Management and analysis of mobility data

Sergi Gil Hernández

Information Technology (IT) Specialty
Informatics Engineering
Thesis in Erasmus+ program

Home Institution: Facultat d'informàtica de Barcelona (FIB)
Home University: Universitat Politècnica de Catalunya (UPC)

Destination University: Università degli Studi di Milano (UNIMI)
Supervisor: Maria Luisa Damiani
Department of Computer Science at Unimi

May 2022

Contents

1 Introduction **6**
1.1 What is a moving object database? . . . 6

2 MobilityDB: introducing the system **8**
2.1 What is MobilityDB? . . . 8
2.1.1 Characteristics . . . 8
2.1.2 Abstract data types . . . 10
2.2 Mobility related operations . . . 13
2.2.1 Constructors . . . 13
2.2.1.1 Tgeompoint_inst() . . . 13
2.2.1.2 Tgeompoint_seq() . . . 13
2.2.2 Accessors . . . 13
2.2.2.1 GetX() . . . 13
2.2.2.2 GetY() . . . 14
2.2.2.3 SRID() . . . 14
2.2.2.4 StartTimestamp() . . . 14
2.2.2.5 EndTimestamp() . . . 14
2.2.2.6 Timespan() . . . 14
2.2.2.7 Duration() . . . 14
2.2.2.8 Period() . . . 15
2.2.2.9 Length() . . . 15
2.2.2.10 TwAvg() . . . 15
2.2.2.11 Speed() . . . 15

3 Experimental part: evaluating the system **16**
3.1 Overview . . . 16
3.2 Installation . . . 18
3.2.1 Requirements . . . 18
3.2.2 Building the application . . . 19
3.3 Importing and testing a trajectory data set . . . 20
3.4 Importing and managing mobility data from Google Timeline . . . 22
3.5 Examples of queries . . . 30
3.5.1 Speed function . . . 30
3.5.2 Scenario 1: Tram trip in Milan . . . 31
3.5.2.1 Overview . . . 31
3.5.2.2 Query execution . . . 33
3.5.3 Scenario 2: Plane trip Milan-Amsterdam . . . 37
3.5.3.1 Overview . . . 37
3.5.3.2 Query execution . . . 38
3.5.4 Scenario 3: A visit to a museum . . . 41
3.5.4.1 Overview . . . 41
3.5.4.2 Query execution . . . 43
3.6 Conclusions . . . 46
3.6.1 Complex issues . . . 47

List of Figures

1 Comparison between a trajectory of scattered points and concentrated points . . . 9
2 The picture shows graphics that exemplify how the different temporal types durations work . . . 13
3 Data names and types inserted into the new table *traceexample* in PgAdmin4 . . . 16
4 New table created called *trips* . . . 16
5 SQL command used to insert the first row into the *trips* table from the (x, y, timestamp) given values . . . 17
6 SQL command used to insert the second row into the *trips* table from the (x_kf_filtered, y_kf_filtered, timestamp) given values 17
7 Filled *trips* final table . . . 18
8 Picture with an example of the data we used as example to practice at first . . . 20
9 Picture map with the relating the position coordinates of the museum's visitor according to its timestamp . . . 21
10 Picture map with the relating the position coordinates (with filtered values) of the museum's visitor according to its timestamp 22
11 Picture map overlapping the map with no filtered coordinate and the one that has Kalman filter applied . . . 22
12 The picture shows the option Location History in the Google Takeout page. . . . 23
13 Here we can see the different options Google Takeout offers us to export the data (frequency of exportation, compression type and size of package). . . . 24
14 In the picture above we can observe the specific content of our Records.json before mentioned. . . . 25
15 A picture of our terminal with the command-line executing the jq command. . . . 25
16 Here we have the content of the folder where we executed the jq command mentioned above and showed in Figure 15. . . . 26
17 In the picture above we can observe the content of the file location_history.csv . . . 26
18 In the picture above we can observe the content of the file location_history0322.csv which reflects the mobility information about March 2022 . . . 27
19 The picture shows a QGIS graphic representation of the data mentioned above about March 2022: a route through Italy. . . . 27
20 The picture shows the location_history0322 database (before doing any changes on it) exported from QGIS system to our PgAdmin4 system. . . . 28
21 SQL query that set values to date row in the location_history0322 table . . . 28

22 SQL query that create an empty table named trips0322 which will be filled with all the trips done in March 2022 (03-2022), organized by days 29
23 SQL query that fills the table trips0322 with all the trips done in March 2022, organized by days 29
24 The content of the table trips0322: all the daily trips ordered by its date. 30
25 The picture shows a graphic representation of what linear interpolation is. 31
26 The picture shows an speed query 31
27 The picture map represents the first scenario: a tram trip in Milan. 32
28 Query to create the table tramTrip which will contain the trip info about Scenario 1. 32
29 Query that inserts the tram trip related info into the table created in Figure 28. 33
30 The query that calculates the start and end timestamps of the trip of the Scenario 1, with both results below the query. 33
31 The query that calculates the period timestamps of the trip of Scenario 1, with the result below the query. 34
32 The query that calculates the duration of the trip of the Scenario 1, with the result below the query. 34
33 The query that calculates the length of the trip of the Scenario 1, with the result below the query. 35
34 The query that calculates the speed value. 35
35 The query that calculates the average speed value, in meters per second. 36
36 The query that calculates the average speed value, and casts it to KmH. 36
37 The picture shows a plane travel someone did from Milan, Italy to Amsterdam, Netherlands on 03-03-2022. 37
38 Query to create the table planeTrip which will contain the trip info about Scenario 2. 37
39 Query that inserts the plane trip related info into the table created in Figure 38. 38
40 The query that calculates the start and end timestamps of the trip of the Scenario 2, with both results below the query. 38
41 The query that calculates the period timestamps of the trip of Scenario 2, with the result below the query. 39
42 The query that calculates the duration of the trip of the Scenario 2, with the result below the query. 39
43 The query that calculates the length of the trip of the Scenario 2, with the result below the query. 40
44 The query that calculates the speed value. 40
45 The query that calculates the average speed value, in meters per second. 41

46 The query that calculates the average speed value, and casts it to KmH. 41
47 Picture map with the relating the position coordinates of the museum's visitor according to its timestamp. 42
48 Picture map with the relating the position coordinates (with filtered values) of the museum's visitor according to its timestamp. 42
49 The query that gets the start and the end timestamps from both trips. 43
50 The query that gets the period from both trips. 43
51 The query that gets the duration from both trips. 44
52 The query to get the length from both trips. 44
53 The query that calculates the speed value. 45
54 The query that calculates the average speed value, in meters per second. 45
55 The query that calculates the average speed value, and casts it to KmH. 46

1 Introduction

Mobility data describe the movement of objects in a time interval, such as vehicles, pedestrians, animals. Commonly, mobility data take the form of spatial trajectories, i.e. sequences of timestamped positions in a coordinated space. For example, GPS trajectories are a popular class of spatial trajectories collected outdoors. In this project, we consider a variety of mobility data, including indoor trajectories collected using a positioning infrastructure based on Ultra-Wide Band (UWB). Goal of the project is to investigate on recent Moving Object database solutions for the management of UWB trajectories with focus on recent commercial databasesfor IOT data. At first, the possible results of this project were:

- A brief survey on databases for the management of trajectories
- Presentation of one specific database for the management of IOT data and experiments with data provided by Unimi.
- Other possible contributions related to the handling of mobility data on top of the chosen database

This research focuses on the second aspect. In particular, the goal is to analyze a recent Moving Object database called MobilityDB [9]. The report is structured as follows:

- Section 2: introduces the MobilityDB system.
- Section 3: reports a few experiments with real outdoor/indoor trajectories. This section concludes the report with some final considerations.

1.1 What is a moving object database?

Moving Object databases handle spatio-temporal data, namely geometries changing over time.

We can distinguish two types of spatio-temporal objects, namely discretely moving objects and continuously moving objects. Discrete moving objects are simple to store in the database. The spatial and temporal properties can be straight forwadly represented by attributes of proper type. However, objects that change position or extent continuously are much more difficult to accommodate in the database. We are mainly interested in this last one type, and how a framework of abstract data types could work for those continuously moving objects [2].

Despite two decades of research in moving object databases and a few research prototypes that have been proposed, there is not yet a mainstream system targeted for industrial use. MobilityDB is an open-source moving object database developed on top of PostgreSQL and PostGIS. In particular, MobilityDB extends the type system with abstract data types for the representation of trajectories and moving entities. The types are fully integrated into the platform to

reuse its powerful data management features. Furthermore, MobilityDB builds on existing operations, indexing, aggregation, and optimization framework. This is all made accessible via the SQL query interface.

2 MobilityDB: introducing the system

2.1 What is MobilityDB?

A trajectory, from a geometrical point of view, can be defined as a set of isolated points in space and their timestamps [5]:

- Point: a 0-dimensional mathematical object which can be specified in n-dimensional space using an n-tuple (x1, x2, ..., xn) consisting of n coordinates. [7]
- Timestamp: a digital record of the time of occurrence of a particular event.

2.1.1 Characteristics

Location tracking devices are being used a lot nowadays, in a lot of day-to-day objects like the GPS in smart phones or in vehicles. Obviously, whenever we have any data collected, we need to store it somewhere. So, that is the main objective of MobilityDB: store that mobility data. MobilityDB provides the support to the database necessary to store and consult those geospatial trajectory data.

MobilityDB is implemented as an extension of PostgreSQL and PostGIS (as we will see deeply in the Installation section) [3]. The main important new characteristic implemented with MobilityDB is that it lets us to work with variable values in function of an another value (i.e. the timestamp one). This characteristic allows that this modern extension (from 2019) implements persistent database types and query operations for managing geospatial trajectories and their time-varying properties.

A geospatial trajectory can be defined as a collection of discrete location points and timestamps, as its illustrated in the top most figure in Figure 1. However, the movement is continuous. MobilityDB interpolates the movement track between the input observations, as illustrated in the figure in the middle in Figure 1. As illustrated in the bottom most figure in Figure 1, this interpolation does not correspond to an increased storage size not even it restores the movement continuity. This process is called normalization and usually results in significant reduction in storage size compared to input points.

MobilityDB provides two new main geospatial trajectory type: tgeompoint (temporal geometry point) and tgeogpoint (temporal geography point). Both of them are useful to represent geospatial points that move over time (timestamp), where the coordinates are expressed like the geometry and geography in PostGIS. We have also available a big set of functions to manage those types. For example, we will see functions about the manage of speed, direction and distance. Obviously, those returned values are temporal ones, like the speed of a car is a temporal value which value depends on the given timestamp specific

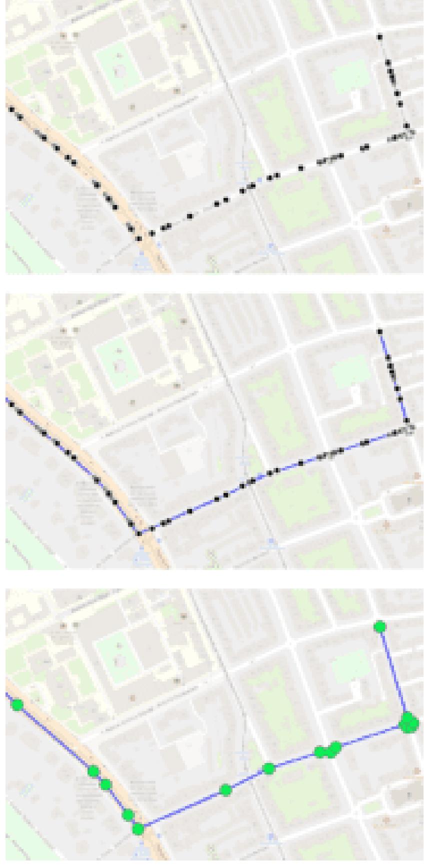


Figure 1: Comparison between a trajectory of scattered points and concentrated points

value. Therefore, we have available the temporal version of the typical main database types, like floats, Booleans, texts and integers.

