MobilityDB Workshop

COLLABORATORS			
	TITLE : MobilityDB Workshop		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Mahmoud SAKR and Esteban ZIMÁNYI	June 29, 2020	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

List of Figures

List of Tables

Abstract

Every module in this workshop illustrates a usage scenario of MobilityDB. The data sets and the tools are described inside each of the modules. Eventually, additional modules will be added to discover more MobilityDB features.

While this workshop illustrates the usage of MobilityDB functions, it does not explain them in detail. If you need help concerning the functions of MobilityDB, please refer to the documentation.

If you have questions, ideas, comments, etc., please contact me on mahmoud.sakr@ulb.ac.be.

Chapter 1

Managing Ship Trajectories (AIS)

AIS stands for Automatic Identification System. It is the location tracking system for sea vessels. This module illustrates how to load big AIS data sets into MobilityDB and do basic exploration.

The idea of this module is inspired from the tutorial of MovingPandas on ship data analysis by Anita Graser.

1.1 Contents

This module covers the following topics:

- Loading large trajectory datasets into MobilityDB
- Create proper indexes to speed up trajectory construction
- Select trajectories by a spatial window
- Join trajectories tables by proximity
- Select certain parts inside individual trajectories
- Manage the temporal speed and azimuth features of ships

1.2 Data

The Danish Maritime Authority publishes about 3 TB of AIS routes in CSV format here. The columns in the CSV are listed in Table 1.1. This module uses the data of one day April 1st 2018. The CSV file size is 1.9 GB, and it contains about 10 M rows.

1.3 Tools

The tools used in this module are as follows:

- MobilityDB, on top of PostgreSQL and PostGIS. Here I use the MobilityDB docker image.
- QGIS

Timestamp	Timestamp from the AIS base station, format: 31/12/2015 23:59:59
Type of mobile	Describes what type of target this message is received from (class A AIS Vessel, Class B AIS vessel, etc)
MMSI	MMSI number of vessel
Latitude	Latitude of message report (e.g. 57,8794)
Longitude	Longitude of message report (e.g. 17,9125)
Navigational status	Navigational status from AIS message if available, e.g.: 'Engaged in fishing', 'Under way using engine', mv.
ROT	Rot of turn from AIS message if available
SOG	Speed over ground from AIS message if available
COG	Course over ground from AIS message if available
Heading	Heading from AIS message if available
IMO	IMO number of the vessel
Callsign	Callsign of the vessel
Name	Name of the vessel
Ship type	Describes the AIS ship type of this vessel
Cargo type	Type of cargo from the AIS message
Width	Width of the vessel
Length	Lenght of the vessel
Type of position fixing device	Type of positional fixing device from the AIS message
Draught	Draugth field from AIS message
Destination	Destination from AIS message
ETA	Estimated Time of Arrival, if available
Data source type	Data source type, e.g. AIS
Size A	Length from GPS to the bow
Size B	Length from GPS to the stern
Size C	Length from GPS to starboard side
Size D	Length from GPS to port side

Table 1.1: AIS columns

1.4 Preparing the Database

Create a new database `DanishAIS`, then use your SQL editor to create the MobilityDB extension as follows:

```
CREATE EXTENSION MobilityDB CASCADE;
```

The `CASCADE` command will additionally create the PostGIS extension.

Now create a table in which the CSV file will be loaded:

```
CREATE TABLE AISInput(
  T timestamp,
  TypeOfMobile varchar(50),
  MMSI integer,
  Latitude float,
  Longitude float,
  navigationalStatus varchar(50),
  ROT float,
  SOG float,
  COG float,
  Heading integer,
  IMO varchar(50),
  Callsign varchar(50),
  Name varchar(100),
  ShipType varchar(50),
  CargoType varchar(100),
  Width float,
  Length float,
  TypeOfPositionFixingDevice varchar(50),
  Draught float,
  Destination varchar(50),
  ETA varchar(50),
  DataSourceType varchar(50),
  SizeA float,
  SizeB float,
  SizeC float,
  SizeD float,
  Geom geometry(Point, 4326)
);
```

1.5 Loading the Data

For importing CSV data into a PostgreSQL database one can use the `COPY` command as follows:

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

This import took about 3 minutes on my machine, which is an average laptop. The CSV file has 10,619,212 rows, all of which were correctly imported. For bigger datasets, one could alternative could use the program pgloader.

We clean up some of the fields in the table and create spatial points with the following command.

```
UPDATE AISInput SET
NavigationalStatus = CASE NavigationalStatus WHEN 'Unknown value' THEN NULL END,
IMO = CASE IMO WHEN 'Unknown' THEN NULL END,
ShipType = CASE ShipType WHEN 'Undefined' THEN NULL END,
```

```
TypeOfPositionFixingDevice = CASE TypeOfPositionFixingDevice
  WHEN 'Undefined' THEN NULL END,
Geom = ST_SetSRID( ST_MakePoint( Longitude, Latitude ), 4326);
```

This took about 5 minutes on my machine. Let's visualize the spatial points on QGIS.

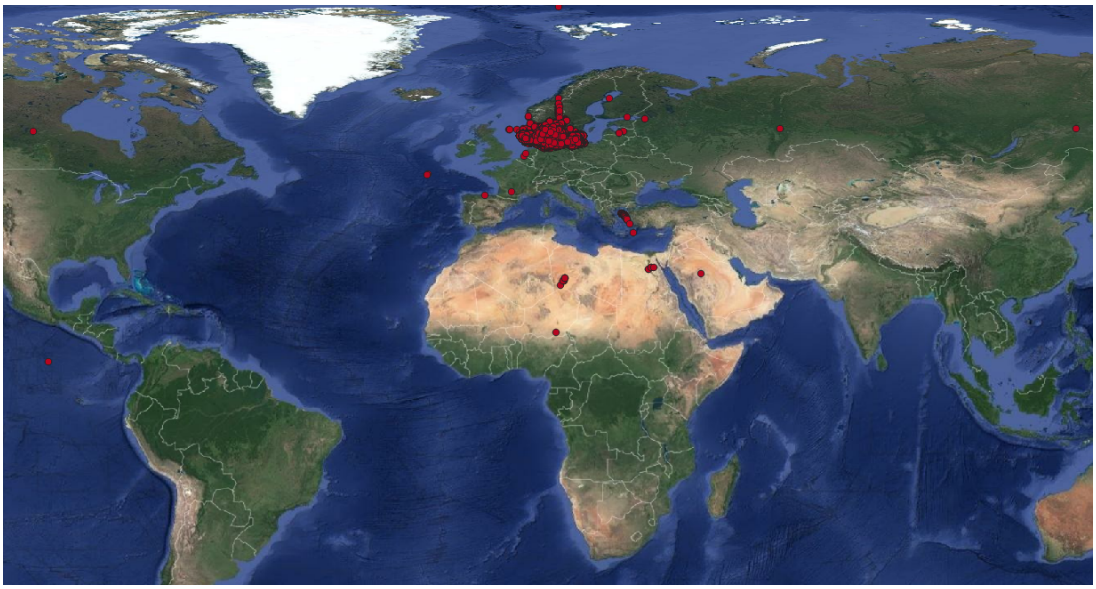

Figure 1.1: Visualizing the input points

Clearly, there are noise points that are far away from Denmark or even outside earth. This module will not discuss a thorough data cleaning. However, we do some basic cleaning in order to be able to construct trajectories:

- Filter out points that are outside the window defined by bounds point(-16.1,40.18) and point(32.88, 84.17). This window is obtained from the specifications of the projection in https://epsg.io/25832.
- Filter out the rows that have the same identifier (MMSI, T)

```
CREATE TABLE AISInputFiltered AS
SELECT DISTINCT ON(MMSI,T) *
FROM AISInput
WHERE Longitude BETWEEN -16.1 and 32.88 AND Latitude BETWEEN 40.18 AND 84.17;
-- Query returned successfully: 10357703 rows affected, 01:14 minutes execution time.
SELECT COUNT(*) FROM AISInputFiltered;
--10357703
```

1.6 Constructing Trajectories

Now we are ready to construct ship trajectories out of their individual observations:

```
CREATE TABLE Ships(MMSI, Trip, SOG, COG) AS
SELECT MMSI,
  tgeompointseq(array_agg(tgeompointinst( ST_Transform(Geom, 25832), T) ORDER BY T)),
  tfloatseq(array_agg(tfloatinst(SOG, T) ORDER BY T) FILTER (WHERE SOG IS NOT NULL)),
  tfloatseq(array_agg(tfloatinst(COG, T) ORDER BY T) FILTER (WHERE COG IS NOT NULL))
FROM AISInputFiltered
GROUP BY MMSI;
-- Query returned successfully: 2995 rows affected, 01:16 minutes execution time.
```

This query constructs, per ship, its spatiotemporal trajectory `Trip`, and two temporal attributes `SOG` and `COG`. `Trip` is a temporal geometry point, and both `SOG` and `COG` are temporal floats. MobilityDB builds on the coordinate transformation feature of PostGIS. Here the SRID 25832 (European Terrestrial Reference System 1989) is used, because it is the one advised by Danish Maritime Authority in the download page of this dataset. Now, let's visualize the constructed trajectories in QGIS.

```
ALTER TABLE Ships ADD COLUMN Traj geometry;
UPDATE Ships SET Traj= trajectory(Trip);
-- Query returned successfully: 2995 rows affected, 3.8 secs execution time.
```

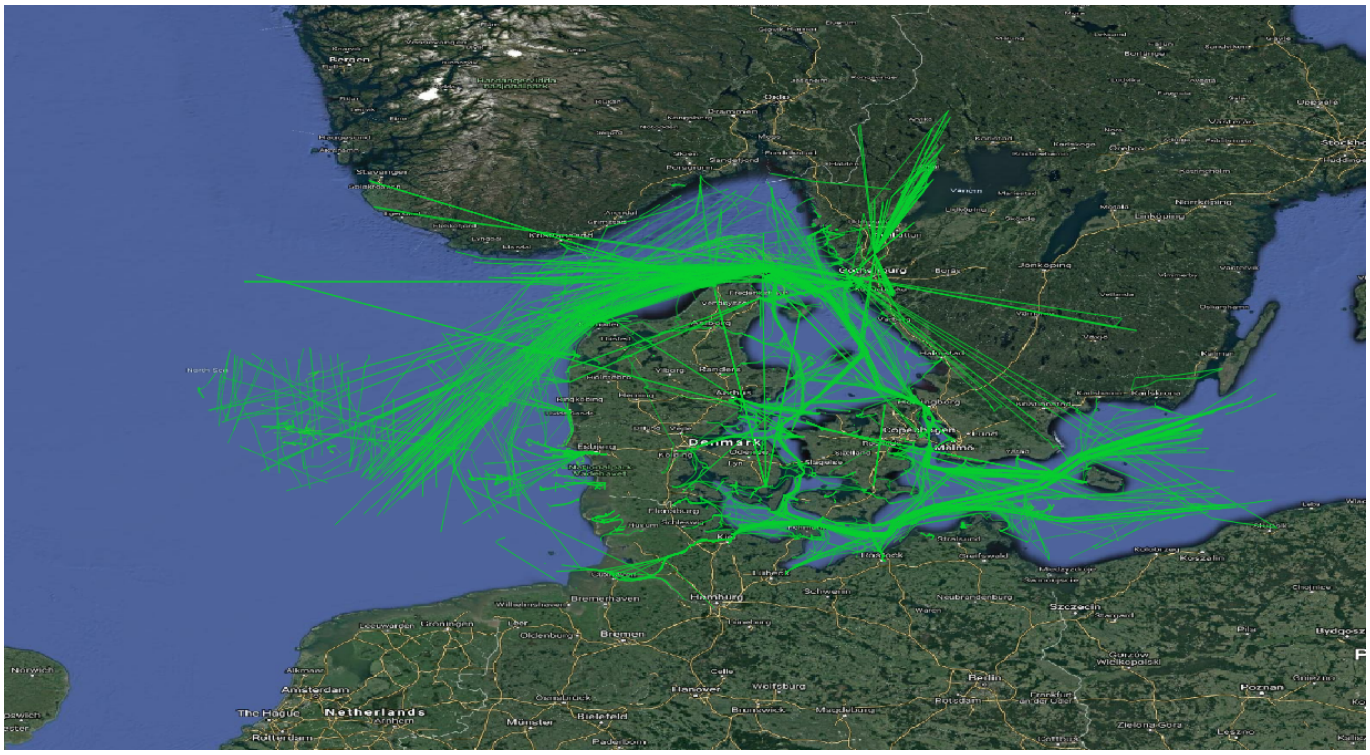

Figure 1.2: Visualizing the ship trajectories

1.7 Basic Data Exploration

The total distance traveled by all ships:

```
SELECT SUM( length( Trip ) ) FROM Ships;
--500433519.121321
```

This query uses the `length` function to compute per trip the sailing distance in meters. We then aggregate over all trips and calculate the sum. Let's have a more detailed look, and generate a histogram of trip lengths:

```
WITH buckets (bucketNo, RangeKM) AS (
  SELECT 1, floatrange '[0, 0]' UNION
  SELECT 2, floatrange '(0, 50)' UNION
  SELECT 3, floatrange '[50, 100)' UNION
  SELECT 4, floatrange '[100, 200)' UNION
  SELECT 5, floatrange '[200, 500)' UNION
  SELECT 6, floatrange '[500, 1500)' UNION
  SELECT 7, floatrange '[1500, 10000)' ),
histogram AS (
  SELECT bucketNo, RangeKM, count(MMSI) as freq
```

```
  FROM buckets left outer join Ships on (length(Trip)/1000) <@ RangeKM
  GROUP BY bucketNo, RangeKM
  ORDER BY bucketNo, RangeKM
)
SELECT bucketNo, RangeKM, freq,
  repeat('■', ( freq::float / max(freq) OVER () * 30 )::int ) AS bar
FROM histogram;
--Total query runtime: 5.6 secs

bucketNo,   bucketRange,        freq     bar
1;          "[0,0]";            303;       ■■■■■
2;          "(0,50)";           1693;      ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
3;          "[50,100)";         267;       ■■■■■
4;          "[100,200)";        276;       ■■■■■
5;          "[200,500)";        361;       ■■■■■■
6;          "[500,1500)";       86;        ■■
7;          "[1500,10000)";     6;
```

Surprisingly there are trips with zero length. These are clearly noise that can be deleted. Also there are very many short trips, that are less than 50 km long. On the other hand, there are few long trips that are more than 1,500 km long. Let's visualize these last two cases in Figure 1.3. They look like noise. Normally one should validate more, but to simplify this module, we consider them as noise, and delete them.

```
DELETE FROM Ships
WHERE length(Trip) = 0 OR length(Trip) >= 1500000;
-- Query returned successfully in 7 secs 304 msec.
```

Now the `Ships` table looks like Figure 1.4.

