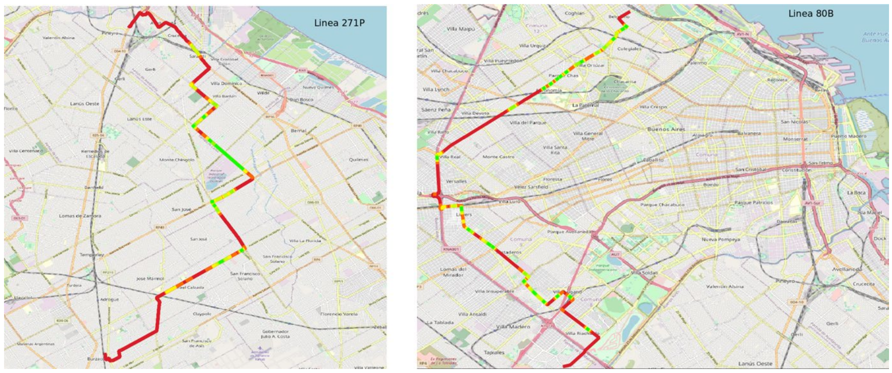


Fig. 20 Heatmaps for bus lines #271P (left) and #80B (right)

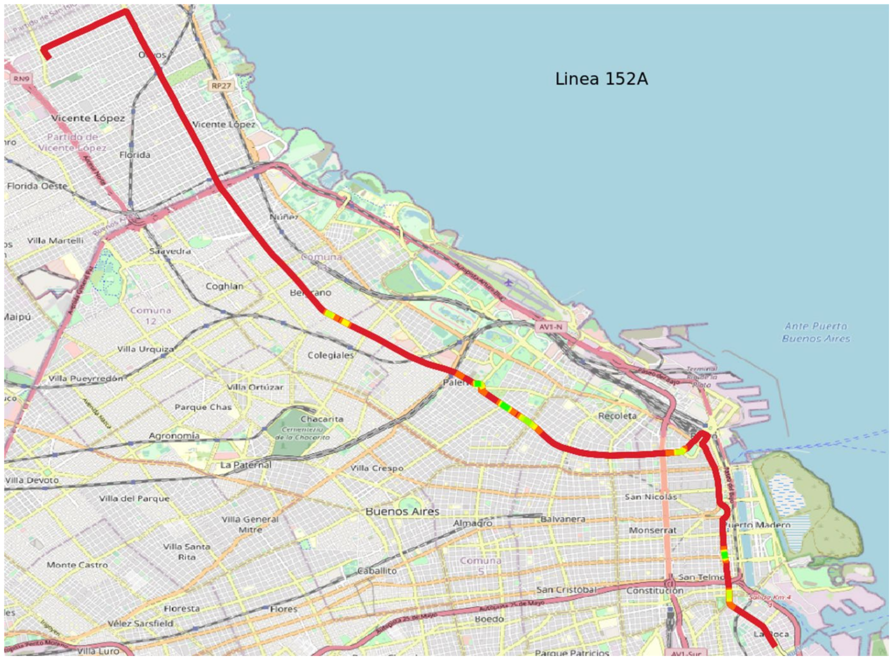


Fig. 21 Heatmaps for bus line 152A

reference for similar cases. The queries displayed in the analysis section illustrate how transit data can be queried in MobilityDB, and displayed using visualization tools. Although Grafana was used in this paper, any other similar tool could be used to visualize the results.

MobilityDB is continuously evolving in several directions. In particular, to be able to process the massive amounts of movement data that are currently being generated, a distributed version of MobilityDB that works in cloud environments

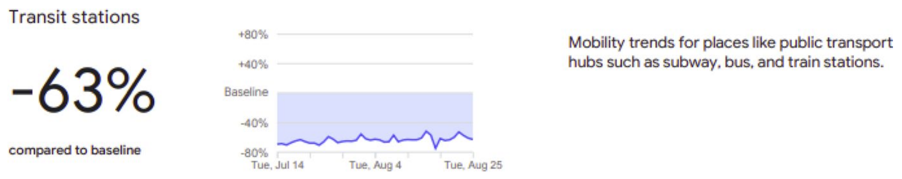


Fig. 22 Google Mobility Report Buenos Aires 25-Aug-2020

such as Azure, AWS, or Google Cloud Platform, is under development. As another direction, GTFS Static represents periodic movement data that are valid during a certain time interval (Behr et al. 2006). The current approach used in this paper requires to "instantiate" these periodic data to represent each actual occurrence. For instance, a service occurring every Monday at 8 am will be replicated for each Monday in the period of analysis. To avoid this, it is planned to implement in MobilityDB a new data type to account for periodic movements.

Acknowledgements Alejandro Vaisman was partially supported by Project PICT 2017-1054, from the Argentinian Scientific Agency.