2.1.2 Abstract data types

Before all, it is crucial to explain some key concept semantics, which is the result of a lifted function obtained by applying static function to each instant of a temporal value. There are different data type semantics:

- Base types: int, real, string and bool. All of them have the usual interpretation, but adding the possibility to set them the undefined value.
- Spatial types: point, line and region. However, we are going to use points type, too. It is quite relevant to mention that the line data type is used to represent a finite union of curves from some class of simple curves. If we talk about using a discrete design in order to implement the abstract design of data types, some class of curves will be selected for representation.
- Time type: instant. It represents a unique point in time or is undefined, otherwise. Time is considered to be linear and continuous.
- Temporal types. From the base types and spatial types, we want to derive corresponding temporal types. The type constructor moving is used for this purpose.
- Range types (sets of intervals). For all those temporal types mentioned before, it would be fine to have operations which correspond to projections into the domain and the range of the functions. For the moving counterparts of the base types, the projections are, or can be, compactly represented as sets of intervals over the one-dimensional domain. Hence, we are also interested in types to represent sets of intervals over the real numbers, over the integers, etc. Such types are obtained through a range type constructor.

Then, it is important to make a brief description of what abstract data types are, so we are going to talk about them a lot since now.

Abstract data types consist of data types and operations that encapsulate the data types. It is important to note that to manage those abstract data types they must be embedded in a query language (about which we are going to talk in a future section), so it will be easier to illustrate the use of the framework we are talking about.

It is important to know that the system we are using (MobilityDB), like all systems, is defined by its data types (abstract data types here) and its related operations (what we are talking now about). The definition of this system consists of two main steps. In the first one, basic types and their constructors are defined, mainly. But it is in the second step where the collection of operations, over those types established in the first step, is designed (defining each

syntax operation, its argument types, etc.). And here, is where are we right now.

One thing to consider is that designing operations in MobilityDB, this design adheres to three principles:

1. Design operations that are as generic as possible.
2. Achieve consistency between operations on non-temporal and temporal types.
3. Capture the interesting phenomena.

The first principle is the most important when the system is quite large. IF we want to avoid the proliferation of operations it is mandatory to find a unifying view of collections of types. The basic approach to achieve this is to relate each type to either a one-dimensional or a two-dimensional space and to consider all values either as single elements or subsets of the respective space. It will be easier to see if we use first various examples:

- The type int describes single elements of the one-dimensional space of integers, while range(int) describes sets of integers.
- The point type describes single elements of two-dimensional space, whereas points, line, and region describe subsets of the two-dimensional space.

The second principle makes reference to the idea of achieving consistency of operations on non-temporal and temporal types, for which we proceed in two steps. We first have to define the operations, and then we systematically extend operations defined in the first step to the temporal variants of the respective types, which is called lifting. Lifted functions are functions that have static counterparts, but because some of the arguments are temporal, the return is also temporal:

- Static: st_intersects: geometry x geometry -> bool
- Lifted:

  tintersects: tgeompoint x geometry -> tbool

  tintersects: tgeompoint x tgeompoint -> tbool

Third, in order to obtain a powerful query language, it is necessary to include operations that address the most important concepts from various domains. There is no clear recipe to achieve closure of interesting phenomena; nevertheless, this should not keep us from having concepts and operations such as distance, size of a region, relationships of boundaries, etc.

As we mentioned before, we have different data types. So as we have different data types, we can distinguish between operations on non-temporal types and

operations on temporal types (which are our study focus for now).

It is obvious that we want to focus on moving objects, but it is crucial to be aware about non-temporal types (non-moving objects), so later all these operations will be lifted into operations on temporal types as well. We deal with five different one-dimensional spaces called Integer, Boolean, etc. For example, the two types belonging to data space Integer are int and range(int), depending if we are referring to a single object or a set.

Some operations are restricted to certain classes of spaces:

- 1D = {Integer, Boolean, String, Real, Time}
- 2D = {2D}
- 1Dcont = {Real, Time}
- 1Dnum = {Integer, Real, Time}
- cont = {Real, Time, 2D}

As a curious fact, a single operation may have several functionalities (signatures). Sometimes, for a generic operation, there exist more appropriate names for arguments of more specific types. For example, there is a size operation for any point set type; however, for type periods, it makes more sense to call this size duration. In such a case, we introduce the more specific name as an alias with the notation size[duration].

One of the most important aspects when operating on non-temporal types on different spaces is that depending on its dimensional aspects: one-dimensional (one single point set type) or two-dimensional spaces (three point set types). This needs an analysis about which argument type combinations return the most interesting results and what are the result types it returns.

On the other hand, and what we are interested in: operations on temporal types. Some of the most relevant temporal types are tfloat, tbool, ttext, tgeompoint and tgeogpoint. As can be seen, the temporal types are usually the same as non-temporal types but adding a 't' at the beginning.

It is interesting to say that temporal types come in four durations: instant, instantSet, sequence and sequenceSet:

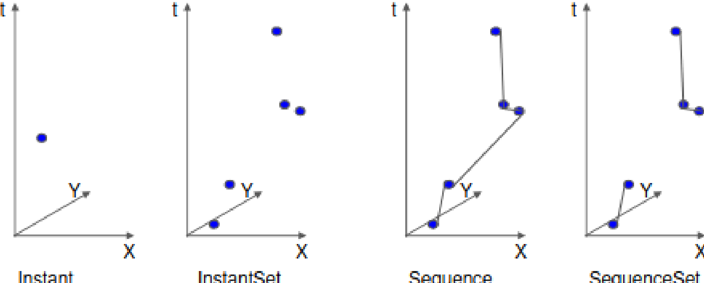


Figure 2: The picture shows graphics that exemplify how the different temporal types durations work

Once introduced a few key ideas, it is time to describe some interesting and useful functions, in order to exemplify later what we have been talking about above.

2.2 Mobility related operations

2.2.1 Constructors

Each temporal type has a constructor function whose name is composed by the name of the type and a suffix that varies depending on the subtype we are referring to ("_inst" for an instance, "_instset" for set of instances, "_seq" for a sequence and "_seqset" for a set of sequences). Here we have few constructor functions:

2.2.1.1 Tgeompoint_inst()

This function is a temporal data type constructor that generates a temporal geometric 2D point of an instant subtype. Its definition is the following:

tgeompoint_inst(tpoint, timestamptz): tgeompoint_inst

2.2.1.2 Tgeompoint_seq()

This function is a temporal data type constructor that generates a temporal geometric 2D point of a sequence subtype. Its definition is the following:

tgeompoint_seq(tpoint, timestamptz[]): tgeompoint_seq

2.2.2 Accessors

2.2.2.1 GetX()

A function to get the X coordinate values as a temporal float (tfloat). Its definition is the following:

getX(tpoint): tfloat

2.2.2.2 GetY()

A function to get the Y coordinate values as a temporal float (tfloat). Its definition is the following:

getY(tpoint): tfloat

2.2.2.3 SRID()

This function returns the spatial reference identifier. Its definition is the following:

SRID(tpoint): integer

2.2.2.4 StartTimestamp()

A function for getting the timestamp when the given trip began. Its definition is the following:

startTimestamp(ttype): timestamptz

2.2.2.5 EndTimestamp()

A function for getting the timestamp when the given trip ended. Its definition is the following:

endTimestamp(ttype): timestamptz

2.2.2.6 Timespan()

Useful to get the timespan ignoring the potential time gaps. Its definition is the following:

timespan(ttype): interval

2.2.2.7 Duration()

A function to get the time spent in a given route (using the first and the last timestamps as references). Its definition is the following:

duration(ttype): interval

2.2.2.8 Period()

A function that return the period on which the temporal value is defined ignoring the potential time gaps. Its definition is the following:

period(ttype): period

2.2.2.9 Length()

This function returns the total length of a temporal point. Since the coordinates of the points are stored in meters, this length will also be returned in meters. Its definition is the following:

length(tpoint): float

2.2.2.10 TwAvg()

This function can be used to get the time-weighted average. It is useful if we combine it with functions like the basic speed function so we can obtain an average speed as a real number. Its definition is the following:

twAvg(tnumber): float

2.2.2.11 Speed()

One of the most interesting functions, so it calculates the speed in a given temporal point (tpoint) in units per second, and its notation is:

speed(tpoint): tfloat

3 Experimental part: evaluating the system

3.1 Overview

For this part, we will use the PgAdmin4 program to manage all the data from QGIS and execute the different necessary SQL commands. So, first of all we upload the data mentioned in Figure 8 into a base table in PgAdmin4, called *traceexample* in our case (see the following Figure 3).

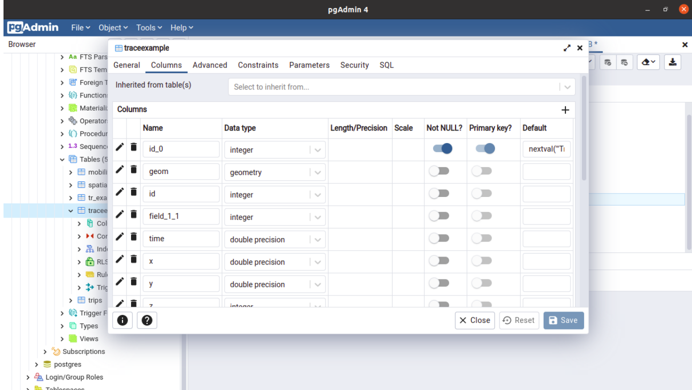


Figure 3: Data names and types inserted into the new table *traceexample* in PgAdmin4

Now, we create another table called *trips* where we are going to introduce the managed data from the *traceexample* table, with a single column:

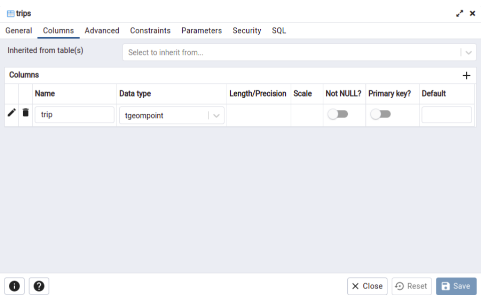


Figure 4: New table created called *trips*

The idea is to have in this new table, two rows of the *tgeompointseq* data type which reflects the potential of MobilityDB new characteristics for treating with temporal mobility data. But for it, we need to manage the data from the table *traceexample* in order to convert it into temporal data type and introduce it into the *trips* table.

What we are doing next is to open the SQL editor in PgAdmin4 and begin moving and managing these data. For it, it is important to take a look about some functions, so the process we are going to do now uses 4 functions to build the tgeompoint sequence from the x,y,timestamp relation (the 2 most inner ones are native of PostGIS, but the 2 other ones are newly incorporated from the MobilityDB extension):

- ST_MakePoint(float x, float y): a function to make points with XY coordinates. In realty, PostGIS gives us the possibility to make points with until 4 coordinates (XYZM) just using ST_MakePoint(float x, float y, float z, float m) instead of the one mentioned before.
- ST_SetSRID(geometry geom, integer srid): a function that sets the SRID on a geometry to a particular value. In our case the *geometry geom* field is composed by the result obtained of executing ST_MakePoint, and the *integer srid* is given by the timestamp obtained from the main base table, *traceexample*.
- tgeompoint_inst: generates a temporal geometric 2D point of instant subtype.
- tgeompoint_seq: generate a temporal geometric 2D point of sequence subtype

With those functions explained, the following picture shows the command used to insert the data from the *traceexample* table, once managed and adapted to the temporal new data type, into *trips*:

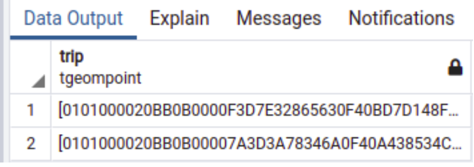
```
1  SELECT tgeompoint_seq(array_agg(tgeompoint_inst(
2      ST_SetSRID(ST_MakePoint(x,y), 3003), timestamptz_) ORDER
3  BY timestamptz_))
4  FROM traceexample;
```

Figure 5: SQL command used to insert the first row into the *trips* table from the (x, y, timestamp) given values

```
6  INSERT INTO trips
7  SELECT tgeompoint_seq(array_agg(tgeompoint_inst(
8      ST_SetSRID(ST_MakePoint(x_kf_fast,y_kf_fast), 3003), timestamptz_) ORDER
9  BY timestamptz_))
10 FROM traceexample;
```

Figure 6: SQL command used to insert the second row into the *trips* table from the (x_kf_filtered, y_kf_filtered, timestamp) given values

With both of inserts, we have filled as we wanted the *trips* table with the 2 rows we expected to, as seen in the Figure 7.