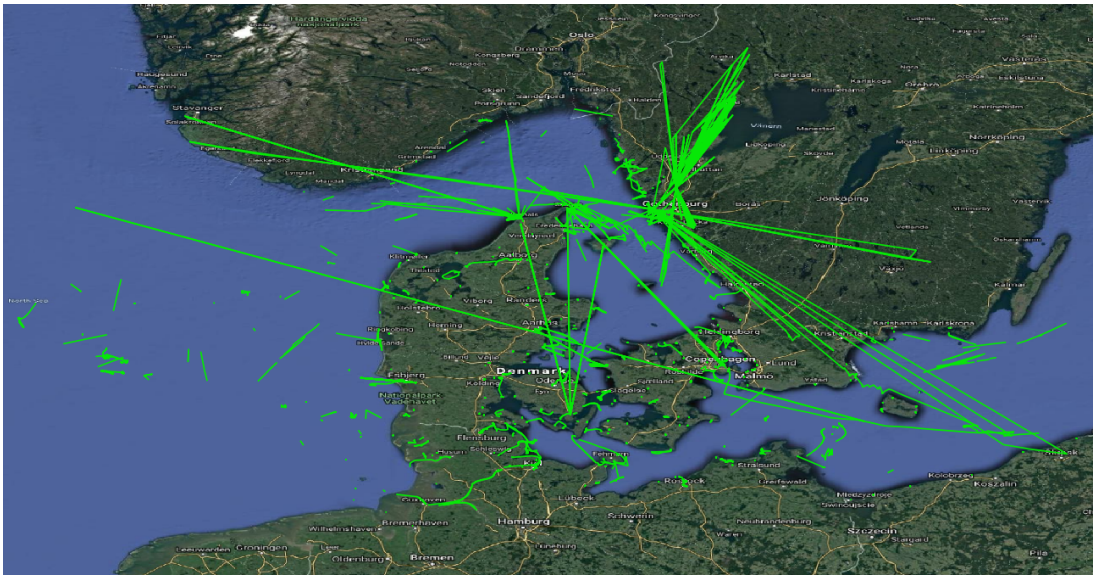

Figure 1.3: Visualizing trips with abnormal lengths

Let's have a look at the speed of the ships. There are two speed values in the data; the speed calculated from the spatiotemporal trajectory `speed(Trip)`, and the `SOG` attribute. Optimally, the two will be the same. A small variance would still be OK, because of sensor errors. Note that both are temporal floats. In the next query, we compare the averages of the two speed values for every ship:

```
SELECT ABS(twavg(SOG) * 1.852 - twavg(speed(Trip))* 3.6 ) SpeedDifference
FROM Ships
ORDER BY SpeedDifference DESC;
--Total query runtime: 8.2 secs
```

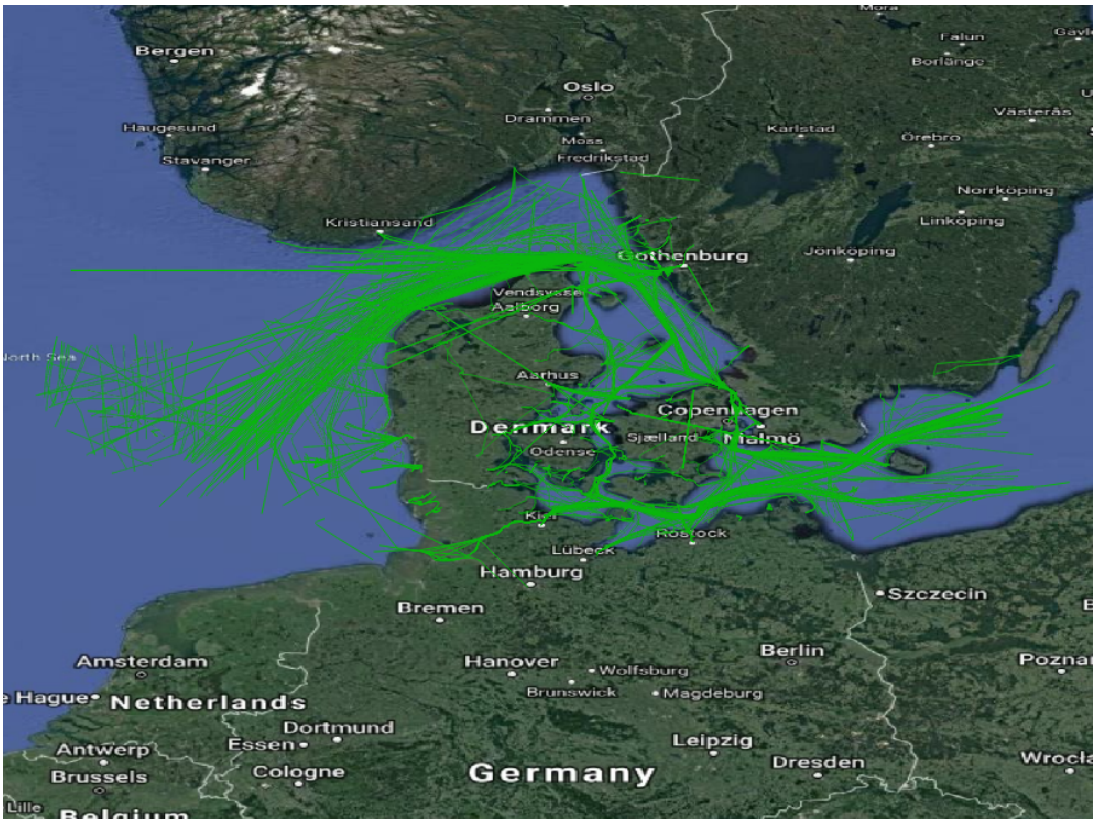

Figure 1.4: Ship trajectories after filtering

```
--990 rows retrieved.

SpeedDifference
NULL
NULL
NULL
NULL
NULL
107.861100067879
57.1590253627668
42.4207839833568
39.5819188407125
33.6182789410313
30.9078594633161
26.514042447366
22.1312646226031
20.5389022294181
19.8500569368283
19.4134688682774
18.180139457754
17.4859077178001
17.3155991287105
17.1739822139821
12.9571603234404
12.6195380496344
12.2714437568609
10.9619033557275
10.4164745930929
10.3306155308426
```

```
9.46457823214455
...
```

The `twavg` computes a time-weighted average of a temporal float. It basically computes the area under the curve, then divides it by the time duration of the temporal float. By doing so, the speed values that remain for longer durations affect the average more than those that remain for shorter durations. Note that SOG is in knot, and Speed(Trip) is in m/s. The query converts both to km/h.

The query shows that 26 out of the 990 ship trajectories in the table have a difference of more than 10 km/h or NULL. These trajectories are shown in Figure 1.5. Again they look like noise, so we remove them.

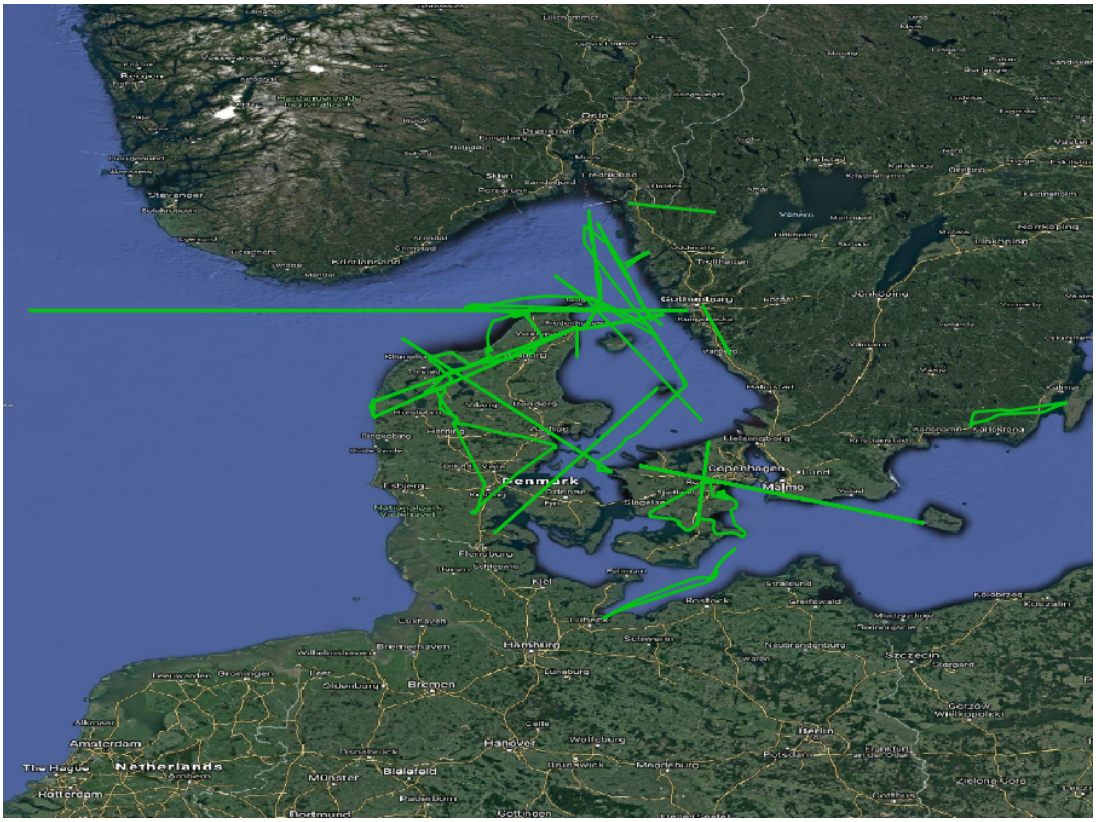

Figure 1.5: Ship trajectories with big difference between `speed(Trip)` and `SOG`

Now we do a similar comparison between the calculated azimuth from the spatiotemporal trajectory, and the attribute COG:

```
SELECT ABS(twavg(COG) - twavg(azimuth(Trip)) * 180.0/pi() ) AzimuthDifference
FROM Ships
ORDER BY AzimuthDifference DESC;
--Total query runtime: 4.0 secs
--964 rows retrieved.

264.838740787458
220.958372832234
180.867071483688
178.774337481463
154.239639388087
139.633953692907
137.347542674865
128.239459879571
121.107566199195
119.843262642657
116.685117326047
```

```
116.010477588934
109.830338231363
106.94301191915
106.890186229337
106.55297972109
103.20192549283
102.585009756697
...
```

Here we see that the COG is not as accurate as the SOG attribute. More than 100 trajectories have an azimuth difference bigger than 45 degrees. Figure 1.6 visualizes them. Some of them look like noise, but some look fine. For simplicity, we keep them all.