References

Agarwal A (2004) A comparison of weekend and weekday travel behavior characteristics in urban areas. Master thesis, Department of Civil and Environmental Engineering, University of South Florida. https://digitalcommons.usf.edu/cgi/viewcontent.cgi?article=1935&context=etd

Andersen O, Krogh BB, Thomsen C, Torp K (2014) An advanced data warehouse for integrating large sets of GPS data. In: Proceedings of the 17th international workshop on data warehousing and OLAP, DOLAP '14. ACM, pp 13–22

Bakli MS, Sakr MA, Zimányi E (2019) Distributed moving object data management in MobilityDB. In: Proceedings of the 8th ACM SIGSPATIAL international workshop on analytics for big geospatial data, BigSpatial@SIGSPATIAL 2019. ACM, pp 1:1–1:10

Bast H, Brosi P, Storandt S (2014) Real-time movement visualization of public transit data. In: Proceedings of the 22nd ACM SIGSPATIAL international conference on advances in geographic information systems. ACM, pp 331–340

Behr T, de Almeida VT, Güting RH (2006) Representation of periodic moving objects in databases. In: 14th ACM international symposium on geographic information systems, ACM-GIS 2006, November 10–11, 2006, Arlington. ACM, pp 43–50

Braga M, Santos MY, Moreira AJC (2014) Integrating public transportation data: creation and editing of GTFS data. In: New perspectives in information systems and technologies, vol 2 WorldCIST'14, Advances in Intelligent Systems and Computing, vol 276. Springer, pp 53–62

da Silva MCT, Times VC, de Macêdo JAF, Renso C (2015) SWOT: a conceptual data warehouse model for semantic trajectories. In: Proceedings of the 18th ACM international workshop on data warehousing and OLAP, DOLAP 2015, pp 11–14

de Queiroz ARM, Santos VB, Nascimento DC, Pires CES (2019) Conformity analysis of GTFS routes and bus trajectories. In: XXXIV Simpósio Brasileiro de Banco de Dados, SBBD 2019. SBC, pp 199–204

Fu Z, Tian Z, Xu Y, Qiao C (2016) A two-step clustering approach to extract locations from individual GPS trajectory data. ISPRS Int J Geo-Inf **5**(10):166

Güting RH, Schneider M (2005) Moving objects databases. Morgan Kaufmann, San Francisco

Hohmann S, Geistefeldt J (2016) Traffic flow quality from the user's perspective. Transp Res Proc 15:721–731. International Symposium on Enhancing Highway Performance (ISEHP)

Kaeoruean K, Phithakkitnukoon S, Demissie MG, Kattan L, Ratti C (2020) Analysis of demand-supply gaps in public transit systems based on census and GTFS data: a case study of Calgary, Canada. Public Transp 12(3):483–516. https://doi.org/10.1007/s12469-020-00252-y

Krogh B, Andersen O, Torp K (2012) Trajectories for novel and detailed traffic information. In: Proceedings of the 3rd ACM SIGSPATIAL international workshop on geostreaming, IWGS '12. ACM, pp 32–39

Kunama N, Worapan M, Phithakkitnukoon S, Demissie MG (2017) 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, UbiComp/ISWC. ACM, pp 388–396

Leonardi L, Orlando S, Raffaetà A, Roncato A, Silvestri C, Andrienko GL, Andrienko NV (2014) A general framework for trajectory data warehousing and visual OLAP. GeoInformatica 18(2):273–312. https://doi.org/10.1007/s10707-013-0181-3

Meng C, Yi X, Su L, Gao J, Zheng Y (2017) City-wide traffic volume inference with loop detector data and taxi trajectories. In: Proceedings of the 25th ACM SIGSPATIAL international conference on advances in geographic information systems. ACM. https://doi.org/10.1145/3139958.3139984

Parent C, Spaccapietra S, Renso C, Andrienko G, Andrienko N, Bogorny V, Damiani ML, Gkoulalas-Divanis A, Macedo J, Pelekis N, Theodoridis Y, Yan Z (2013) Semantic trajectories modeling and analysis. ACM Comput Surv 45(4):42:1-42:32

Pelekis N, Theodoridis Y (2014) Mobility data management and exploration. Springer, New York. https://doi.org/10.1007/978-1-4939-0392-4

Pelekis N, Theodoridis Y, Vosinakis S, Panayiotopoulos T (2006) Hermes—a framework for location-based data management. In: Proceedings of the 10th international conference on extending database technology, EDBT 2006, Lecture Notes in Computer Science, vol. 3896. Springer, pp 1130–1134

Renso C, Spaccapietra S, Zimányi E (eds) (2013) Mobility data: modeling, management, and understanding. Cambridge University Press, Cambridge

Spaccapietra S, Parent C, Spinsanti L (2013) Trajectories and their representations. In: Renso C, Spaccapietra S, Zimányi E (eds) Mobility data: modeling, management, and understanding. Cambridge University Press, pp 3–22

Vaisman A, Zimányi E (2014) Data warehouse systems: design and implementation. Springer, New York

Vaisman AA, Zimányi E (2019) Mobility data warehouses. ISPRS Int J Geo-Inf 8(4):170

Vuurstaek J, Cich G, Knapen L, Ectors W, Yasar A, Bellemans T, Janssens D (2020) GTFS bus stop mapping to the OSM network. Future Gen Comput Syst 110:393–406

Wei H, Wang Y, Forman G, Zhu Y (2013) Map matching: comparison of approaches using sparse and noisy data. In: Proceedings of the 21st ACM SIGSPATIAL international conference on advances in geographic information systems. ACM, pp 444–447

Wessel N, Widener MJ (2017) Discovering the space-time dimensions of schedule padding and delay from GTFS and real-time transit data. J Geogr Syst 19(1):93–107

Xu J, Güting RH (2013) A generic data model for moving objects. GeoInformatica 17(1):125–172

Yan Z, Chakraborty D, Parent C, Spaccapietra S, Aberer K (2013) Semantic trajectories: mobility data computation and annotation. ACM Trans Intell Syst Technol 4(3):49:1-49:38

Zimányi E, Sakr MA, Lesuisse A (2020) MobilityDB: a mobility database based on PostgreSQL and PostGIS. ACM Trans Database Syst 45(4):19:1-19:42