Data Output | Explain | Messages | Notifications

	trip tgeompoint
1	[0101000020BB0B0000F3D7E32865630F40BD7D148F...
2	[0101000020BB0B00007A3D3A78346A0F40A438534C...

Figure 7: Filled *trips* final table

3.2 Installation

3.2.1 Requirements

First of all, we have to check the requirements for using the MobilityDB extension, so we go checking what its main page says about its installation: `https://www.mobilitydb.com/install.html`. In our case, we took the Linux installation way, and we built all on Ubuntu 20.04.3 LTS Focal Fossa, and for it we have the next requirements [8]:

1. PostgreSQL version > 10. We took PostgreSQL v11. It is important to respect the minimmal version needed so sometimes there are patchs on new versions that "correct" something in the previous version but the software we are trying to install (MobilityDB in our case) is demanding for those data before being patched.

2. CMake version >= 3.1. We took CMake v3.16.3. It will be necessary to build up the MobilityDB extension in the future. We installed it using the Linux native terminal with the following command line:

   $sudo apt-get install cmake

3. PostGIS version 2.5. As we said in the Characteristics section above, MobilityDB is built above PostgreSQL and PostGIS.

4. JSON-C. It implements a reference counting object model that allows you to easily construct JSON objects in C, output them as JSON formatted strings and parse JSON formatted strings back into the C representation of JSON objects. It aims to conform to RFC 7159. Like CMake, we used the following command line:

   $sudo apt-get install libjson-c-dev

5. The GNU Scientific Library (GSL). The GSL is a numerical library for C and C++ programmers. It is free software under the GNU General Public License. The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. It only was needed the following command line:

   $sudo apt-get install libgsl-dev

6. The development files for PostgreSQL, PostGIS/liblwgeom, PROJ & JSON-C, Protobuf-C. In our case, we built the following command to install all of these MobilityDB build dependences:

   $sudo apt install build-essential cmake postgresql-server-dev-11 liblwgeom-dev libproj-dev libjson-c-dev libprotobuf-c-dev

3.2.2 Building the application

Now, we had all the requirements installed properly on our system, it was time to begin building up MobilityDB and its installation. For it, like for the requirements, we have a step-by-step, all of them are command lines that were built through our Linux terminal:

1. In our case, as we departed from the minimal installation package version of our Ubuntu, it was necessary first to install the git package:

   $sudo apt-get install git

   The installation of git package is not detailed in the guide because it is considerated obvious that it is necessary so it is used in the first detailed command line:

   $git clone https://github.com/MobilityDB/MobilityDB

2. *$sudo mkdir MobilityDB/build*

3. *$sudo cd MobilityDB/build*

4. At this step, we needed to install first the geos lib in order to be able to compile correctly:

   $sudo apt-get install libgeos-dev

   After that, we just followed the step with:

   $sudo cmake ..

5. *$sudo make*

6. *$sudo make install*

7. First of all we had to activate the extension of the MobilityDB we installed previously:

   $psql -c 'CREATE EXTENSION MobilityDB CASCADE'

   Once we added the MobilityDB extension to our PostgreSQL system, we had to add a non-default user (different than the postgres, which is the default one), so it will be necessary to run some important features. For doing it, we just need to build the following commands on the terminal.

(a) $sudo su -postgres
(b) $createuser –superuser [username]
(c) $createdb [username]
(d) $exit

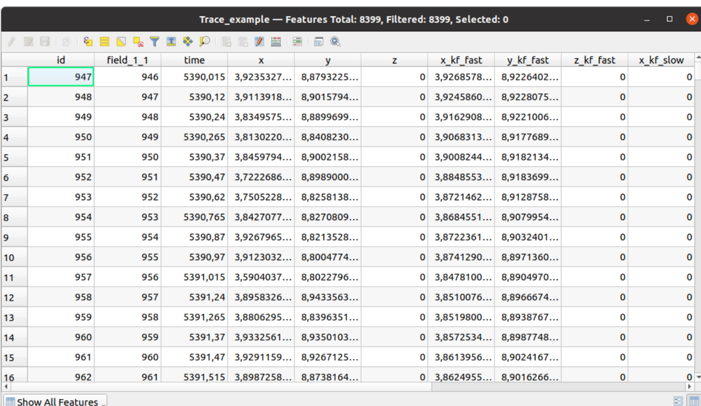
8. As an extra step, we have to add some parameters in the postgresql.conf file (path: /etc/postgresql/11/main/postgresql.conf):

   (a) *shared_preload_libraries = 'postgis-2.5'*
   (b) *max_locks_per_transaction = 128*

It is important to focus on the idea we are going to use PgAdmin4 in order to manage all the data of PostGIS. Because of it, it was necessary to activate the MobilityDB extension in the point 7 of the Building the application section.

3.3 Importing and testing a trajectory data set

First all, for practicing at first and getting familiar with the environment, depart from some data obtained from a trajectory that an specific person did on his visit to a museum.



Trace_example — Features Total: 8399, Filtered: 8399, Selected: 0

	id	field_1_1	time	x	y	z	x_kf_fast	y_kf_fast	z_kf_fast	x_kf_slow
1	947	946	5390,015	3,9235327...	8,8793225...	0	3,9268578...	8,9226402...	0	0
2	948	947	5390,12	3,9113918...	8,9015794...	0	3,9245860...	8,9228075...	0	0
3	949	948	5390,24	3,8349575...	8,8899699...	0	3,9162908...	8,9221006...	0	0
4	950	949	5390,265	3,8130220...	8,8408230...	0	3,9068313...	8,9177689...	0	0
5	951	950	5390,37	3,8459794...	8,9002158...	0	3,9008244...	8,9182134...	0	0
6	952	951	5390,47	3,7222686...	8,8989000...	0	3,8848553...	8,9183699...	0	0
7	953	952	5390,62	3,7505228...	8,8258138...	0	3,8721462...	8,9128758...	0	0
8	954	953	5390,765	3,8427077...	8,8270809...	0	3,8684551...	8,9079954...	0	0
9	955	954	5390,87	3,9267965...	8,8213528...	0	3,8722361...	8,9032401...	0	0
10	956	955	5390,97	3,9123032...	8,8004774...	0	3,8741290...	8,8971360...	0	0
11	957	956	5391,015	3,5904037...	8,8022796...	0	3,8478100...	8,8904970...	0	0
12	958	957	5391,24	3,8958326...	8,9433563...	0	3,8510076...	8,8966674...	0	0
13	959	958	5391,265	3,8806295...	8,8396351...	0	3,8519800...	8,8938767...	0	0
14	960	959	5391,37	3,9332561...	8,9350103...	0	3,8572534...	8,8987748...	0	0
15	961	960	5391,47	3,9291159...	8,9267125...	0	3,8613956...	8,9024167...	0	0
16	962	961	5391,515	3,8987258...	8,8738164...	0	3,8624955...	8,9016266...	0	0

Show All Features

Figure 8: Picture with an example of the data we used as example to practice at first

In the Figure 8 we have the data used at first to begin practicing with this new type of managing mobility data. Among other parameters, we could observe and comment some interesting fields:

- id: identifier of the row object.
- time: a double type value, that represents the moment when the point with coordinates (x, y) is taken. Not confuse it with the field *timestamp* that, even if it does not appear explicitly in the image mentioned above, it is there in the table.

- x: first coordinate of the point represented by the row.
- y: second coordinate of the point represented by the row.
- z: third coordinate of the point represented by the row. It is curious to focus on the fact that its value is always 0 in this table, because we are talking about a 2D trajectory, so it will only have values on X and Y coordinates, but we have the possibility to have a 3r coordinate (the Z one).
- x_kf_fast: its the X coordinate passed by the Kalman filter.
- y_kf_fast: its the Y coordinate passed by the Kalman filter.
- timestamp_: as we mentioned above, when we talked about the *time* field seen in the Figure 8, it is different, so it is a *datatime* type value, instead of the *double* type of the *time* field.

One we have defined the most important variables used in the practice example, we have to build it on PostGIS to get a summary image of the trajectory the visitors did in the museum. The trajectory map relates all the points (x,y) according to its specific timestamp value. All of that is reflected in the image below:

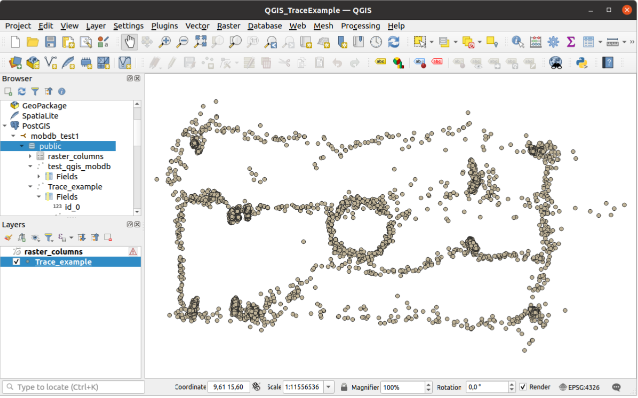


Figure 9: Picture map with the relating the position coordinates of the museum's visitor according to its timestamp

Thanks to this map picture it is obvious to see that the visitor stopped in front of some of the art objects, giving us relevant information such as those pieces of art could have been more attractive to him than the others.

As we mentioned when talking about the variables, there is a type of (x,y) coordinates, passed by the Kalman filtered, so let's see what is the difference if we print the map once again but with those filtered coordinate values:

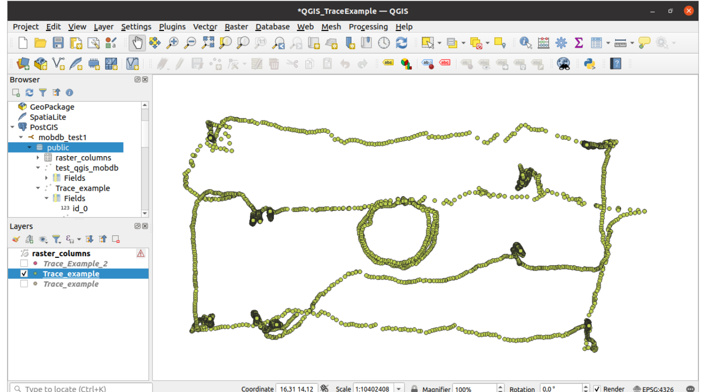


Figure 10: Picture map with the relating the position coordinates (with filtered values) of the museum's visitor according to its timestamp

It is obvious that the values are not the same, we can easily observe that the points of the trajectory printed are not too much scattered in comparison to what we have in the Figure 9.