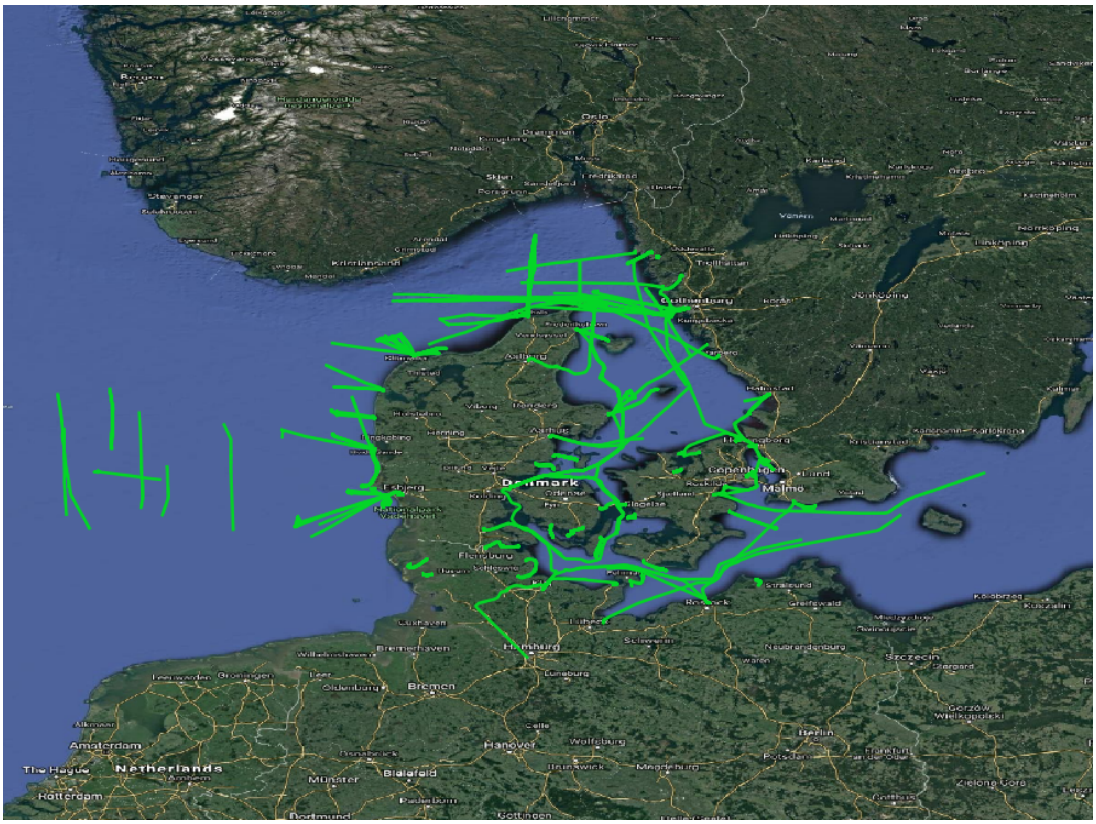

Figure 1.6: Ship trajectories with big difference between `azimuth(Trip)` and `COG`

1.8 Analyzing the Trajectories

Now we dive into MobilityDB and explore more of its functions. In Figure 1.7, we notice trajectories that keep going between Rødby and Puttgarden. Most probably, these are the ferries between the two ports. The task is simply to spot which Ships do so, and to count how many one way trips they did in this day. This is expressed in the following query:

```
CREATE INDEX Ships_Trip_Idx ON Ships USING GiST(Trip);

WITH Ports(Rodby, Puttgarden) AS (
  SELECT ST_MakeEnvelope(651135, 6058230, 651422, 6058548, 25832),
    ST_MakeEnvelope(644339, 6042108, 644896, 6042487, 25832)
)
SELECT S.*, Rodby, Puttgarden
FROM Ports P, Ships S
WHERE intersects(S.Trip, P.Rodby) AND intersects(S.Trip, P.Puttgarden)
```

```
--Total query runtime: 462 msec
--4 rows retrieved.
```

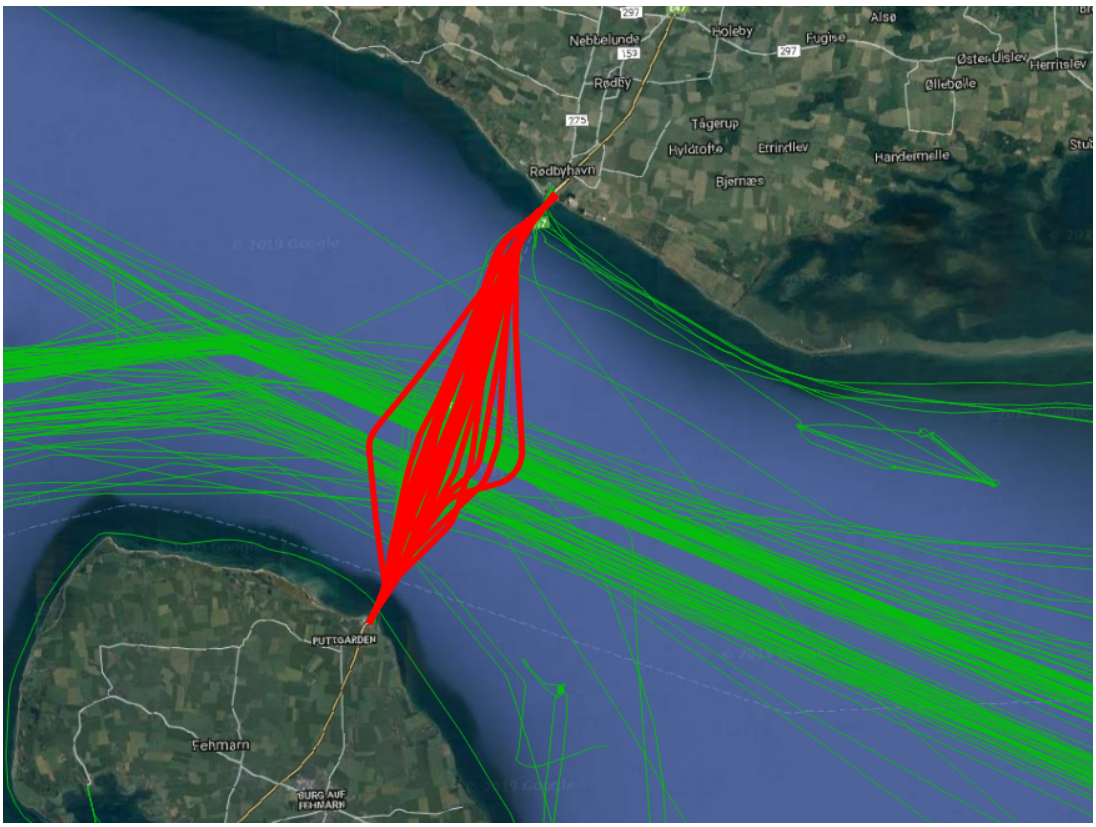

Figure 1.7: A sample ship trajectory between Rødby and Puttgarden

This query creates two envelope geometries that represent the locations of the two ports, then intersects them with the spatiotemporal trajectories of the ships. The `intersects` function checks whether a temporal point has ever intersects a geometry. To speed up the query, a spatiotemporal GiST index is first built on the `Trip` attribute. The query identified four Ships that commuted between the two ports, Figure 1.8. To count how many one way trips each of them did, we extend the previous query as follows:

```
WITH Ports(Rodby, Puttgarden) AS (
  SELECT ST_MakeEnvelope(651135, 6058230, 651422, 6058548, 25832),
    ST_MakeEnvelope(644339, 6042108, 644896, 6042487, 25832)
)
SELECT MMSI, (numSequences(atGeometry(S.Trip, P.Rodby)) +
  numSequences(atGeometry(S.Trip, P.Puttgarden)))/2.0 AS NumTrips
FROM Ports P, Ships S
WHERE intersects(S.Trip, P.Rodby) AND intersects(S.Trip, P.Puttgarden)
--Total query runtime: 1.1 secs

MMSI    NumTrips
219000429;  24.0
211188000;  24.0
211190000;  25.0
219000431;  16.0
```

The function `atGeometry` restricts the temporal point to the parts where it is inside the given geometry. The result is thus a temporal point that consists of multiple pieces (sequences), with temporal gaps in between. The function `numSequences` counts the number of these pieces.

Figure 1.8: All ferries between Rødby and Puttgarden

With this high number of ferry trips, one wonders whether there are collision risks with ships that traverse this belt (the green trips in Figure 1.7). To check this, we query whether a pair of ship come very close to one another as follows:

```
WITH B(Belt) AS (
  SELECT ST_MakeEnvelope(640730, 6058230, 654100, 6042487, 25832)
),
BeltShips AS (
  SELECT MMSI, atGeometry(S.Trip, B.Belt) AS Trip,
    trajectory(atGeometry(S.Trip, B.Belt)) AS Traj
  FROM Ships S, B
  WHERE intersects(S.Trip, B.Belt)
)
SELECT S1.MMSI, S2.MMSI, S1.Traj, S2.Traj, shortestLine(S1.tripETRS, S2.tripETRS) Approach
FROM BeltShips S1, BeltShips S2
WHERE S1.MMSI > S2.MMSI AND
  dwithin(S1.tripETRS, S2.tripETRS, 300)
--Total query runtime: 28.5 secs
--7 rows retrieved.
```

The query first defines the area of interest as an envelope, the red dashed line in Figure 1.9). It then restricts/crops the trajectories to only this envelope using the `atGeometry` function. The main query then find pairs of different trajectories that ever came within a distance of 300 meters to one another (the `dwithin`). For these trajectories, it computes the spatial line that connects the two instants where the two trajectories were closest to one another (the `shortestLine` function). Figure 1.9 shows the green trajectories that came close to the blue trajectories, and their shortest connecting line in solid red. Most of the approaches occur at the entrance of the Rødby port, which looks normal. But we also see two interesting approaches, that may indicate danger of collision away from the port. They are shown with more zoom in Figure 1.10 and Figure 1.11

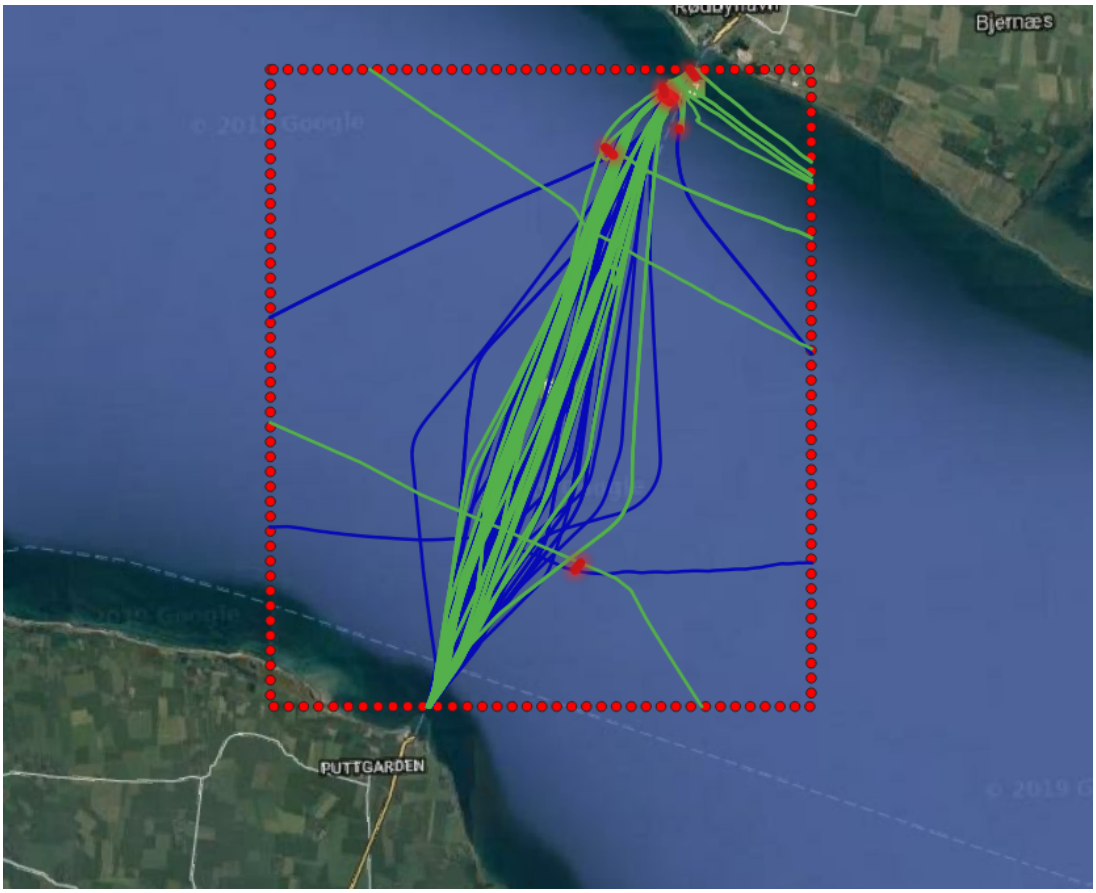

Figure 1.9: Ship that come closer than 300 meters to one another

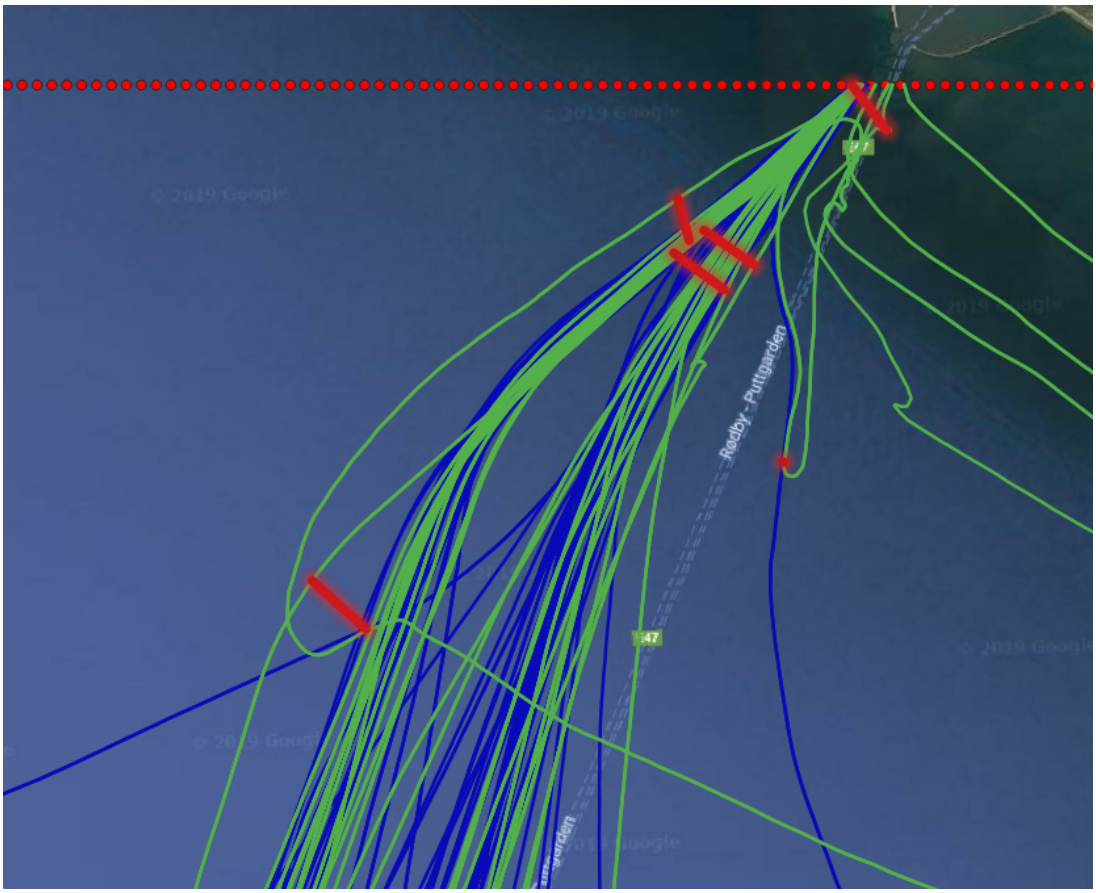

Figure 1.10: A zoom-in on a dangerous approach

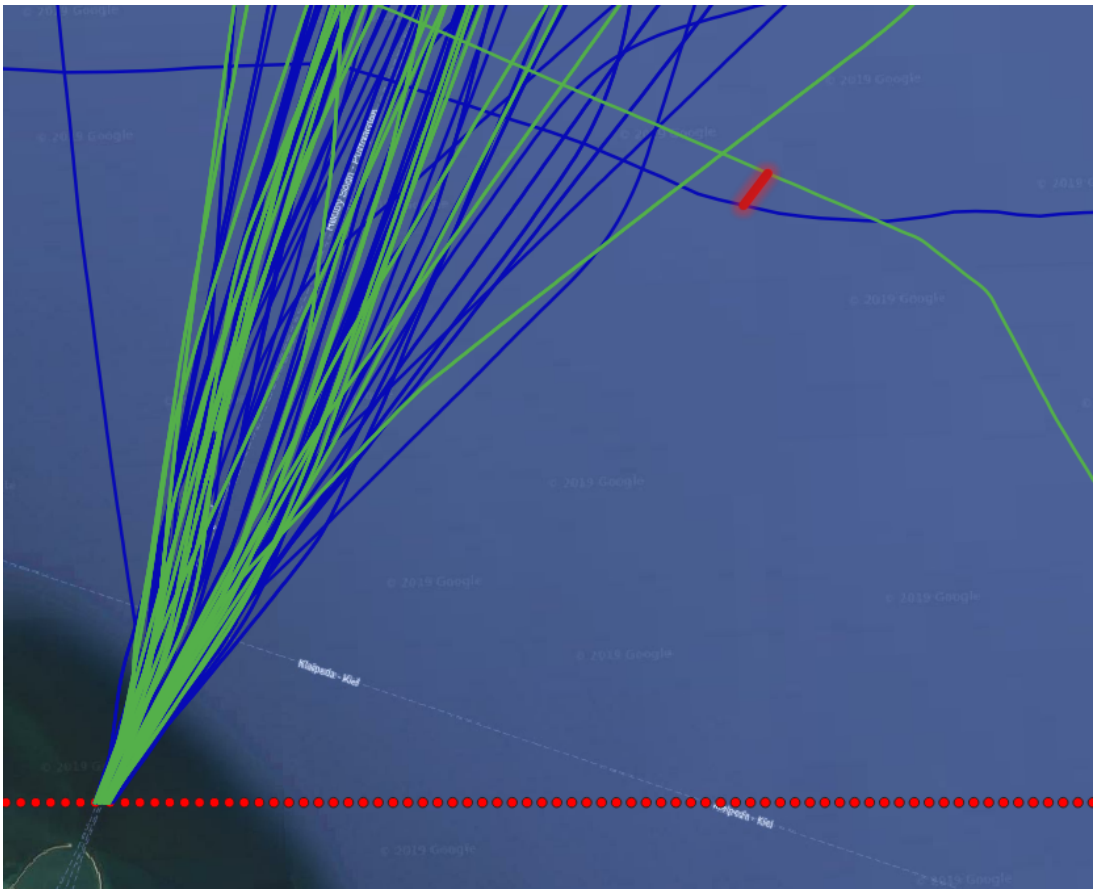

Figure 1.11: Another dangerous approach

Chapter 2

Managing GTFS Data

The General Transit Feed Specification (GTFS) defines a common format for public transportation schedules and associated geographic information. GTFS-realtime is used to specify real-time transit data. Many transportation agencies around the world publish their data in GTFS and GTFS-realtime format and make them publicly available. A well-known repository containing such data is OpenMobilityData.

In this chapter, we illustrate how to load GTFS data in MobilityDB. For this, we first need to import the GTFS data into PostgreSQL and then transform this data so that it can be loaded into MobilityDB. The data used in this tutorial is obtained from STIB-MIVB, the Brussels public transportation company and is available as a ZIP file. You must be aware that GTFS data is typically of big size. In order to reduce the size of the dataset, this file only contains schedules for one week and five transportation lines, whereas typical GTFS data published by STIB-MIVB contains schedules for one month and 99 transportation lines. In the reduced dataset used in this tutorial the final table containing the GTFS data in MobilityDB format has almost 10,000 trips and its size is 241 MB. Furtheremore, we need several temporary tables to transform GTFS format into MobilityDB and these tables are also big, the largest one has almost 6 million rows and its size is 621 MB.

Several tools can be used to import GTFS data into PostgreSQL. For example, one publicly available in Github can be found here. These tools load GTFS data into PostgreSQL tables, allowing one to perform multiple imports of data provided by the same agency covering different time frames, perform various complex tasks including data validation, and take into account variations of the format provided by different agencies, updates of route information among multiple imports, etc. For the purpose of this tutorial we do a simple import and transformation using only SQL. This is enough for loading the data set we are using but a much more robust solution should be used in an operational environment, if only for coping with the considerable size of typical GTFS data, which would require parallelization of this task.

2.1 Loading GTFS Data in PostgreSQL

The ZIP file with the data for this tutorial contains a set of CSV files (with extension `.txt`) as follows:

- `agency.txt` contains the description of the transportation agencies provinding the services (a single one in our case).
- `calendar.txt` contains service patterns that operate recurrently such as, for example, every weekday.
- `calendar_dates.txt` define exceptions to the default service patterns defined in `calendar.txt`. There are two types of exceptions: 1 means that the service has been added for the specified date, and 2 means that the service has been removed for the specified date.
- `route_types.txt` contains transportation types used on routes, such as bus, metro, tramway, etc.
- `routes.txt` contains transit routes. A route is a group of trips that are displayed to riders as a single service.
- `shapes.txt` contains the vehicle travel paths, which are used to generate the corresponding geometry.
- `stop_times.txt` contains times at which a vehicle arrives at and departs from stops for each trip.

- `translations.txt` contains the translation of the route information in French and Dutch. This file is not used in this tutorial.
- `trips.txt` contains trips for each route. A trip is a sequence of two or more stops that occur during a specific time period.

We decompress the file with the data into a directory. This can be done using the command.

```
unzip gtfs_data.zip
```

We suppose in the following that the directory used is as follows `/home/gtfs_tutorial/`.

We create the tables to be loaded with the data in the CSV files as follows.

```
CREATE TABLE agency (
  agency_id text DEFAULT '',
  agency_name text DEFAULT NULL,
  agency_url text DEFAULT NULL,
  agency_timezone text DEFAULT NULL,
  agency_lang text DEFAULT NULL,
  agency_phone text DEFAULT NULL,
  CONSTRAINT agency_pkey PRIMARY KEY (agency_id)
);

CREATE TABLE calendar (
  service_id text,
  monday int NOT NULL,
  tuesday int NOT NULL,
  wednesday int NOT NULL,
  thursday int NOT NULL,
  friday int NOT NULL,
  saturday int NOT NULL,
  sunday int NOT NULL,
  start_date date NOT NULL,
  end_date date NOT NULL,
  CONSTRAINT calendar_pkey PRIMARY KEY (service_id)
);
CREATE INDEX calendar_service_id ON calendar (service_id);

CREATE TABLE exception_types (
  exception_type int PRIMARY KEY,
  description text
);

CREATE TABLE calendar_dates (
  service_id text,
  date date NOT NULL,
  exception_type int REFERENCES exception_types(exception_type)
);
CREATE INDEX calendar_dates_dateidx ON calendar_dates (date);

CREATE TABLE route_types (
  route_type int PRIMARY KEY,
  description text
);

CREATE TABLE routes (
  route_id text,
  route_short_name text DEFAULT '',
  route_long_name text DEFAULT '',
  route_desc text DEFAULT '',
  route_type int REFERENCES route_types(route_type),
  route_url text,
  route_color text,
```

```
  route_text_color text,
  CONSTRAINT routes_pkey PRIMARY KEY (route_id)
);

CREATE TABLE shapes (
  shape_id text NOT NULL,
  shape_pt_lat double precision NOT NULL,
  shape_pt_lon double precision NOT NULL,
  shape_pt_sequence int NOT NULL
);
CREATE INDEX shapes_shape_key ON shapes (shape_id);

-- Create a table to store the shape geometries
CREATE TABLE shape_geoms (
  shape_id text NOT NULL,
  shape_geom geometry('LINESTRING', 4326),
  CONSTRAINT shape_geom_pkey PRIMARY KEY (shape_id)
);
CREATE INDEX shape_geoms_key ON shapes (shape_id);

CREATE TABLE location_types (
  location_type int PRIMARY KEY,
  description text
);

CREATE TABLE stops (
  stop_id text,
  stop_code text,
  stop_name text DEFAULT NULL,
  stop_desc text DEFAULT NULL,
  stop_lat double precision,
  stop_lon double precision,
  zone_id text,
  stop_url text,
  location_type integer  REFERENCES location_types(location_type),
  parent_station integer,
  stop_geom geometry('POINT', 4326),
  platform_code text DEFAULT NULL,
  CONSTRAINT stops_pkey PRIMARY KEY (stop_id)
);

CREATE TABLE pickup_dropoff_types (
  type_id int PRIMARY KEY,
  description text
);

CREATE TABLE stop_times (
  trip_id text NOT NULL,
  -- Check that casting to time interval works.
  arrival_time interval CHECK (arrival_time::interval = arrival_time::interval),
  departure_time interval CHECK (departure_time::interval = departure_time::interval),
  stop_id text,
  stop_sequence int NOT NULL,
  pickup_type int REFERENCES pickup_dropoff_types(type_id),
  drop_off_type int REFERENCES pickup_dropoff_types(type_id),
  CONSTRAINT stop_times_pkey PRIMARY KEY (trip_id, stop_sequence)
);
CREATE INDEX stop_times_key ON stop_times (trip_id, stop_id);
CREATE INDEX arr_time_index ON stop_times (arrival_time);
CREATE INDEX dep_time_index ON stop_times (departure_time);

CREATE TABLE trips (
```

```
  route_id text NOT NULL,
  service_id text NOT NULL,
  trip_id text NOT NULL,
  trip_headsign text,
  direction_id int,
  block_id text,
  shape_id text,
  CONSTRAINT trips_pkey PRIMARY KEY (trip_id)
);
CREATE INDEX trips_trip_id ON trips (trip_id);

INSERT INTO exception_types (exception_type, description) VALUES
  (1, 'service has been added'),
  (2, 'service has been removed');

INSERT INTO location_types(location_type, description) VALUES
  (0,'stop'),
  (1,'station'),
  (2,'station entrance');

INSERT INTO pickup_dropoff_types (type_id, description) VALUES
  (0,'Regularly Scheduled'),
  (1,'Not available'),
  (2,'Phone arrangement only'),
  (3,'Driver arrangement only');
```

We created one table for each CSV file. In addition, we created a table `shape_geoms` in order to assemble all segments composing a route into a single geometry and auxiliary tables `exception_types`, `location_types`, and `pickup_dropoff_types` containing acceptable values for some columns in the CSV files.