Taking profit from a feature of QGIS, we are able to overlap both trajectory maps in order to see how different are them:

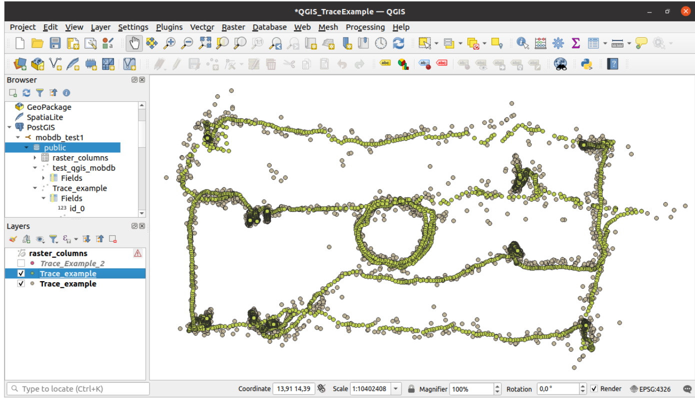


Figure 11: Picture map overlapping the map with no filtered coordinate and the one that has Kalman filter applied

3.4 Importing and managing mobility data from Google Timeline

Google Maps Timeline is a tool that show an estimate of places that we may have been and routes that we may have taken based on our Location History.

So we to export mobility data from there in order to use it as the database for querying is necessary to do it this way:

1. We downloaded our data using the Google Takeout tool [6]. At first, we logged in the Google Takeout page and marked only the Location History option to in order to only download its information (Figure 12).

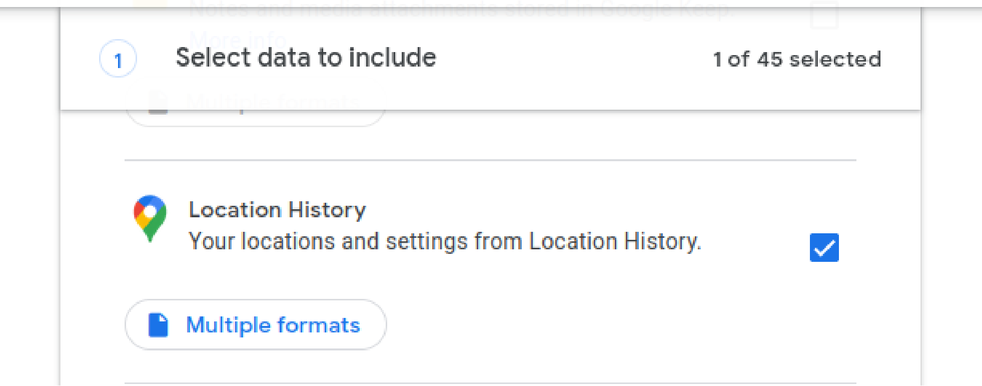


Figure 12: The picture shows the option Location History in the Google Takeout page.

Next, we chose to export this selected data, and it is downloaded in a .zip format (Figure 13).

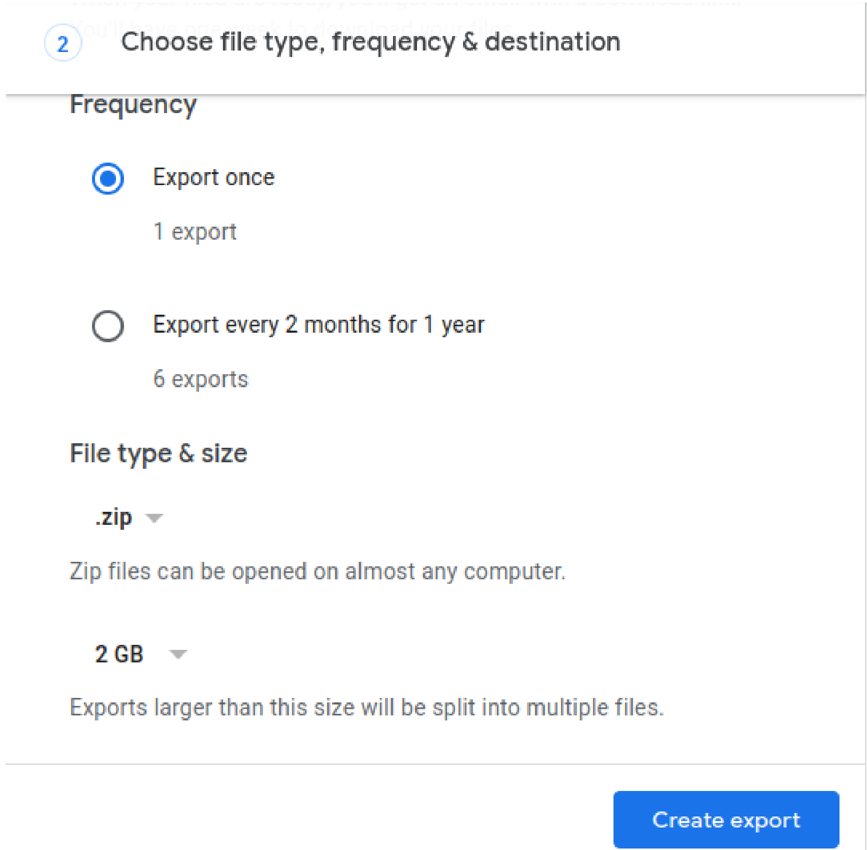


Figure 13: Here we can see the different options Google Takeout offers us to export the data (frequency of exportation, compression type and size of package).

2. Once the zip is exported, it is time to go to the folder where it has been downloaded and unzip it to extract its content. There, we will find a folder named Takeout followed by another folder named Location History which contains the most interesting file to obtain our data: Records.json (Figure 14. It contains a relation of a lot of aspects of the different location points in a given time.

```
1 {
2   "locations": [{
3     "latitudeE7": 414150942,
4     "longitudeE7": 21647770,
5     "accuracy": 696,
6     "activity": [{
7       "activity": [{
8         "type": "TILTING",
9         "confidence": 100
10      }],
11      "timestamp": "2015-11-13T15:11:43.469Z"
12    }],
13    "source": "CELL",
14    "deviceTag": 1436890739,
15    "timestamp": "2015-11-13T15:13:16.207Z"
16  }, {
17    "latitudeE7": 414133075,
18    "longitudeE7": 21685435,
19    "accuracy": 30,
20    "activity": [{
21      "activity": [{
22        "type": "STILL",
23        "confidence": 46
24      }, {
25        "type": "UNKNOWN",
26        "confidence": 29
27      }, {
28        "type": "IN_VEHICLE",
29        "confidence": 20
30      }, {
31        "type": "ON_FOOT",
32        "confidence": 3
33      }, {
```


Figure 14: In the picture above we can observe the specific content of our Records.json before mentioned.

But in our case, we want to load location information into MobilityDB, so we only need the fields longitudeE7, latitudeE7 and timestamp [1]. In order to convert our JSON data file into a CSV file with those 3 needed fields, so we can introduce them into MobilityDB as our database, we proceed using the following jq command-line for processing JSON files (Figure 15).

```
sergi@sergi-HP-Notebook:~/Documents/LocationMapsGoogle/Takeout/Historial de ubicaciones$
cat Records.json | jq -r ".locations[] | {latitudeE7, longitudeE7, timestamp} | [.latitud
eE7, .longitudeE7, .timestamp] | @csv" > location_history.csv
```

Figure 15: A picture of our terminal with the command-line executing the jq command.

```
sergi@sergi-HP-Notebook:~/Documents/LocationMapsGoogle/Takeout/Historial de ubicaciones$
ls -la
total 646244
drwxrwxr-x  3 sergi sergi      4096 may 11 18:12  .
drwxrwxr-x  3 sergi sergi      4096 may  9 21:21  ..
-rw-rw-r--  1 sergi sergi  33141321 may 12 19:40  location_history.csv
-rw-rw-r--  1 sergi sergi 628585434 may  9 12:15  Records.json
drwxrwxr-x 10 sergi sergi      4096 may  9 21:21 'Semantic Location History'
-rw-rw-r--  1 sergi sergi      2511 may  9 12:15  Settings.json
```

Figure 16: Here we have the content of the folder where we executed the jq command mentioned above and showed in Figure 15.

So as we can see in Figure 16 we have now created a file named location_history.csv, whose content is showed below, in Figure 17

```
sergi@sergi-HP-Notebook:~/Documents/LocationMapsGoogle/Takeout/Historial de ubicaciones$
more location_history.csv
414150942,21647770,"2015-11-13T15:13:16.207Z"
414133075,21685435,"2015-11-13T15:14:07.162Z"
414132802,21685506,"2015-11-13T15:16:43.345Z"
414132836,21685705,"2015-11-13T15:18:43.828Z"
414133066,21685439,"2015-11-13T15:20:23.138Z"
414133070,21685390,"2015-11-13T15:22:44.530Z"
414132928,21685257,"2015-11-13T15:26:45.250Z"
414133155,21685640,"2015-11-13T15:29:10.268Z"
414132774,21685273,"2015-11-13T15:31:33.661Z"
414132881,21685545,"2015-11-13T15:32:26.127Z"
414132823,21685624,"2015-11-13T15:33:31.485Z"
414132626,21685214,"2015-11-13T15:35:31.444Z"
414132759,21684978,"2015-11-13T15:39:32.096Z"
414133048,21685450,"2015-11-13T15:41:21.496Z"
414136355,21689666,"2015-11-13T15:43:15.169Z"
414129239,21699352,"2015-11-13T15:45:23.666Z"
414116175,21715218,"2015-11-13T15:47:23.697Z"
414113951,21717269,"2015-11-13T15:48:23.685Z"
414109249,21738765,"2015-11-13T15:50:59.164Z"
414117900,21750954,"2015-11-13T15:53:27.991Z"
```

Figure 17: In the picture above we can observe the content of the file location$_{h}istory.csv$

3. Now we can import the generated CSV file into PostgreSQL. But first, we imported from Google Timeline all the mobility data in around 6 years (as showed in Figure 17 whose first timestamp begins with 2015-11-13 that refers to 13th November 2015), so for this example guide we decided to limit it to manage less data at the same time. We chose March 2022 because in that month there was a plane travel from Italy to Netherlands (at the beginning of the month), a train travel through Italy (the last weekend of the month) and a plane travel from Italy to Spain (last day of the March: 2022-03-31). We thought that would be enough to exemplify our functions of mobility management. In consequence, we are going to work with location_history0322.csv whose location information only refers to March 2022 (as showed in Figure 18 where the first timestamp variable begins with 2022-05-01 that refers to the 1st March 2022).

```
sergi@sergi-HP-Notebook:~/location_history$ more location_history0322.csv
454755329,92487810,2022-03-01T00:05:30.633Z
454755339,92487778,2022-03-01T00:41:38.633Z
454755408,92487857,2022-03-01T01:15:32.230Z
454755376,92487793,2022-03-01T01:19:59.631Z
454754773,92490060,2022-03-01T01:26:14.734Z
454756633,92490939,2022-03-01T01:28:23.620Z
454755156,92491014,2022-03-01T01:30:25.614Z
454755637,92490074,2022-03-01T01:32:27.532Z
454756433,92491186,2022-03-01T01:34:29.695Z
454756433,92491186,2022-03-01T01:36:29.812Z
454756433,92491186,2022-03-01T01:38:29.822Z
454714448,92524921,2022-03-01T01:40:30.945Z
454714448,92524921,2022-03-01T01:42:37.799Z
454757773,92496086,2022-03-01T01:47:40.594Z
454757785,92496082,2022-03-01T02:56:37.114Z
454757785,92496082,2022-03-01T03:35:42.175Z
454757785,92496082,2022-03-01T04:11:44.904Z
454757785,92496082,2022-03-01T04:47:47.768Z
454757785,92496077,2022-03-01T05:01:56.449Z
454757370,92496869,2022-03-01T05:38:16.723Z
454757370,92496869,2022-03-01T06:15:32.876Z
```

Figure 18: In the picture above we can observe the content of the file location_history0322.csv which reflects the mobility information about March 2022

Then, it is already the time to import the data to our PgAdmin4 in order to manage and analyse it with the MobilityDB functions. For it, and for doing it as easy and fast as possible, we will use QGIS software to import this data to our PgAdmin4 (Figure 19).