We can load the CSV files into the corresponding tables as follows.

```
COPY calendar(service_id,monday,tuesday,wednesday,thursday,friday,saturday,sunday,
  start_date,end_date) FROM '/home/gtfs_tutorial/calendar.txt' DELIMITER ',' CSV HEADER;
COPY calendar_dates(service_id,date,exception_type)
  FROM '/home/gtfs_tutorial/calendar_dates.txt' DELIMITER ',' CSV HEADER;
COPY stop_times(trip_id,arrival_time,departure_time,stop_id,stop_sequence,
  pickup_type,drop_off_type) FROM '/home/gtfs_tutorial/stop_times.txt' DELIMITER ','
  CSV HEADER;
COPY trips(route_id,service_id,trip_id,trip_headsign,direction_id,block_id,shape_id)
  FROM '/home/gtfs_tutorial/trips.txt' DELIMITER ',' CSV HEADER;
COPY agency(agency_id,agency_name,agency_url,agency_timezone,agency_lang,agency_phone)
  FROM '/home/gtfs_tutorial/agency.txt' DELIMITER ',' CSV HEADER;
COPY route_types(route_type,description)
  FROM '/home/gtfs_tutorial/route_types.txt' DELIMITER ',' CSV HEADER;
COPY routes(route_id,route_short_name,route_long_name,route_desc,route_type,route_url,
  route_color,route_text_color) FROM '/home/gtfs_tutorial/routes.txt' DELIMITER ','
  CSV HEADER;
COPY shapes(shape_id,shape_pt_lat,shape_pt_lon,shape_pt_sequence)
  FROM '/home/gtfs_tutorial/shapes.txt' DELIMITER ',' CSV HEADER;
COPY stops(stop_id,stop_code,stop_name,stop_desc,stop_lat,stop_lon,zone_id,stop_url,
  location_type,parent_station) FROM '/home/gtfs_tutorial/stops.txt' DELIMITER ','
  CSV HEADER;
```

Finally, we create the geometries for routes and stops as follows.

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

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

The visualization of the routes and stops in QGIS is given in Figure 2.1. In the figure, red lines correspond to the trajectories of vehicles, while orange points correspond to the location of stops.

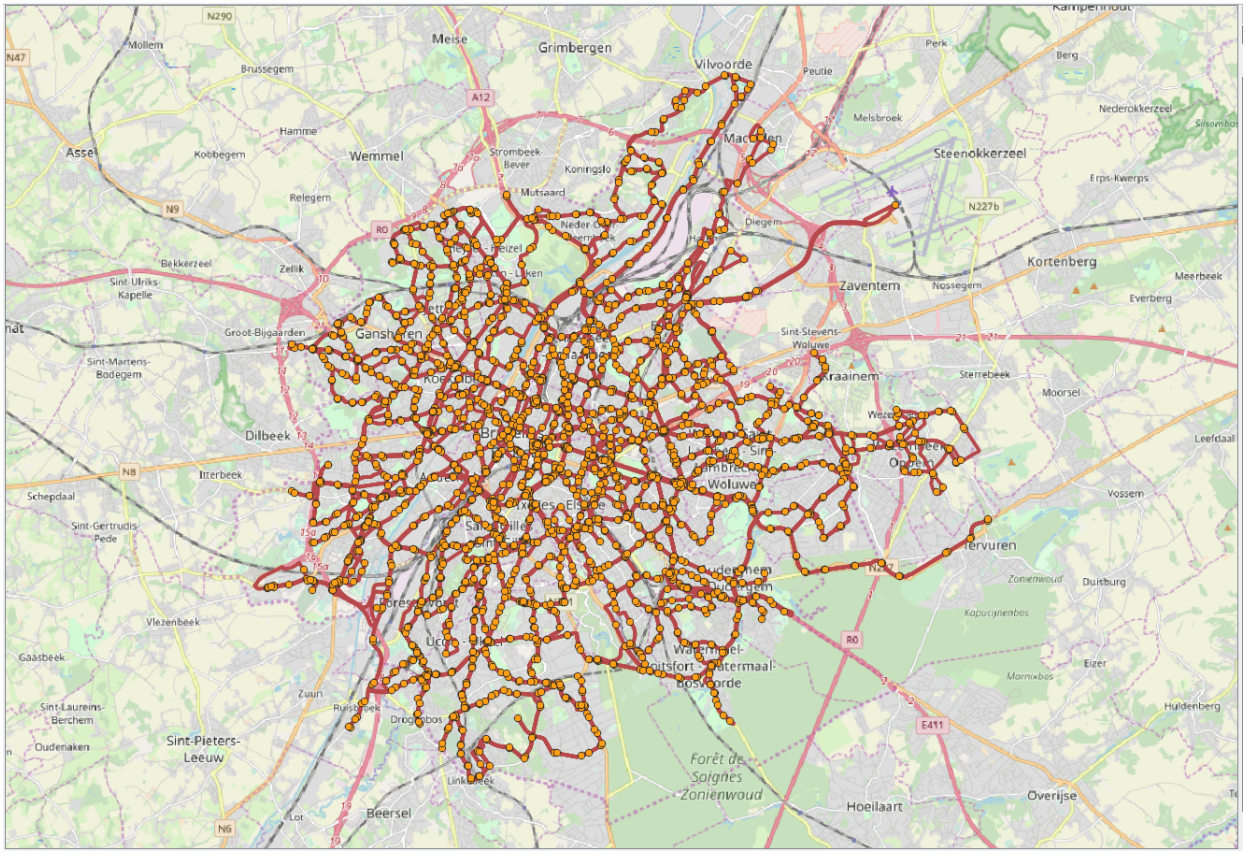

Figure 2.1: Visualization of the routes and stops for the GTFS data from Brussels.

2.2 Transforming GTFS Data for MobilityDB

We start by creating a table that contains couples of `service_id` and `date` defining the dates at which a service is provided.

```
DROP TABLE IF EXISTS service_dates;
CREATE TABLE service_dates AS (
  SELECT service_id, date_trunc('day', d)::date AS date
  FROM calendar c, generate_series(start_date, end_date, '1 day'::interval) AS d
  WHERE (
    (monday = 1 AND extract(isodow FROM d) = 1) OR
    (tuesday = 1 AND extract(isodow FROM d) = 2) OR
    (wednesday = 1 AND extract(isodow FROM d) = 3) OR
    (thursday = 1 AND extract(isodow FROM d) = 4) OR
    (friday = 1 AND extract(isodow FROM d) = 5) OR
    (saturday = 1 AND extract(isodow FROM d) = 6) OR
    (sunday = 1 AND extract(isodow FROM d) = 7)
  )
  EXCEPT
  SELECT service_id, date
  FROM calendar_dates WHERE exception_type = 2
```

```
  UNION
  SELECT c.service_id, date
  FROM calendar c JOIN calendar_dates d ON c.service_id = d.service_id
  WHERE exception_type = 1 AND start_date <= date AND date <= end_date
);
```

This table transforms the service patterns in the `calendar` table valid between a `start_date` and an `end_date` taking into account the week days, and then remove the exceptions of type 2 and add the exceptions of type 1 that are specified in table `calendar_dates`.

We now create a table `trip_stops` that determines the stops for each trip.

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

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

UPDATE trip_stops t
SET perc = CASE
  WHEN stop_sequence =  1 then 0.0
  WHEN stop_sequence =  no_stops then 1.0
  ELSE ST_LineLocatePoint(g.the_geom, s.the_geom)
END
FROM shape_geoms g, stops s
WHERE t.shape_id = g.shape_id AND t.stop_id = s.stop_id;
```

We perform a join between `trips` and `stop_times` and determines the number of stops in a trip. Then, we compute the relative location of a stop within a trip using the function `ST_LineLocatePoint`.

We now create a table `trip_segs` that defines the segments between two consecutive stops of a trip.

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

```
);

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

UPDATE trip_segs t
SET seg_geom = ST_LineSubstring(g.the_geom, perc1, perc2)
FROM shape_geoms g
WHERE t.shape_id = g.shape_id;

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

We use twice the `LEAD` window function for obtaning the next stop and the next percentage of a given stop and the `MAX` window function for obtaining the total number of stops in a trip. Then, we generate the geometry of the segment betwen two stops using the function `ST_LineSubstring` and compute the length and the number of points in the segment with functions `ST_Length` and `ST_NumPoints`.

The geometry of a segment is a linestring containing multiple points. From the previous table we know at which time the trip arrived at the first point and at the last point of the segment. To determine at which time the trip arrived at the intermediate points of the segments, we create a table `trip_points` that contains all the points composing the geometry of a segment.

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

INSERT INTO trip_points (trip_id, route_id, service_id, stop1_sequence,
  point_sequence, point_geom, point_arrival_time)
WITH temp1 AS (
  SELECT trip_id, route_id, service_id, stop1_sequence, stop2_sequence,
    no_stops, stop1_arrival_time, stop2_arrival_time, seg_length,
    (dp).path[1] AS point_sequence, no_points, (dp).geom as point_geom
  FROM trip_segs, ST_DumpPoints(seg_geom) AS dp
),
temp2 AS (
  SELECT trip_id, route_id, service_id, stop1_sequence, stop1_arrival_time,
    stop2_arrival_time, seg_length, point_sequence, no_points, point_geom
  FROM temp1
  WHERE point_sequence <> no_points OR stop2_sequence = no_stops
),
temp3 AS (
  SELECT trip_id, route_id, service_id, stop1_sequence, stop1_arrival_time,
    stop2_arrival_time, point_sequence, no_points, point_geom,
    ST_Length(ST_MakeLine(array_agg(point_geom) OVER w)) / seg_length AS perc
  FROM temp2 WINDOW w AS (PARTITION BY trip_id, service_id, stop1_sequence
    ORDER BY point_sequence)
```

```
)
SELECT trip_id, route_id, service_id, stop1_sequence, point_sequence, point_geom,
  CASE
  WHEN point_sequence = 1 then stop1_arrival_time
  WHEN point_sequence = no_points then stop2_arrival_time
  ELSE stop1_arrival_time + ((stop2_arrival_time - stop1_arrival_time) * perc)
  END AS point_arrival_time
FROM temp3;
```

In the temporary table `temp1` we use the function `ST_DumpPoints` to obtain the points composing the geometry of a segment. Nevertheless, this table contains duplicate points, that is, the last point of a segment is equal to the first point of the next one. In the temporary table `temp2` we filter out the last point of a segment unless it is the last segment of the trip. In the temporary table `temp3` we compute in the attribute `perc` the relative position of a point within a trip segment with window functions. For this we use the function `ST_MakeLine` to construct the subsegment from the first point of the segment to the current one, determine the length of the subsegment with function `ST_Length` and divide this length by the overall segment length. Finally, in the outer query we use the computed percentage to determine the arrival time to that point.

Our last temporary table `trips_input` contains the data in the format that can be used for creating the MobilityDB trips.

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

INSERT INTO trips_input
SELECT trip_id, route_id, t.service_id, date, point_geom, date + point_arrival_time AS t
FROM trip_points t JOIN
  ( SELECT service_id, MIN(date) AS date FROM service_dates GROUP BY service_id) s
  ON t.service_id = s.service_id;
```

In the inner query of the `INSERT` statement, we select the first date of a service in the `service_dates` table and then we join the resulting table with the `trip_points` table to compute the arrival time at each point composing the trips. Notice that we filter the first date of each trip for optimization purposes because in the next step below we use the `shift` function to compute the trips to all other dates. Alternatively, we could join the two tables but this will be considerably slower for big GTFS files.

Finally, table `trips_mdb` contains the MobilityDB trips.

```
DROP TABLE IF EXISTS trips_mdb;
CREATE TABLE trips_mdb (
  trip_id text NOT NULL,
  route_id text NOT NULL,
  date date NOT NULL,
  trip tgeompoint,
  PRIMARY KEY (trip_id, date)
);

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

INSERT INTO trips_mdb(trip_id, service_id, route_id, date, trip)
SELECT trip_id, route_id, t.service_id, d.date,
  shift(trip, make_interval(days => d.date - t.date))
FROM trips_mdb t JOIN service_dates d ON t.service_id = d.service_id AND t.date <> d.date;
```

In the first `INSERT` statement we group the rows in the `trips_input` table by `trip_id` and `date` while keeping the `route_id` atribute, use the `array_agg` function to construct an array containing the temporal points composing the trip ordered by time, and compute the trip from this array using the function `tgeompointseq`. As explained above, table `trips_input` only contains the first date of a trip. In the second `INSERT` statement we add the trips for all the other dates with the function `shift`.

Chapter 3

Managing Google Location History

3.1 Loading Google Location History Data

By activating the Location History in your Google account, you let Google track where you go with every mobile device. You can view and manage your Location History information through Google Maps Timeline. The data is provided in JSON format. An example of such a file is as follows.

```
{
  "locations" : [ {
    "timestampMs" : "1525373187756",
    "latitudeE7" : 508402936,
    "longitudeE7" : 43413790,
    "accuracy" : 26,
    "activity" : [ {
      "timestampMs" : "1525373185830",
      "activity" : [ {
        "type" : "STILL",
        "confidence" : 44
      }, {
        "type" : "IN_VEHICLE",
        "confidence" : 16
      }, {
        "type" : "IN_ROAD_VEHICLE",
        "confidence" : 16
      }, {
        "type" : "UNKNOWN",
        "confidence" : 12
      }, {
        "type" : "IN_RAIL_VEHICLE",
        "confidence" : 12
...
```

If we want to load location information into MobilityDB we only need the fields `longitudeE7`, `latitudeE7`, and `timestampMs`. To convert the original JSON file into a CSV file containing only these fields we can use jq, a command-line JSON processor. The following command

```
cat location_history.json | jq -r ".locations[] | {latitudeE7, longitudeE7, timestampMs}
| [.latitudeE7, .longitudeE7, .timestampMs] | @csv" > location_history.csv
```

produces a CSV file of the following format

```
508402936,43413790,"1525373187756"
508402171,43413455,"1525373176729"
508399229,43413304,"1525373143463"
```

```
508377525,43411499,"1525373113741"
508374906,43412597,"1525373082542"
508370337,43418136,"1525373052593"
...
```

The above command works well for files of moderate size since by default jq loads the whole input text in memory. For very large files you may consider the `--stream` option of jq, which parses input texts in a streaming fashion.

Now we can import the generated CSV file into PostgreSQL as follows.

```
DROP TABLE IF EXISTS location_history;
CREATE TABLE location_history (
  latitudeE7 float,
  longitudeE7 float,
  timestampMs bigint,
  date date
);

COPY location_history(latitudeE7, longitudeE7, timestampMs) FROM
  '/home/location_history/location_history.csv' DELIMITER ',' CSV;

UPDATE location_history
SET date = date(to_timestamp(timestampMs / 1000.0)::timestamptz);
```

Notice that we added an attribute `date` to the table so we can split the full location history, which can comprise data for several years, by date. Since the timestamps are encoded in milliseconds since 1/1/1970, we divide them by 1,000 and apply the functions `to_timestamp` and `date` to obtain corresponding date.

We can now transform this data into MobilityDB trips as follows.

```
DROP TABLE IF EXISTS locations_mdb;
CREATE TABLE locations_mdb (
  date date NOT NULL,
  trip tgeompoint,
  trajectory geometry,
  PRIMARY KEY (date)
);

INSERT INTO locations_mdb(date, trip)
SELECT date, tgeompointseq(array_agg(tgeompointinst(
  ST_SetSRID(ST_Point(longitudeE7/1e7, latitudeE7/1e7),4326),
  to_timestamp(timestampMs / 1000.0)::timestamptz) ORDER BY timestampMs))
FROM location_history
GROUP BY date;

UPDATE locations_mdb
SET trajectory = trajectory(trip);
```

We convert the longitude and latitude values into standard coordinates values by dividing them by 10^7. These can be converted into PostGIS points in the WGS84 coordinate system with the functions `ST_Point` and `ST_SetSRID`. Also, we convert the timestamp values in miliseconds to `timestamptz` values. We can now apply the function `tgeompointinst` to create a `tgeompoint` of instant duration from the point and the timestamp, collect all temporal points of a day into an array with the function `array_agg`, and finally, create a temporal point containing all the locations of a day using function `tgeompointseq`. We added to the table a `trajectory` attribute to visualize the location history in QGIS is given in Figure 3.1.

Figure 3.1: Visualization of the Google location history loaded into MobilityDB.

Chapter 4

Managing GPX Data

4.1 Loading GPX Data

GPX, or GPS Exchange Format, is an XML data format for GPS data. Location data (and optionally elevation, time, and other information) is stored in tags and can be interchanged between GPS devices and software. Conceptually, a GPX file contains tracks, which are a record of where a moving object has been, and routes, which are suggestions about where it might go in the future. Furthermore, both tracks and routes and composed by points. The following is a truncated (for brevity) example GPX file.

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<gpx version="1.1"
  xmlns="http://www.topografix.com/GPX/1/1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.topografix.com/GPX/1/1
  http://www.topografix.com/GPX/1/1/gpx.xsd"
  creator="Example creator">
  <metadata>
    <name>Dec 14, 2014 4:32:04 PM</name>
    <author>Example creator</author>
    <link href="https://..." />
    <time>2014-12-14T14:32:04.650Z</time>
  </metadata>
  <trk>
    <name>Dec 14, 2014 4:32:04 PM</name>
    <trkseg>
      <trkpt lat="30.16398" lon="31.467701">
        <ele>76</ele>
        <time>2014-12-14T14:32:10.339Z</time>
      </trkpt>
      <trkpt lat="30.16394" lon="31.467333">
        <ele>73</ele>
        <time>2014-12-14T14:32:16.00Z</time>
      </trkpt>
      <trkpt lat="30.16408" lon="31.467218">
        <ele>74</ele>
        <time>2014-12-14T14:32:19.00Z</time>
      </trkpt>
      [...]
    </trkseg>
    <trkseg>
      [...]
    </trkseg>
    [...]
  </trk>
```

```
  <trk>
    [...]
  </trk>
  [...]
<gpx>
```

The following Python program called `gpx_to_csv.py` uses `expat`, a stream-oriented XML parser library, to convert the above GPX file in CSV format.

```
import sys
import xml.parsers.expat

stack = []
def start_element(name, attrs):
  stack.append(name)
  if name == 'gpx' :
    print("lon,lat,time")
  if name == 'trkpt' :
    print("{},{},".format(attrs['lon'], attrs['lat']), end="")

def end_element(name):
  stack.pop()

def char_data(data):
  if stack[-1] == "time" and stack[-2] == "trkpt" :
    print(data)

p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.ParseFile(sys.stdin.buffer)
```

This Python program can be executed as follows.

```
python3 gpx_to_csv.py < example.gpx > example.csv
```

The resulting CSV file is given next.

```
lon,lat,time
31.46032,30.037502,2015-02-09T08:10:16.00Z
31.460901,30.039026,2015-02-09T08:10:31.00Z
31.461981,30.039816,2015-02-09T08:10:57.00Z
31.461996,30.039801,2015-02-09T08:10:58.00Z
...
```

The above CSV file can be loaded into MobilityDB as follows.

```
DROP TABLE IF EXISTS trips_input;
CREATE TABLE trips_input (
  date date,
  lon float,
  lat float,
  time timestamptz
);

COPY trips_input(lon, lat, time) FROM
  '/home/gpx_data/example.csv' DELIMITER ',' CSV HEADER;

UPDATE trips_input
```

```
SET date = date(time);

DROP TABLE IF EXISTS trips_mdb;
CREATE TABLE trips_mdb (
  date date NOT NULL,
  trip tgeompoint,
  trajectory geometry,
  PRIMARY KEY (date)
);

INSERT INTO trips_mdb(date, trip)
SELECT date, tgeompointseq(array_agg(tgeompointinst(
  ST_SetSRID(ST_Point(lon, lat),4326), time) ORDER BY time))
FROM trips_input
GROUP BY date;

UPDATE trips_mdb
SET trajectory = trajectory(trip);
```

Chapter 5

Generating Realistic Trajectory Datasets

5.1 Introduction

Do you need an arbitrarily large trajectory dataset to tests your ideas? The workshop module on Managing GTFS Data Chapter 2 has already illustrated how to generate public transport trajectories as per the schedule. This chapter continues and illustrates how to generate car trips in a city. It implements the BerlinMOD benchmark data generator that is described in:

Düntgen, C., Behr, T. and Güting, R.H. BerlinMOD: a benchmark for moving object databases. The VLDB Journal 18, 1335 (2009). https://doi.org/10.1007/s00778-009-0142-5

The data generator can be configured by setting the number of simulated cars and the number of simulation days. It models people trips using their cars to and from work during the week as well as some additional trips at evenings or weekends. The simulation uses multiple ideas to be close to reality, including:

- The home locations are sampled with respect to the population statistics of the different administrative areas in the city
- Similarly the work locations are sampled with respect to employment statistics
- Drivers will try to accelerate to the maximum allowed speed of a road
- Random events will force drivers to slow down or even stop to simulate obstacles, traffic lights, etc
- Drives will slow down in curves

The generator is written in PL/pgSQL, so that it will be easy to insert or adapt simulation rules to reflect other scenarios. It uses MobilityDB types and operations. The generated trajectories are also MobilityDB types. It is controlled by a single parameter, *scale factor*, that determines the size of the generated dataset. Additionally, many other parameters can be used to fine-tune the generation process to reflect various real-world simulation scenarios.

5.2 Contents

This module covers the following topics:

- A quick start using the generator
- Understanding the generation process
- Exploring the generated data
- Customizing the generator to your city
- Tuning the generator parameters
- Modifying the generator by changing the simulation scenario
- Creating a network topology from your own streets layer, to be used for the generator

5.3 Tools and Data

- MobilityDB, hence PostgreSQL and PostGIS. The installation instructions can be found here.
- pgRouting. The installation instructions can be found here. The minimum version required is 3.1.0.
- Download the quick start files here. Extract the archive in any folder. In the following we refer to this folder as generatorHome.

5.4 Quick Start

Running the generator is done in three steps:

Firstly, load the street network. Create a new database `brussels`, then add both PostGIS, MobilityDB, and pgRouting to it.

```
-- in a console:
createdb -h localhost -p 5432 -U dbowner brussels
-- replace localhost with your database host, 5432 with your port, and dbowner with your database user

psql -h localhost -p 5432 -U dbowner -d brussels -c 'CREATE EXTENSION MobilityDB CASCADE'
-- adds the PostGIS and the MobilityDB extensions to the database

psql -h localhost -p 5432 -U dbowner -d brussels -c 'CREATE EXTENSION pgRouting'
-- adds the pgRouting extension
```

For the moment, we will use the OSM map of Brussels. It is given in the data section of this workshop in the two files: `brussels.osm, mapconfig_brussels.xml`. In the next sections, we will explain how to use other maps. It has been downloaded using the Overpass API, hence it is by default in Spherical Mercator (SRID 3857), which is good for calculating distances. Next load the map and convert it into a routable network topology format suitable for pgRouting.

```
-- in a console, go to the generatorHome then:
osm2pgrouting -h localhost -p 5432 -U dbowner -f brussels.osm --dbname brussels -c mapconfig_brussels.xml
```

The configuration file `mapconfig_brussels.xml` tells osm2pgrouting which are the roads that will be selected to build the road network as well as the speed limits of the different road types. During the conversion, osm2pgrouting transforms the data into WGS84 (SRID 4326), so we will need later to convert it back to SRID 3857.

Secondly, prepare the base data for the simulation. Now, the street network is ready in the database. The simulation scenario requires to sample home and work locations. To make it realistic, we want to load a map of the administrative regions of Brussels (called communes) and feed the simulator with real population and employment statistics in every commune.

Load the administrative regions from the downloaded `brussels.osm` file, then run the `brussels_generatedata.sql` script using your PostgreSQL client, for example:

```
osm2pgsql -c -H localhost -P 5432 -U dbowner -d brussels brussels.osm
-- loads all layers in the osm file, including the adminstrative regions

psql -h localhost -p 5432 -U dbowner -d brussels -f brussels_preparedata.sql
-- samples home nodes and work nodes, transforms the projection back to SRID 3857, and does further data preparation
```

Finally, run the generator.

```
psql -h localhost -p 5432 -U dbowner -d brussels -f berlinmod_datagenerator_batch.sql
-- adds the pgplsql functions of the simulation to the database

psql -h localhost -p 5432 -U dbowner -d brussels -c 'select berlinmod_generate(scaleFactor := 0.005)'
-- calls the main pgplsql function to start the simulation
```

If everything is correct, you should see an output like that starts with this:

```
INFO:  ------------------------------------------------------------------
INFO:  Starting the BerlinMOD data generator with scale factor 0.005
INFO:  ------------------------------------------------------------------
INFO:  Parameters:
INFO:  ------------
INFO:  No. of vehicles = 141, No. of days = 4, Start day = 2020-06-01
INFO:  Path mode = Fastest Path, Disturb data = f
INFO:  Verbosity = minimal, Trip generation = C
...
```

The generator will take about one minute. It will generate trajectories, according to the default parameters, for 141 cars over 4 days starting from Monday, June 1st 2020. As you may have guessed, it is possible to generate more or less data by respectively passing a bigger or a smaller scale factor value. If you want to save the messages produced by the generator in a file you can use a command such as the following one.

```
psql -h localhost -p 5432 -U dbowner -d brussels -c
  "select berlinmod_generate(scaleFactor := 0.005)" 2>&1 | tee trace.txt
```

You can show more messages describing the generation process by setting the optional parameter `messages` with one of the values `'minimal'` (the default), `'medium'`, or `'verbose'`.

Figure 5.1 shows a visualization of the trips generated.

5.5 Understanding the Generation Process

We describe next the main steps in the generation of the BerlinMOD scenario. The generator uses multiple parameters that can be set to customize the generation process. We explain in detail these parameters in Section 5.8. It is worth noting that the procedures explained in this section have been slightly simplified with respect to the actual procedures by removing ancillary details concerning the generation of tracing messages at various verbosity levels.

We start by creating a first set of tables for containing the generated data as follows.

```
CREATE TABLE Vehicle(id int PRIMARY KEY, home bigint NOT NULL, work bigint NOT NULL,
  noNeighbours int);
CREATE TABLE Destinations(vehicle int, source bigint, target bigint,
  PRIMARY KEY (vehicle, source, target));
CREATE TABLE Licences(vehicle int PRIMARY KEY, licence text, type text, model text);
CREATE TABLE Neighbourhood(vehicle int, seq int, node bigint NOT NULL,
  PRIMARY KEY (vehicle, seq));

-- Get the number of nodes
SELECT COUNT(*) INTO noNodes FROM Nodes;

FOR i IN 1..noVehicles LOOP
  -- Fill the Vehicles table
  IF nodeChoice = 'Network Based' THEN
    homeNode = random_int(1, noNodes);
    workNode = random_int(1, noNodes);
  ELSE
    homeNode = berlinmod_selectHomeNode();
    workNode = berlinmod_selectWorkNode();
  END IF;
  IF homeNode IS NULL OR workNode IS NULL THEN
    RAISE EXCEPTION '    The home and the work nodes cannot be NULL';
  END IF;
  INSERT INTO Vehicle VALUES (i, homeNode, workNode);

  -- Fill the Destinations table
```

Figure 5.1: Visualization of the trips generated. The edges of the network are shown in blue, the edges traversed by the trips are shown in black, the home nodes in black and the work nodes in red.