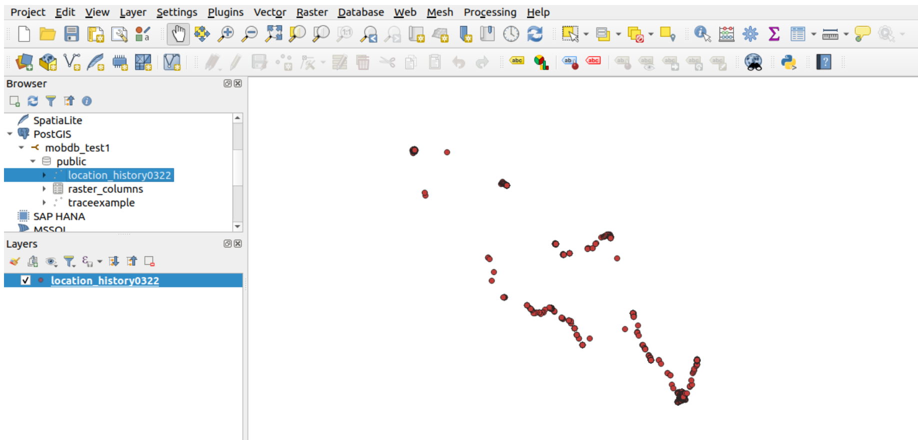


Figure 19: The picture shows a QGIS graphic representation of the data mentioned above about March 2022: a route through Italy.

To import the data from the QGIS software (Figure 19) to PgAdmin4 software, we just need to move the location_history0322 Layer, to the public

database shown in PostGIS Browser above Layers section in QGIS. The result will be a database named as the layer moved (location_history0322) in our PgAdmin4 system (Figure 20).

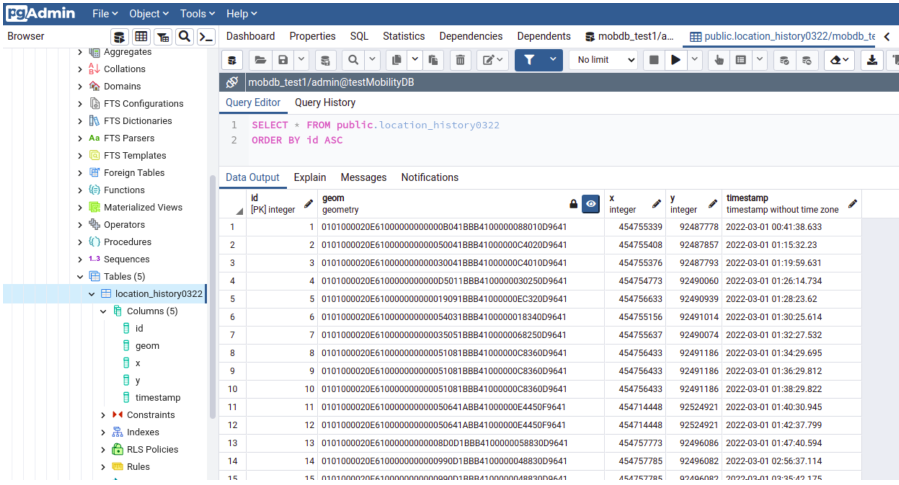


Figure 20: The picture shows the location_history0322 database (before doing any changes on it) exported from QGIS system to our PgAdmin4 system.

4. Then, in order to manage and analyze the values obtained from Google Timeline, we need to cast them into the proper datatypes (longitude and latitude values into standard coordinates values, and timestamp with no time zone into a timestamp with time zone). The change of timestamp into timestamptz (time zone) is easily done by the menu interface, with no SQL command needed. The datatypes of longitude and latitude will be changed when managing them as coordinates later.

5. We create the attribute date (date datatype). Then, in order to organize all the monthly data (31 days in March), we first relate each row value with the specific day it was produced using the SQL query showed in Figure 21.

Query Editor Query History

```
1  UPDATE location_history0322
2  SET date_ = date(timestamptz_);
```

Figure 21: SQL query that set values to date row in the location_history0322 table

6. Now, we need to create the new trip table we are going to use to store the daily trips (1 daily trip per row), as showed in Figure 22.

Query Editor Query History

```sql
CREATE TABLE trips0322 (
date date NOT NULL,
trip tgeompoint,
trajectory geometry,
PRIMARY KEY (date)
);
```

Figure 22: SQL query that create an empty table named trips0322 which will be filled with all the trips done in March 2022 (03-2022), organized by days

7. With the trips0322 table created, it is time to fill it properly. We do it with a similar SQL query we used in Figure 5, but adapting its structure to ours, so now it is a bit more complicated because we are using longitude and latitude coordinates on the Earth's surface (and as defined in the WGS84 standard, also used for the GPS systems, requires us to use the SRID 4326, instead of the 3003, that refers to Italy zone 1, we used in the example in Figure 5). And as we want to organize it for days, we need to use the SQL query showed in Figure 23

Query Editor Query History

```sql
INSERT INTO trips0322(date, trip)
SELECT date_, tgeompoint_seq(array_agg(tgeompoint_inst(
    ST_SetSRID(ST_Point(x/1e7, y/1e7),4326), timestamptz_) ORDER BY timestamptz_))
FROM location_history0322
GROUP BY date_;

```

Figure 23: SQL query that fills the table trips0322 with all the trips done in March 2022, organized by days

As a result of the SQL query showed above in Figure 23, we got the following result table (Figure 24):

Data Output | Explain | Messages | Notifications

	date [PK] date	trip tgeompoint
1	2022-03-01	[0101000020E6100000D6517A4BDEBC464010F1C5CD5F7F2240@2022-03-01 00:41:38.633+01, (
2	2022-03-02	[0101000020E61000002B93D04CE6BC46407AA05FA5CC7F2240@2022-03-02 00:01:25.049+01, (
3	2022-03-03	[0101000020E61000006D73637AC2BD4640ACEBAB06BC5F2240@2022-03-03 00:01:24.293+01,
4	2022-03-04	[0101000020E6100000AA4C8CC0B32E4A40B239BD416A891340@2022-03-04 00:02:50.738+01,
5	2022-03-05	[0101000020E610000019D1D105502F4A4062AEFB7DB5911340@2022-03-05 00:00:41.98+01, 01
6	2022-03-06	[0101000020E61000007F8978EBFC2F4A40D5F83BCA77971340@2022-03-06 00:09:52.222+01, (
7	2022-03-07	[0101000020E610000008BEB38707304A401FE8B2E265971340@2022-03-07 00:00:17.913+01, 0
8	2022-03-08	[0101000020E6100000EF045669E6BC46404B500D45CB7F2240@2022-03-08 00:01:08.93+01, 01
9	2022-03-09	[0101000020E6100000914F7E41C1BD46402F8507CDAE5F2240@2022-03-09 00:00:52.051+01, (
10	2022-03-10	[0101000020E6100000E95AC52CC5BD4640750B13FC25602240@2022-03-10 00:02:53.753+01, (

Figure 24: The content of the table trips0322: all the daily trips ordered by its date.

8. With all of these queries and procedures we have built a very interesting database (trips0322 table) that contains all the coordinates of a person (and its timestamptz) in a whole month, ordered by days (31 rows). Despite of this, in order to establish more specific databases for exemplifying the functions we are going to explain, we are going to need to create new exact-timestamp interval tables because of the table trip0322 takes into account all the points in the 24h of a day, and for doing tests like calculating the average speed of a plane travel, we need to value only the interval where we were in the plane.

3.5 Examples of queries

3.5.1 Speed function

Probably, the speed function is one of the most interesting functions to analyze because its complexity and utility. Before continuing, let's introduce an important concept named linear interpolation:

Linear interpolation is a method of curve fitting using linear polynomials to construct new data points within the range of a discrete set of known data points. Concretely, linear interpolation on a data set (red points in Figure 25) is defined as the concatenation of linear interpolants (blue lines in Figure 25) between each pair of data points, which results in a continuous curve.

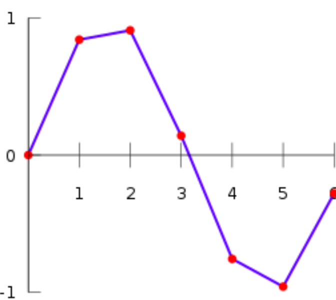


Figure 25: The picture shows a graphic representation of what linear interpolation is.

So, it is vital to know that the data set used as argument in the speed function must have linear interpolation.

```
SELECT speed(tgeompoint '{[Point(0 0)@2000-01-01, Point(1 1)@2000-01-02,
  Point(1 0)@2000-01-03], [Point(1 0)@2000-01-04, Point(0 0)@2000-01-05]}') * 3600 * 24;
-- "Interp=Stepwise;{[1.4142135623731@2000-01-01, 1@2000-01-02, 1@2000-01-03],
  [1@2000-01-04, 1@2000-01-05]}"
SELECT speed(tgeompoint 'Interp=Stepwise;[Point(0 0)@2000-01-01, Point(1 1)@2000-01-02,
  Point(1 0)@2000-01-03]');
-- ERROR:  The temporal value must have linear interpolation
```

Figure 26: The picture shows an speed query

In the picture above we can observe the difference between two speed queries. The first one, calculate properly the speed factor with a properly defined set of points. On the other hand, the second query, the temporal value have no linear interpolation as the error message says.

Once introduced the linear interpolation concept and its importance, it would be interesting to mention the TwAvg function (time-weighted average function) which will help us to transform the speed interpolation calculated by the speed function into an integer value. We are going to explain it deeply in the next section, when exemplifying with the scenarios, so it will be easier to explain the difference with an example.

Then, let's do some querying to go deeply [4]. For executing them, we have used my Google Timeline mobility data (see how we exported it from Google Timeline in Section 3.4)

Taking into account the clarification made in 3.4 point 8, let's see some examples with the speed function, in order to clarify its functionalities:

3.5.2 Scenario 1: Tram trip in Milan

3.5.2.1 Overview

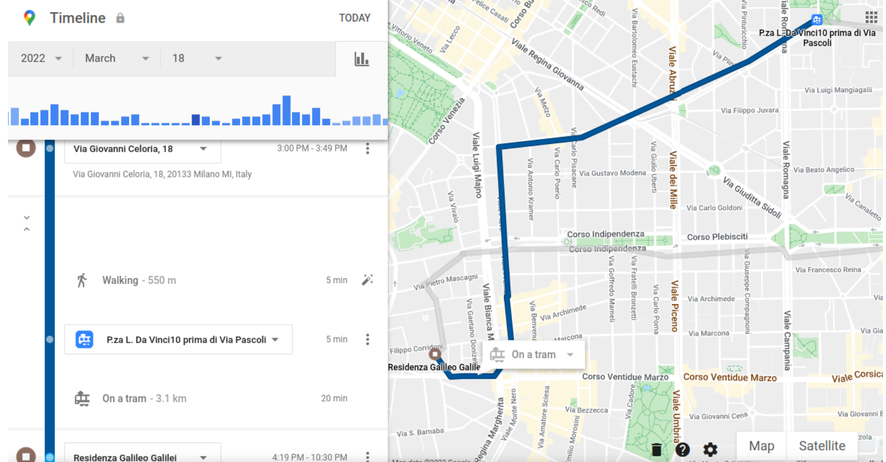


Figure 27: The picture map represents the first scenario: a tram trip in Milan.