```
  INSERT INTO Destinations(vehicle, source, target) VALUES
    (i, homeNode, workNode), (i, workNode, homeNode);

  -- Fill the Licences table
  licence = berlinmod_createLicence(i);
  type = berlinmod_vehicleType();
  model = berlinmod_vehicleModel();
  INSERT INTO Licences VALUES (i, licence, type, model);

  -- Fill the Neighbourhood table
  INSERT INTO Neighbourhood
  WITH Temp AS (
    SELECT i AS vehicle, N2.id AS node
    FROM Nodes N1, Nodes N2
    WHERE N1.id = homeNode AND N1.id <> N2.id AND
      ST_DWithin(N1.geom, N2.geom, P_NEIGHBOURHOOD_RADIUS)
  )
  SELECT i, ROW_NUMBER() OVER () as seq, node
  FROM Temp;
END LOOP;

CREATE UNIQUE INDEX Vehicle_id_idx ON Vehicle USING BTREE(id);
CREATE UNIQUE INDEX Neighbourhood_pkey_idx ON Neighbourhood USING BTREE(vehicle, seq);

UPDATE Vehicle V
SET noNeighbours = (SELECT COUNT(*) FROM Neighbourhood N WHERE N.vehicle = V.id);
```

We start by storing in the `Vehicles` table the home and the work node of each vehicle. Depending on the value of the variable `nodeChoice`, we chose these nodes either with a uniform distribution among all nodes in the network or we call specific functions that take into account population and employment statistics in the area covered by the generation. We then keep track in the `Destinations` table of the two trips to and from work and we store in the `Licences` table information describing the vehicle. Finally, we compute in the `Neighbourhood` table the set of nodes that are within a given distance of the home node of every vehicle. This distance is stated by the parameter `P_NEIGHBOURHOOD_RADIUS`, which is set by default to 3 Km.

We create now auxiliary tables containing benchmarking data. The number of rows these tables is determined by the parameter `P_SAMPLE_SIZE`, which is set by default to 100. These tables are used by the BerlinMOD benchmark to assess the performance of various types of queries.

```
CREATE TABLE QueryPoints(id int PRIMARY KEY, geom geometry(Point));
INSERT INTO QueryPoints
WITH Temp AS (
  SELECT id, random_int(1, noNodes) AS node
  FROM generate_series(1, P_SAMPLE_SIZE) id
)
SELECT T.id, N.geom
FROM Temp T, Nodes N
WHERE T.node = N.id;

CREATE TABLE QueryRegions(id int PRIMARY KEY, geom geometry(Polygon));
INSERT INTO QueryRegions
WITH Temp AS (
  SELECT id, random_int(1, noNodes) AS node
  FROM generate_series(1, P_SAMPLE_SIZE) id
)
SELECT T.id, ST_Buffer(N.geom, random_int(1, 997) + 3.0, random_int(0, 25)) AS geom
FROM Temp T, Nodes N
WHERE T.node = N.id;

CREATE TABLE QueryInstants(id int PRIMARY KEY, instant timestamptz);
INSERT INTO QueryInstants
SELECT id, startDay + (random() * noDays) * interval '1 day' AS instant
FROM generate_series(1, P_SAMPLE_SIZE) id;
```

```
CREATE TABLE QueryPeriods(id int PRIMARY KEY, period period);
INSERT INTO QueryPeriods
WITH Instants AS (
  SELECT id, startDay + (random() * noDays) * interval '1 day' AS instant
  FROM generate_series(1, P_SAMPLE_SIZE) id
)
SELECT id, Period(instant, instant + abs(random_gauss()) * interval '1 day',
  true, true) AS period
FROM Instants;
```

We generate now the leisure trips. There is at most one leisure trip in the evening of a week day and at most two leisure trips each day of the weekend, one in the morning and another one in the afternoon. Each leisure trip is composed of 1 to 3 destinations. The leisure trip starts and ends at the home node and visits successively these destinations. In our implementation, the various subtrips from a source to a destination node of a leisure trip are encoded independently, contrary to what is done in Secondo where a leisure trip is encoded as a single trip and stops are added between successive destinations.

```
CREATE TABLE LeisureTrip(vehicle int, day date, tripNo int, seq int, source bigint,
  target bigint, PRIMARY KEY (vehicle, day, tripNo, seq));
-- Loop for every vehicle
FOR i IN 1..noVehicles LOOP
  -- Get home node and number of neighbour nodes
  SELECT home, noNeighbours INTO homeNode, noNeigh
  FROM Vehicle V WHERE V.id = i;
  day = startDay;
  -- Loop for every generation day
  FOR j IN 1..noDays LOOP
    weekday = date_part('dow', day);
    -- Generate leisure trips (if any)
    -- 1: Monday, 5: Friday
    IF weekday BETWEEN 1 AND 5 THEN
      noLeisTrips = 1;
    ELSE
      noLeisTrips = 2;
    END IF;
    -- Loop for every leisure trip in a day (1 or 2)
    FOR k IN 1..noLeisTrips LOOP
      -- Generate a leisure trip with a 40% probability
      IF random() <= 0.4 THEN
        -- Select a number of destinations between 1 and 3
        IF random() < 0.8 THEN
          noDest = 1;
        ELSIF random() < 0.5 THEN
          noDest = 2;
        ELSE
          noDest = 3;
        END IF;
        sourceNode = homeNode;
        FOR m IN 1..noDest + 1 LOOP
          IF m <= noDest THEN
            targetNode = berlinmod_selectDestNode(i, noNeigh, noNodes);
          ELSE
            targetNode = homeNode;
          END IF;
          IF targetNode IS NULL THEN
            RAISE EXCEPTION '    Destination node cannot be NULL';
          END IF;
          INSERT INTO LeisureTrip VALUES
            (i, day, k, m, sourceNode, targetNode);
          INSERT INTO Destinations(vehicle, source, target) VALUES
            (i, sourceNode, targetNode) ON CONFLICT DO NOTHING;
          sourceNode = targetNode;
```

```
        END LOOP;
      END IF;
    END LOOP;
    day = day + 1 * interval '1 day';
  END LOOP;
END LOOP;

CREATE INDEX Destinations_vehicle_idx ON Destinations USING BTREE(vehicle);
```

For each vehicle and each day, we determine the number of potential leisure trips depending on whether it is a week or weekend day. A leisure trip is generated with a probability of 40% and is composed of 1 to 3 destinations. These destinations are chosen so that 80% of the destinations are from the neighbourhood of the vehicle and 20% are from the complete graph. The information about the composition of the leisure trips is then added to the `LeisureTrip` and `Destinations` tables.

We then call pgRouting to generate the path for each source and destination nodes in the `Destinations` table.

```
CREATE TABLE Paths(
  -- This attribute is needed for partitioning the table for big scale factors
  vehicle int,
  -- The following attributes are generated by pgRouting
  start_vid bigint, end_vid bigint, seq int, node bigint, edge bigint,
  -- The following attributes are filled from the Edges table
  geom geometry NOT NULL, speed float NOT NULL, category int NOT NULL,
  PRIMARY KEY (vehicle, start_vid, end_vid, seq));

-- Select query sent to pgRouting
IF pathMode = 'Fastest Path' THEN
  query1_pgr = 'SELECT id, source, target, cost_s AS cost,'
    'reverse_cost_s as reverse_cost FROM edges';
ELSE
  query1_pgr = 'SELECT id, source, target, length_m AS cost,'
    'length_m * sign(reverse_cost_s) as reverse_cost FROM edges';
END IF;
-- Get the total number of paths and number of calls to pgRouting
SELECT COUNT(*) INTO noPaths FROM (SELECT DISTINCT source, target FROM Destinations) AS T;
noCalls = ceiling(noPaths / P_PGROUTING_BATCH_SIZE::float);

FOR i IN 1..noCalls LOOP
  query2_pgr = format('SELECT DISTINCT source, target FROM Destinations '
    'ORDER BY source, target LIMIT %s OFFSET %s',
    P_PGROUTING_BATCH_SIZE, (i - 1) * P_PGROUTING_BATCH_SIZE);
  INSERT INTO Paths(vehicle, start_vid, end_vid, seq, node, edge, geom, speed, category)
  WITH Temp AS (
    SELECT start_vid, end_vid, path_seq, node, edge
    FROM pgr_dijkstra(query1_pgr, query2_pgr, true)
    WHERE edge > 0
  )
  SELECT D.vehicle, start_vid, end_vid, path_seq, node, edge,
    -- adjusting direction of the edge traversed
    CASE
      WHEN T.node = E.source THEN E.geom
      ELSE ST_Reverse(E.geom)
    END AS geom, E.maxspeed_forward AS speed,
    berlinmod_roadCategory(E.tag_id) AS category
  FROM Destinations D, Temp T, Edges E
  WHERE D.source = T.start_vid AND D.target = T.end_vid AND E.id = T.edge;
END LOOP;

CREATE INDEX Paths_vehicle_start_vid_end_vid_idx ON Paths USING
  BTREE(vehicle, start_vid, end_vid);
```

The variable `pathMode` determines whether pgRouting computes either the fastest or the shortest path from a source to a

destination node. Then, we determine the number of calls to pgRouting. Indeed, depending on the available memory of the computer, there is a limit in the number of paths to be computed by pgRouting in a single call. The paths are stored in the `Paths` table. In addition to the columns generated by pgRouting, we add the geometry (adjusting the direction if necessary), the maximum speed, and the category of the edge. The BerlinMOD data generator considers three road categories: side road, main road, and freeway. The OSM road types are mapped to one of these categories in the function `berlinmod_roadCategory`.

We are now ready to generate the trips.

```
DROP TYPE IF EXISTS step CASCADE;
CREATE TYPE step as (linestring geometry, maxspeed float, category int);

CREATE FUNCTION berlinmod_createTrips(noVehicles int, noDays int, startDay date,
  disturbData boolean)
RETURNS void LANGUAGE plpgsql STRICT AS $$
DECLARE
  /* Declaration of variables and parameters ... */
BEGIN
  DROP TABLE IF EXISTS Trips;
  CREATE TABLE Trips(vehicle int, day date, seq int, source bigint, target bigint,
    trip tgeompoint, trajectory geometry, PRIMARY KEY (vehicle, day, seq));
  -- Loop for each vehicle
  FOR i IN 1..noVehicles LOOP
    -- Get home -> work and work -> home paths
    SELECT home, work INTO homeNode, workNode
    FROM Vehicle V WHERE V.id = i;
    SELECT array_agg((geom, speed, category)::step ORDER BY seq) INTO homework
    FROM Paths WHERE vehicle = i AND start_vid = homeNode AND end_vid = workNode;
    SELECT array_agg((geom, speed, category)::step ORDER BY seq) INTO workhome
    FROM Paths WHERE vehicle = i AND start_vid = workNode AND end_vid = homeNode;
    d = startDay;
    -- Loop for each generation day
    FOR j IN 1..noDays LOOP
      weekday = date_part('dow', d);
      -- 1: Monday, 5: Friday
      IF weekday BETWEEN 1 AND 5 THEN
        -- Crete trips home -> work and work -> home
        t = d + time '08:00:00' + CreatePauseN(120);
        createTrip(homework, t, disturbData);
        INSERT INTO Trips VALUES (i, d, 1, homeNode, workNode, trip, trajectory(trip));
        t = d + time '16:00:00' + CreatePauseN(120);
        trip = createTrip(workhome, t, disturbData);
        INSERT INTO Trips VALUES (i, d, 2, workNode, homeNode, trip, trajectory(trip));
        tripSeq = 2;
      END IF;
      -- Get the number of leisure trips
      SELECT COUNT(DISTINCT tripNo) INTO noLeisTrip
      FROM LeisureTrip L
      WHERE L.vehicle = i AND L.day = d;
      -- Loop for each leisure trip (0, 1, or 2)
      FOR k IN 1..noLeisTrip LOOP
        IF weekday BETWEEN 1 AND 5 THEN
          t = d + time '20:00:00' + CreatePauseN(90);
          leisNo = 1;
        ELSE
          -- Determine whether it is a morning/afternoon (1/2) trip
          IF noLeisTrip = 2 THEN
            leisNo = k;
          ELSE
            SELECT tripNo INTO leisNo FROM LeisureTrip L
            WHERE L.vehicle = i AND L.day = d LIMIT 1;
          END IF;
          -- Determine the start time
```

```
          IF leisNo = 1 THEN
            t = d + time '09:00:00' + CreatePauseN(120);
          ELSE
            t = d + time '17:00:00' + CreatePauseN(120);
          END IF;
        END IF;
        -- Get the number of subtrips (number of destinations + 1)
        SELECT count(*) INTO noSubtrips
        FROM LeisureTrip L
        WHERE L.vehicle = i AND L.tripNo = leisNo AND L.day = d;
        FOR m IN 1..noSubtrips LOOP
          -- Get the source and destination nodes of the subtrip
          SELECT source, target INTO sourceNode, targetNode
          FROM LeisureTrip L
          WHERE L.vehicle = i AND L.day = d AND L.tripNo = leisNo AND L.seq = m;
          -- Get the path
          SELECT array_agg((geom, speed, category)::step ORDER BY seq) INTO path
          FROM Paths P
          WHERE vehicle = i AND start_vid = sourceNode AND end_vid = targetNode;
          trip = createTrip(path, t, disturbData);
          tripSeq = tripSeq + 1;
          INSERT INTO Trips VALUES
            (i, d, tripSeq, sourceNode, targetNode, trip, trajectory(trip));
          -- Add a delay time in [0, 120] min using a bounded Gaussian distribution
          t = endTimestamp(trip) + createPause();
        END LOOP;
      END LOOP;
      d = d + 1 * interval '1 day';
    END LOOP;
  END LOOP;
  RETURN;
END; $$
```

We create a type `step` which is a record composed of the geometry, the maximum speed, and the category of an edge. The procedure loops for each vehicle and each day and calls the procedure `createTrip` for creating the trips. If the day is a weekday, we generate the trips from home to work and from work to home starting, respectively, at 8 am and 4 pm plus a random non-zero duration of 120 minutes using a uniform distribution. We then generate the leisure trips. During the week days, the possible evening leisure trip starts at 8 pm plus a random random non-zero duration of 90 minutes, while during the weekend days, the two possible morning and afternoon trips start, respectively, at 9 am and 5 pm plus a random non-zero duration of 120 minutes. Between the multiple destinations of a leisure trip we add a delay time of maximum 120 minutes using a bounded Gaussian distribution.

Finally, we explain the procedure that create a trip.