The first scenario we are going to analyze is a tram trip from a tram stop near Piazza Leonardo (an square in Milan, Italy) to Residenza Galileo Galilei (an student residence also in Milan, Italy). Specifically, the piece of the trip in tram went from 15:59 to 16:19 the 18th of March (2022).

For this reason, we are going to work with a trip built from the table database location_history0322 specifying the proper timestamptz, as we can see below (Figure 28 and Figure 29).

Query Editor Query History Scratch Pad

```
1 CREATE TABLE tramTrip (
2 date_ date,
3 trip tgeompoint,
4 trajectory_ geometry
5 );
```

Figure 28: Query to create the table tramTrip which will contain the trip info about Scenario 1.

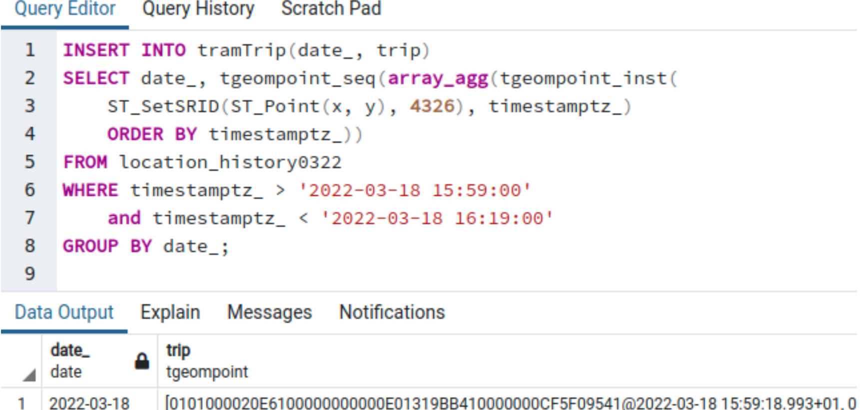


Figure 29: Query that inserts the tram trip related info into the table created in Figure 28.

As we can see in the picture above we built a trip (tgeompoint datatype) with the (x,y) coordinates and the corresponding timestamps as similar as seen in Section 3.4, but using the unnested data obtained at first part of that Section (location_history0322 database).

3.5.2.2 Query execution

Now, we are going to check some details about this Scenario, using the functions mentioned in Section 2.2. We are using the database created and filled in Figure 28 and Figure 29 for this purpose.

First, we will check its start and end timestamps:

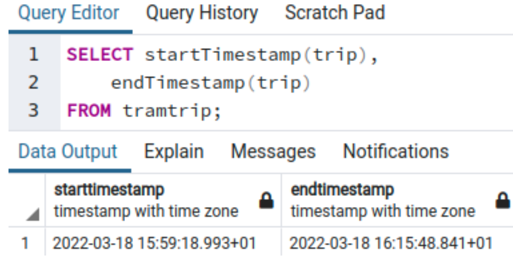


Figure 30: The query that calculates the start and end timestamps of the trip of the Scenario 1, with both results below the query.

As we can see in Figure 30, the start timestamp returned us the first timestamp (with time zone) value in the trip sequence ('2022-03-18 15:59:18.993+01') that

matches the value seen in Figure 27. However, the end timestamp returned did not match with the value seen in Figure 27, that is because the last timestamp value that is less than what we set as maximum (2022-03-18 16:19:00) is '2022-03-18 16:15:48.841+01'.

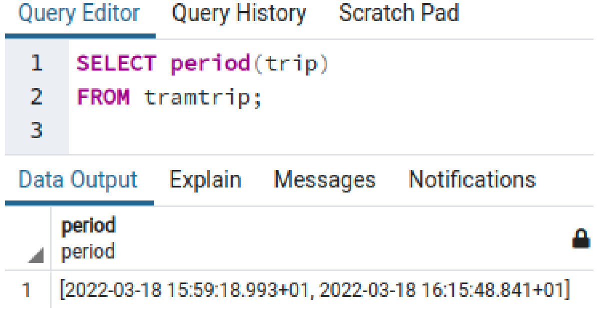


Figure 31: The query that calculates the period timestamps of the trip of Scenario 1, with the result below the query.

In the Figure 31 we have the a query that executes the period function mentioned in Section 2.2 with its result below. The result expresses and confirm what we obtained above with the startTimestamp and endTimestamp functions: the first and the last timestamps in the trip tgeompoint sequence.

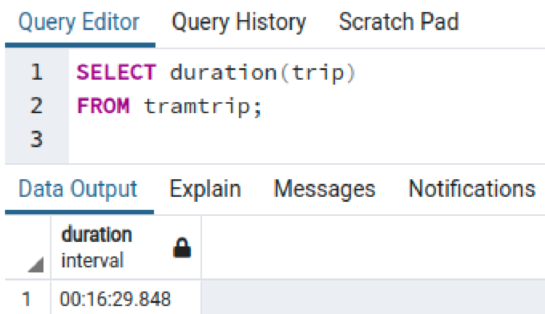


Figure 32: The query that calculates the duration of the trip of the Scenario 1, with the result below the query.

In the Figure 32 we have the a query that executes the duration function mentioned in Section 2.2 with its result below. The result expresses 16 min and 29.848 seconds that reflect the time distance between first and last timestamps, 4 min less than what we expected in Figure 27 (20 min).

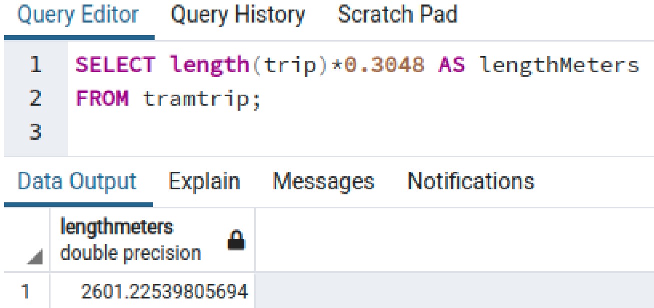


Figure 33: The query that calculates the length of the trip of the Scenario 1, with the result below the query.

In the Figure 33 we have the a query that executes the length function mentioned in Section 2.2 with its result below. It is interesting to note that, by default, the result given by length was in feets, so we had to cast it to meters multiplying it by 0.3048 and the result was 2600 meters (2.6km) while the distance expected in Figure 27 was about 3.1km. We lost 500m. So is easy to conclude at this point that we have lost 4 minutes of travel that is explicitly reflected in the functions results, which have probably made the 500m lost when calculating the length of the trip, because the last timestamp taken by the trip tgeompoint sequence is at 16:15 aprox, and the real trip ended at 16:20 aprox, so taking into account that we were in a tram, not walking around, that would be a very possible reason for the mismatching.

Finally, let's test one of the most interesting functions: speed function:

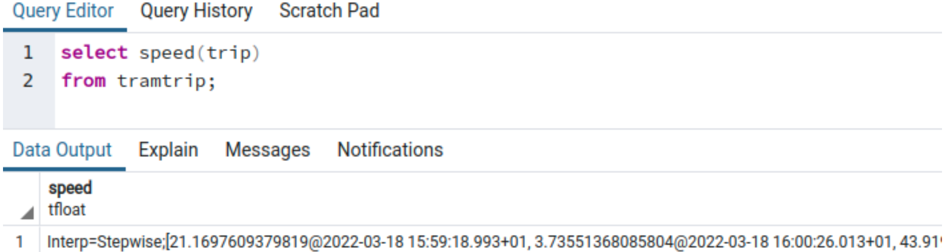


Figure 34: The query that calculates the speed value.

Above we have a simple execution of the speed function. But, contrary to what would be expected, the result is not an integer/double/float value, it is a tfloat. What returns the speed function is a sequence of each speed reached in each timestamp, which is referred by the 'Stepwise' at the beginning of the sequence. This sequence is what refers to the interpolation about what we were talking

before, in section 3.5.1. That is why speed function must have linear interpolation or it would cause and error at its execution: because the result of the speed function, with no cast added, is an interpolation of different speed values at the different timestamps.

But we can cast it to get a real and more intuitive number (integer, double, float or whatever) with the function we mentioned together with speed function: TwAvg. This function (Time-weighted Average) computes the average of a time-weighted sequence, which really fits what we need right now for the speed function.

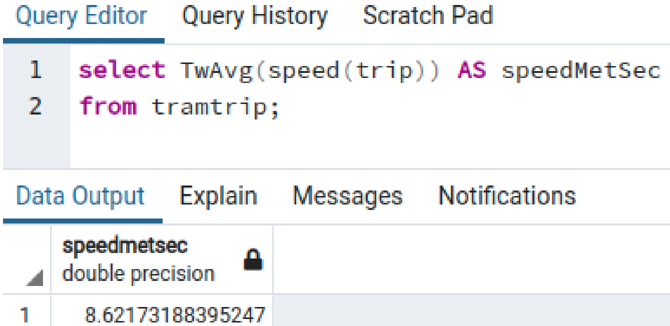


Figure 35: The query that calculates the average speed value, in meters per second.

As showed in the Figure 35 above, the TwAvg function lets us to cast that speed interpolation into a double real value (8.622 meters per second), whose value casted into KmH is 31.04 KmH (as seen below in Figure 36).

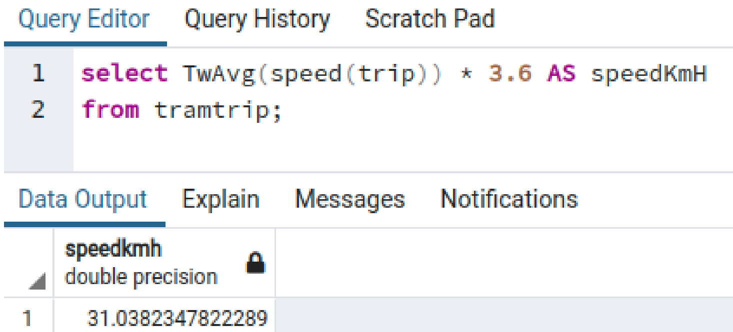


Figure 36: The query that calculates the average speed value, and casts it to KmH.

3.5.3 Scenario 2: Plane trip Milan-Amsterdam

3.5.3.1 Overview

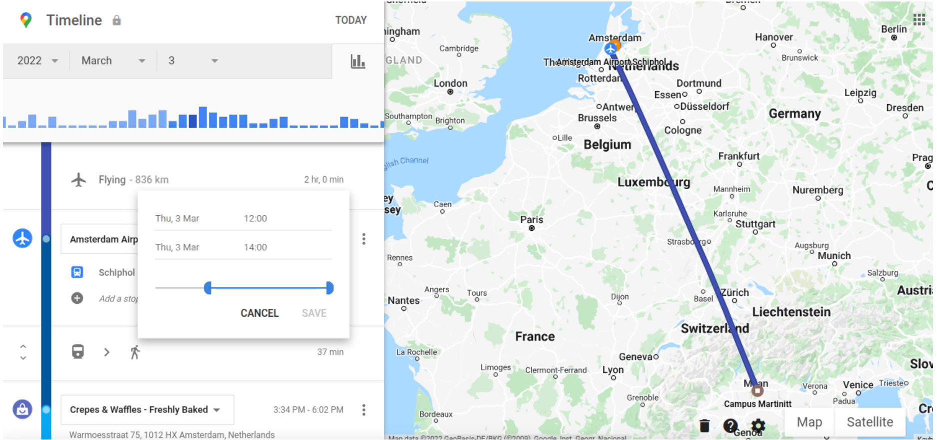


Figure 37: The picture shows a plane travel someone did from Milan, Italy to Amsterdam, Netherlands on 03-03-2022.

The second scenario we are going to analyze is a plane trip from Milan, Italy to Amsterdam, Netherlands. Specifically, the trip went from 12h to 14h, the 3rd of March (2022).

For this reason, we are going to work with a trip built from the table database location_history0322 specifying the proper timestamptz, as we can see below (Figure 38 and Figure 38).

Query Editor Query History Scratch Pad

```
1 CREATE TABLE planeTrip (
2 date_ date,
3 trip tgeompoint,
4 trajectory_ geometry
5 );
```

Figure 38: Query to create the table planeTrip which will contain the trip info about Scenario 2.

Figure 39: Query that inserts the plane trip related info into the table created in Figure 38.

As we can see in the picture above we built a trip (tgeompoint datatype) with the (x,y) coordinates and the corresponding timestamps as similar as seen in Section 3.4, but using the unnested data obtained at first part of that Section (location_history0322 database).

3.5.3.2 Query execution

Now, we are going to check some details about this Scenario, using the functions mentioned in Section 2.2. We are using the database created and filled in Figure 38 and Figure 39 for this purpose.

First, we will check its start and end timestamps:

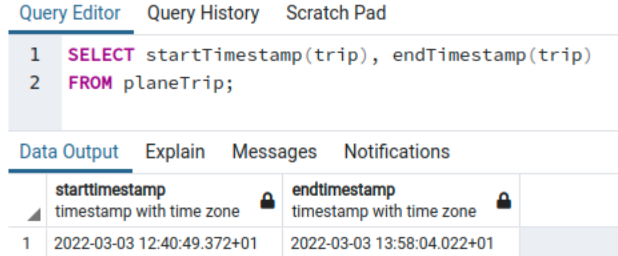


Figure 40: The query that calculates the start and end timestamps of the trip of the Scenario 2, with both results below the query.

As we can see in Figure 40, the start timestamp returned us the first timestamp (with time zone) value in the trip sequence ('2022-03-03 12:40:49.372+01') that

does not match the value seen in Figure 37, the most probably because as the phone that provided the geolocation was with its owner in the airport, it had coverage problems so was not able to provide a proper location all the time. However, the end timestamp returned match with the value seen in Figure 37 ('2022-03-03 13:58:04.022+01').

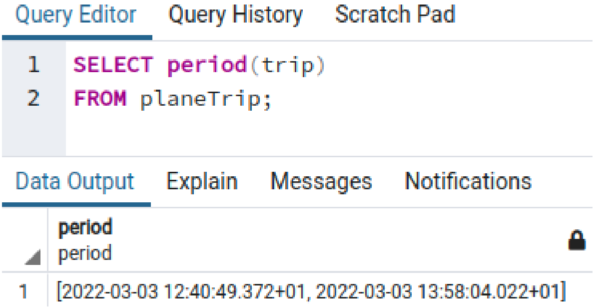


Figure 41: The query that calculates the period timestamps of the trip of Scenario 2, with the result below the query.

In the Figure 41 we have the a query that executes the period function mentioned in Section 2.2 with its result below. The result expresses and confirm what we obtained above with the startTimestamp and endTimestamp functions: the first and the last timestamps in the trip tgeompoint sequence.

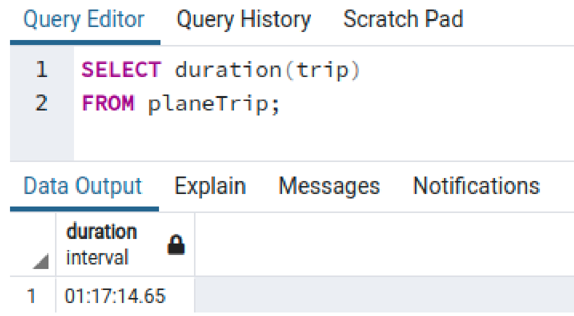


Figure 42: The query that calculates the duration of the trip of the Scenario 2, with the result below the query.

In the Figure 42 we have the a query that executes the duration function mentioned in Section 2.2 with its result below. The result expresses 1h, 17min and 14.65 seconds that reflects the time distance between first and last timestamps, but because of what we mentioned before (coverage issues in the departure airport) we have received less duration than what we expected in Figure 37 (20

min).

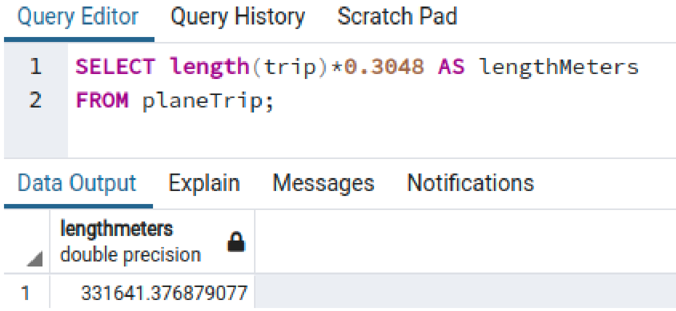


Figure 43: The query that calculates the length of the trip of the Scenario 2, with the result below the query.

In the Figure 43 we have the a query that executes the length function mentioned in Section 2.2 with its result below. There occurs the same as in Scenario 1: the result was given in feets and we casted it to meters multiplying it by 0.3048. The result is 331641.38 meters (331,6Km), while the distance expected in Figure 37 was about 800km. We lost almost 500km because of the coverage issues we mentioned before, same issue as we saw with duration result.

Finally, it is the turn of the speed function:

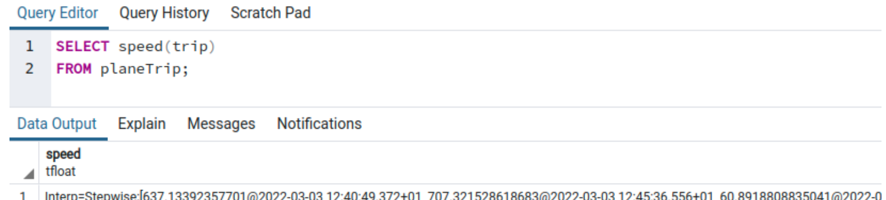


Figure 44: The query that calculates the speed value.

We follow the same procedure as the first Scenario: first we show the interpolation speed sequence (Figure 44) and then, below, we have the same speed, but after applying the TwAvg function in order to get a real unique number. Below we have the result in meters per second (Figure 45) and then in KmH (Figure 46).
But we can cast it to get a real and more intuitive number (integer, double, float or whatever) with the function we mentioned together with speed function: TwAvg. This function (Time-weighted Average) computes the average of a time-weighted sequence, which really fits what we need right now for the speed

function.

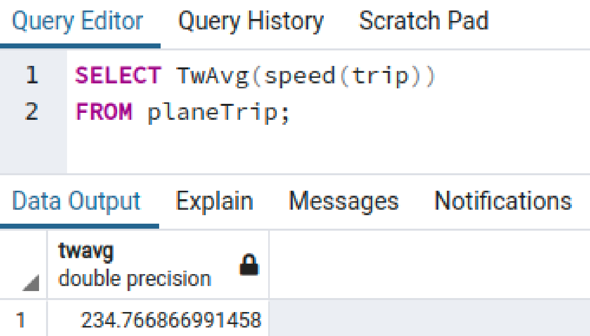


Figure 45: The query that calculates the average speed value, in meters per second.

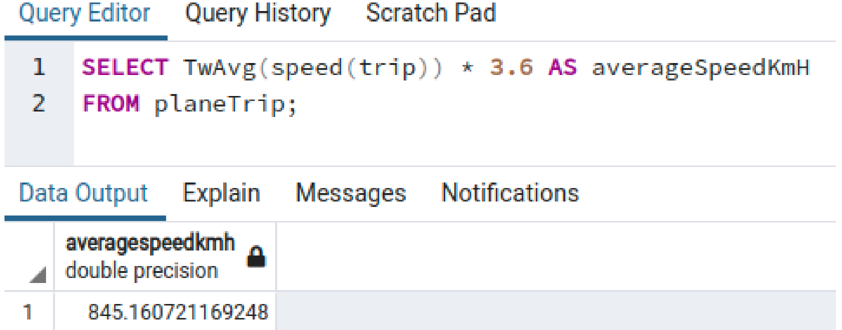


Figure 46: The query that calculates the average speed value, and casts it to KmH.

It is really interesting to note that despite of having seen the error with the coverage issue that made losing data and not having too much accurate results, the average speed values are still curiously proper, so an speed of 845KmH is a real average speed for a plane. Despite of the errors with distances, the speed is maintaining its correctness.

3.5.4 Scenario 3: A visit to a museum

3.5.4.1 Overview

For the last example, we wanted to try something different, something related to indoor mobility data. So, we are going to consider one data set about a visit

to a museum. As the two first scenarios were about continuous trajectories, we considered would be interesting to analyze one discontinuous. The scenario to analyze is based in the data set imported in Section 3.3. The main idea is to evaluate how much can change the results between using (x,y) coordinates or (x,y) with Kalman filtered, which is supposed to be more accurate.

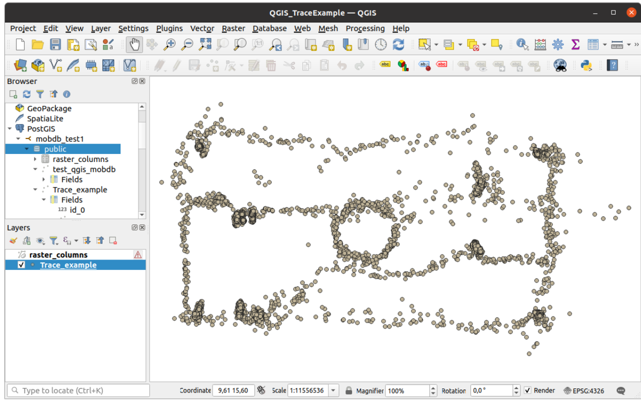


Figure 47: Picture map with the relating the position coordinates of the museum's visitor according to its timestamp.

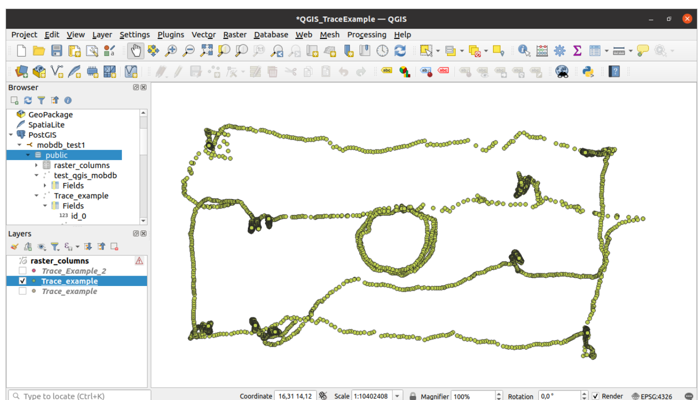


Figure 48: Picture map with the relating the position coordinates (with filtered values) of the museum's visitor according to its timestamp.

Now, we take the SQL query seen in Figure 6, to get the table database result shown in Figure 7, where the first row refers to the trip of Figure 47, and the second one refers to the trip of filtered values, from Figure 48.

3.5.4.2 Query execution

The first thing we can compute in order to compare them is the start and the end timestamps (Figure ??).

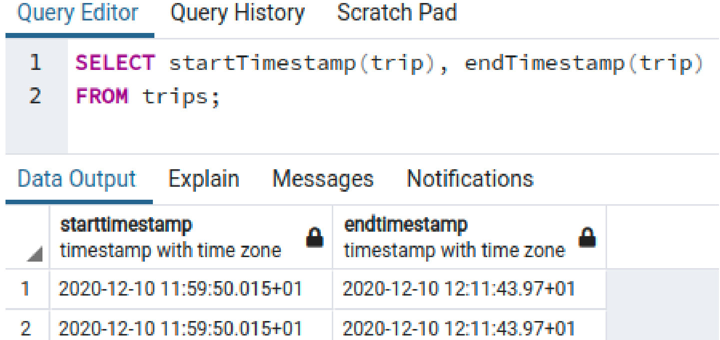


Figure 49: The query that gets the start and the end timestamps from both trips.

Obviously, like the Kalman filter only is changing the x and y values, the timestamp values remain the same. So both values are the same.

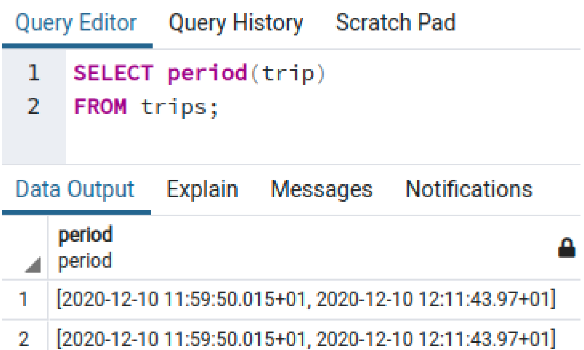


Figure 50: The query that gets the period from both trips.

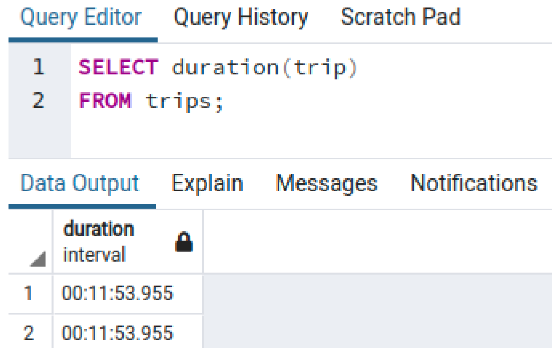


Figure 51: The query that gets the duration from both trips.

In the same line as before, as both values (period and duration) depend of the timestamps, both values are the same, does not matter if x and y are filtered. Now, the first change should come: length.

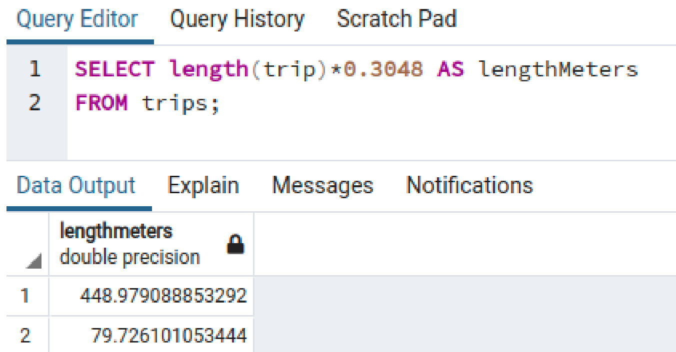
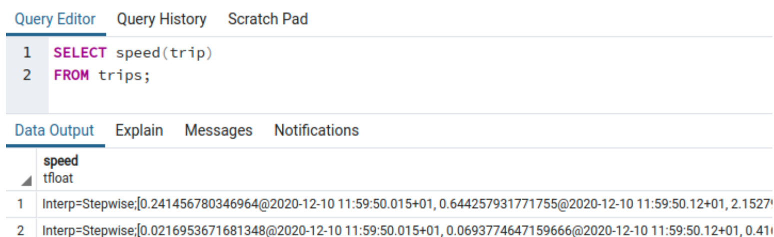


Figure 52: The query to get the length from both trips.

As we supposed before, here we can already see some changes. That is because the Kalman filter uses a series of measurements observed over time (like inaccuracies) and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone, by estimating a joint probability distribution over the variables for each timeframe. Therefore, it is easily observable a big difference in their lengths (Figure 52): trip with non-filtered coordinates (x,y) has almost 449 meters of length, almost 6 times the filtered one, while the trip that applied Kalman filter has a length of 79.73 meters (both results are in meters because in the query there is the cast from feets to meters multiplying by 0.3048).

Now, let's check the speed. Theoretically, if there are huge differences between

the lengths, there should be an obvious difference between the speeds.


Query Editor
Query History
Scratch Pad

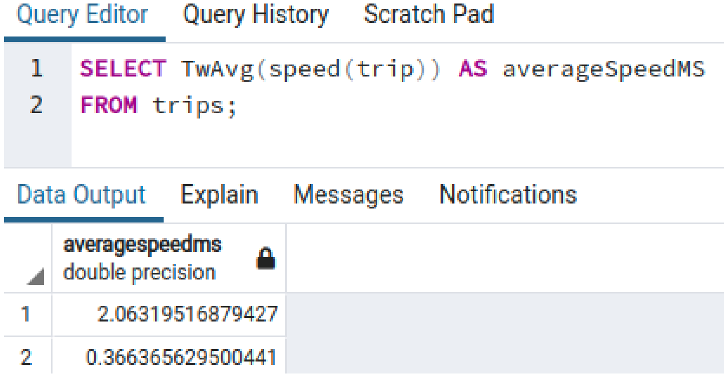
```
1 SELECT speed(trip)
2 FROM trips;
```

Data Output
Explain
Messages
Notifications

	speed tfloat
1	Interp=Stepwise;[0.241456780346964@2020-12-10 11:59:50.015+01, 0.644257931771755@2020-12-10 11:59:50.12+01, 2.1527
2	Interp=Stepwise;[0.0216953671681348@2020-12-10 11:59:50.015+01, 0.0693774647159666@2020-12-10 11:59:50.12+01, 0.41

Figure 53: The query that calculates the speed value.

At first sight, is not really evident, but if we look deeply inside the interpolation sequences (Figure 53, we can see that the speed coefficient of the non-filtered coordinates trip is much bigger. But, let's see first the average values to establish a better comparison.


Query Editor
Query History
Scratch Pad

```
1 SELECT TwAvg(speed(trip)) AS averageSpeedMS
2 FROM trips;
```

Data Output
Explain
Messages
Notifications

	averagespeedms double precision
1	2.06319516879427
2	0.366365629500441

Figure 54: The query that calculates the average speed value, in meters per second.

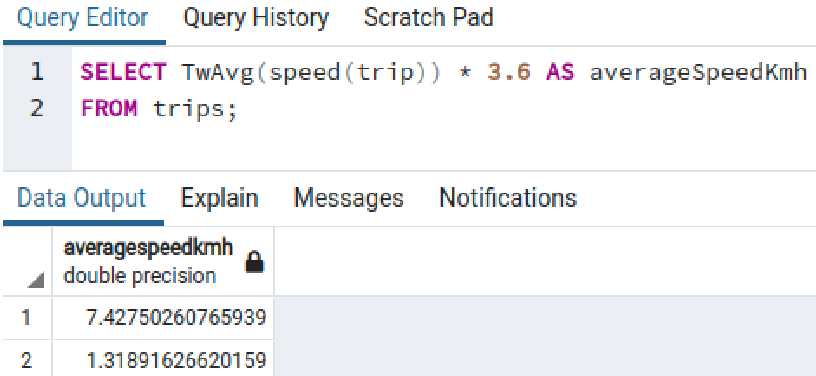


Figure 55: The query that calculates the average speed value, and casts it to KmH.

Now is pretty much more obvious. As we can see in Figure 55, the Kalman-filtered trip has an speed value of 1.32kmh, while the non-filtered trip has an amazing 7.43kmh value. The reason why this occurs is evident. As we saw at the beginning of this Scenario, the timestamps were the same, the duration of the trips (Figure 51) were exactly the same, as the Kalman filter only modify some coordinates aspects, nothing about time. Then, we saw when computing the lengths (Figure 52) that there was a big difference between them. So, it is obvious that if both trips have the same duration, the speed of the trip with the biggest length will be the biggest too, while the smallest length will have the smallest speed, because distance is directly proportional to the speed: the higher the distance, the higher the speed (having same duration).

3.6 Conclusions

This thesis is about the management of mobility data through a recent and promising system called MobilityDB.

Undoubtly, there is a strict relationship between mobility data management and IOT. Therefore the techniques used for the management of mobility data can be potentially applied for handling data from sensors (the GPS receiver is one example of sensor).

In this thesis, I have learned to recognize and analyze space-time trajectory data, to use modern and interesting tools such as Google Timeline and Google Takeout to obtain mobility data, as well as to learn to use two very interesting programs (QGIS and PgAdmin, with its corresponding MobilityDB extension) that facilitate all this data management and analysis.

Concerning MobilityDB (so it is the main focus of this thesis), it is a tool that I did not even know about, and I have found it to be a very useful, modern

tool that offers an infinity of possibilities that, although I have not been able to get the most out of it, I consider myself minimally knowledgeable about said system and its operation.

3.6.1 Complex issues

The first complex issue I faced was the installation of all the necessary software for the experimental part of the project. There was a guide on the MobilityDB developers home page to install MobilityDB, along with all its dependencies. However, to begin with, as of today it is still not possible to use this tool in a Windows environment (although both QGIS and PgAdmin can). Although this was not really such a big problem, since I had experience installing software in Linux/Ubuntu environments. However, as you followed the guide laid out on the developer's website, more and more problems with your installation appeared, which can become frustrating. I started trying to install the entire environment in an Ubuntu 20.04 virtual machine on Windows 10, but all were problems due to the processing requirements of the environment and the low capacity that a virtual machine can provide compared to a real environment. Therefore, I ended up installing the entire environment on a disk partition with Ubuntu 20.04 on my laptop and from there everything was easier.

Another complex issue was the conceptualization of the velocity function. The speed function in the MobilityDB system is quite important and its meaning can be a bit complex, as I explain in some section, it is expressed as the interpolation of an interval, and the concept of interpolation was not familiar to me. I needed to watch the occasional video and read a workshop with examples to be able to understand what exactly said function expressed and what its limits were.

It is worth mentioning a small complexity that I found while establishing the database to use for the creation of the scenarios at the end of the experimental part of my thesis, it was the management and adaptation of a large volume of mobility data (since such and As I explain in the section regarding Google Timeline, I obtain a quantity of data referring to movement trajectories of between 4 and 5 years). It was not a complication at the workload level, but in terms of conceptualization, since certain concepts of SQL querying had to be very clear.

References

[1] Regina O. Obe and Leo S. Hsu. *PostGIS in Action*. 2nd. USA: Manning Publications Co., 2015. ISBN: 1617291390.

[2] Esteban Zimányi et al. "MobilityDB: A Mainstream Moving Object Database System". In: *Proceedings of the 16th International Symposium on Spatial and Temporal Databases*. SSTD '19. Vienna, Austria: Association for Computing Machinery, 2019, pp. 206–209. ISBN: 9781450362801. DOI: 10.1145/3340964.3340991. URL: https://doi.org/10.1145/3340964.3340991.

[3] Esteban Zimányi, Mahmoud Sakr, and Arthur Lesuisse. "MobilityDB: A Mobility Database Based on PostgreSQL and PostGIS". In: *ACM Transactions on Database Systems* 45 (Dec. 2020), pp. 1–42. DOI: 10.1145/3406534.

[4] Esteban Zimányi et al. "MobilityDB: Hands on Tutorial on Managing and Visualizing Geospatial Trajectories in SQL". In: *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on APIs and Libraries for Geospatial Data Science*. New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 9781450391030. URL: https://doi.org/10.1145/3486189.3490016.

[5] *Definition of Point*. URL: https://www.mathsisfun.com/definitions/point.html.

[6] *How to Visualize Your Google Location History - Make Tech Easier*. URL: https://www.maketecheasier.com/visualize-google-location-history/#export-location-history.

[7] *Making MathWorld « The Mathematica Journal*. en-US. URL: https://www.mathematica-journal.com/2007/08/06/making-mathworld/ (visited on 06/05/2022).

[8] *MobilityDB installation requirements*. URL: https://mobilitydb.com/install.html.

[9] *MobilityDB webpage*. URL: https://mobilitydb.com/.