```
CREATE OR REPLACE FUNCTION createTrip(edges step[], startTime timestamptz,
  disturbData boolean)
RETURNS tgeompoint LANGUAGE plpgsql STRICT AS $$
DECLARE
  /* Declaration of variables and parameters ... */
BEGIN
  srid = ST_SRID((edges[1]).linestring);
  p1 = ST_PointN((edges[1]).linestring, 1); x1 = ST_X(p1); y1 = ST_Y(p1);
  curPos = p1; t = startTime;
  instants[1] = tgeompointinst(p1, t);
  curSpeed = 0; l = 2; noEdges = array_length(edges, 1);
  -- Loop for every edge
  FOR i IN 1..noEdges LOOP
    -- Get the information about the current edge
    linestring = (edges[i]).linestring; maxSpeedEdge = (edges[i]).maxSpeed;
    category = (edges[i]).category;
    -- Determine the number of segments
    SELECT array_agg(geom ORDER BY path) INTO points
```

```
    FROM ST_DumpPoints(linestring);
    noSegs = array_length(points, 1) - 1;
    -- Loop for every segment
    FOR j IN 1..noSegs LOOP
      p2 = points[j + 1]; x2 = ST_X(p2); y2 = ST_Y(p2);
      -- If there is a segment ahead in the current edge compute the angle of the turn
      -- and the maximum speed at the turn depending on this angle
      IF j < noSegs THEN
        p3 = points[j + 2];
        alpha = degrees(ST_Angle(p1, p2, p3));
        IF abs(mod(alpha::numeric, 360.0)) < P_EPSILON THEN
          maxSpeedTurn = maxSpeedEdge;
        ELSE
          maxSpeedTurn = mod(abs(alpha - 180.0)::numeric, 180.0) / 180.0 * maxSpeedEdge;
        END IF;
      END IF;
      -- Determine the number of fractions
      segLength = ST_Distance(p1, p2);
      IF segLength < P_EPSILON THEN
        RAISE EXCEPTION 'Segment % of edge % has zero length', j, i;
      END IF;
      fraction = P_EVENT_LENGTH / segLength;
      noFracs = ceiling(segLength / P_EVENT_LENGTH);
      -- Loop for every fraction
      k = 1;
      WHILE k < noFracs LOOP
        -- If the current speed is zero, apply an acceleration event
        IF curSpeed >= P_EPSILON_SPEED THEN
          -- If we are not approaching a turn
          IF k < noFracs THEN
            curSpeed = least(P_EVENT_ACC, maxSpeedEdge);
          ELSE
            curSpeed = least(P_EVENT_ACC, maxSpeedTurn);
          END IF;
        ELSE
          -- If the current speed is not zero, apply a deceleration or a stop event
          -- with a probability proportional to the maximun speed
          IF random() <= P_EVENT_C / maxSpeedEdge THEN
            IF random() <= P_EVENT_P THEN
              -- Apply a stop event
              curSpeed = 0.0;
            ELSE
              -- Apply a deceleration event
              curSpeed = curSpeed * random_binomial(20, 0.5) / 20.0;
            END IF;
          ELSE
            -- Otherwise, apply an acceleration event
            IF k = noFracs AND j < noSegs THEN
              maxSpeed = maxSpeedTurn;
            ELSE
              maxSpeed = maxSpeedEdge;
            END IF;
            curSpeed = least(curSpeed + P_EVENT_ACC, maxSpeed);
          END IF;
        END IF;
        -- If speed is zero add a wait time
        IF curSpeed < P_EPSILON_SPEED THEN
          waitTime = random_exp(P_DEST_EXPMU);
          IF waitTime < P_EPSILON THEN
            waitTime = P_DEST_EXPMU;
          END IF;
          t = t + waitTime * interval '1 sec';
```

```
        ELSE
          -- Otherwise, move current position towards the end of the segment
          IF k < noFracs THEN
            x = x1 + ((x2 - x1) * fraction * k);
            y = y1 + ((y2 - y1) * fraction * k);
            IF disturbData THEN
              dx = (2 * P_GPS_STEPMAXERR * rand()) - P_GPS_STEPMAXERR;
              dy = (2 * P_GPS_STEPMAXERR * rand()) - P_GPS_STEPMAXERR;
              errx = errx + dx; erry = erry + dy;
              IF errx > P_GPS_TOTALMAXERR THEN
                errx = P_GPS_TOTALMAXERR;
              END IF;
              IF errx < - 1 * P_GPS_TOTALMAXERR THEN
                errx = -1 * P_GPS_TOTALMAXERR;
              END IF;
              IF erry > P_GPS_TOTALMAXERR THEN
                erry = P_GPS_TOTALMAXERR;
              END IF;
              IF erry < -1 * P_GPS_TOTALMAXERR THEN
                erry = -1 * P_GPS_TOTALMAXERR;
              END IF;
              x = x + dx; y = y + dy;
            END IF;
            curPos = ST_SetSRID(ST_Point(x, y), srid);
            curDist = P_EVENT_LENGTH;
          ELSE
            curPos = p2;
            curDist = segLength - (segLength * fraction * (k - 1));
          END IF;
          travelTime = (curDist / (curSpeed / 3.6));
          IF travelTime < P_EPSILON THEN
            travelTime = P_DEST_EXPMU;
          END IF;
          t = t + travelTime * interval '1 sec';
          k = k + 1;
        END IF;
        instants[l] = tgeompointinst(curPos, t);
        l = l + 1;
      END LOOP;
      p1 = p2; x1 = x2; y1 = y2;
    END LOOP;
    -- If we are not already in a stop, apply a stop event with a probability
    -- depending on the category of the current edge and the next one (if any)
    IF curSpeed < P_EPSILON_SPEED AND i < noEdges THEN
      nextCategory = (edges[i + 1]).category;
      IF random() <= P_DEST_STOPPROB[category][nextCategory] THEN
        curSpeed = 0;
        waitTime = random_exp(P_DEST_EXPMU);
        IF waitTime < P_EPSILON THEN
          waitTime = P_DEST_EXPMU;
        END IF;
        t = t + waitTime * interval '1 sec';
        instants[l] = tgeompointinst(curPos, t);
        l = l + 1;
      END IF;
    END IF;
  END LOOP;
  RETURN tgeompointseq(instants, true, true, true);
END; $$
```

The procedure receives as first argument a path from a source to a destination node, which is an array of triples composed of the

geometry, the maximum speed, and the category of an edge of the path. The other arguments are the timestamp at which the trip starts and a Boolean value determining whether the points composed the trip are disturbed to simulate GPS errors. The output of the function is a temporal geometry point following this path. The procedure loops for each edge of the path and determines the number of segments of the edge, where a segment is a straight line defined by two consecutive points. For each segment, we determine the angle between the current segment and the next one (if any) to compute the maximum speed at the turn. This is determined by multiplying the maximum speed of the segment by a factor proportional to the angle so that the factor is 1.00 at both 0° and 360° and is 0.0 at 180°. Examples of values of degrees and the associated factor are given next.

```
0: 1.00, 5: 0.97, 45: 0.75, 90: 0.50, 135: 0.25, 175: 0.03
180: 0.00, 185: 0.03, 225: 0.25, 270: 0.50, 315: 0.75, 355: 0.97, 360: 0.00
```

Each segment is divided in fractions of length `P_EVENT_LENGTH`, which is by default 5 meters. We then loop for each fraction and choose to add one event that can be an acceleration, a deceleration, or a stop event. If the speed of the vehicle is zero, only an accelation event can happen. For this, we increase the current speed with the value of `P_EVENT_ACC`, which is by default 12 Km/h, and verify that the speed is not greater than the maximum speed of either the edge or the next turn for the last fraction. Otherwise, if the current speed is not zero, we apply a deceleration or a stop event with a probability proportional to the maximum speed of the edge, otherwise we apply an acceleration event. After applying the event, if the speed is zero we add a waiting time with a random exponential distribution with mean `P_DEST_EXPMU`, which is by default 1 second. Otherwise, we move the current position towards the end of the segment and, depending on the variable `disturbData`, we disturbe the new position to simulate GPS errors. The timestamp at which the vehicle reaches the new position is determined by dividing the distance traversed by the current speed. Finally, at the end of each segment, if the current speed is not zero, we add a stop event depending on the categories of the current segment and the next one. This is determined by a transition matrix given by the parameter `P_DEST_STOPPROB`.

5.6 Exploring the Generated Data

We start by obtaining some statistics about the number, the total duration, and the total length in Km of the trips.

```
SELECT COUNT(*), SUM(timespan(Trip)), SUM(length(Trip)) / 1e3
FROM Trips;

1649  "743:54:57.020944"  27310.3360226806
```

We continue by further analyzing the duration of all the trips

```
SELECT MIN(timespan(Trip)), MAX(timespan(Trip)), AVG(timespan(Trip))
FROM Trips;

"00:00:13.659525" "01:19:22.711481" "00:27:05.609397"
```

or the duration of the trips by trip type.

```
SELECT
  CASE
    WHEN T.source = V.home AND date_part('dow', T.day) BETWEEN 1 AND 5 AND
      date_part('hour', startTimestamp(trip)) < 12 THEN 'home_work'
    WHEN T.source = V.work AND date_part('dow', T.day) BETWEEN 1 AND 5 AND
      date_part('hour', startTimestamp(trip)) > 12  THEN 'work_home'
    WHEN date_part('dow', T.day) BETWEEN 1 AND 5 THEN 'leisure_weekday'
    ELSE 'leisure_weekend'
  END AS TripType, COUNT(*), MIN(timespan(Trip)), MAX(timespan(Trip)), AVG(timespan(Trip))
FROM Trips T, Vehicle V
WHERE T.vehicle = V.id
GROUP BY TripType;

"work_home" 564 "00:02:51.833677" "01:15:28.972713" "00:33:34.064552"
"home_work" 564 "00:02:54.324099" "01:19:22.711481" "00:33:38.269437"
"leisure_weekday" 519 "00:00:13.659525" "01:08:31.188517" "00:12:56.76745"
```

As can be seen, no weekend leisure trips have been generated, which is normal since the data generated covers four days starting on Monday, June 1st 2020.

We can analyze further the length in Km of the trips as follows.

```
SELECT MIN(length(Trip)) / 1e3, MAX(length(Trip)) / 1e3, AVG(length(Trip)) / 1e3
FROM Trips;

0.0673650791287277  56.8296759391481  16.5617562296426
```

As can be seen the longest trip is more than 56 Km long. We investigate which is this trip with the following query.

```
SELECT vehicle, source, target, round(length(Trip)::numeric / 1e3, 3),
  startTimestamp(Trip), timespan(Trip)
FROM Trips
ORDER BY  round(length(Trip)::numeric / 1e3, 3) DESC, vehicle LIMIT 10;

119 32622 16501 56.830  "2020-06-01 17:18:32.471+02"  "01:09:20.124674"
119 32622 16501 56.830  "2020-06-02 16:07:53.498+02"  "01:08:41.880905"
119 32622 16501 56.830  "2020-06-03 17:53:15.174+02"  "01:09:24.495838"
119 32622 16501 56.830  "2020-06-04 17:15:37.33+02"   "01:09:40.474852"
77  1093  20895 55.234  "2020-06-01 08:55:57.454+02"  "01:08:32.461374"
...
```

As can be seen, the longest trips are the work trips of vehicle 119. They are depicted in Figure 5.2, and we realize that the home and the work nodes of the vehicle are located at two extremities in Brussels.

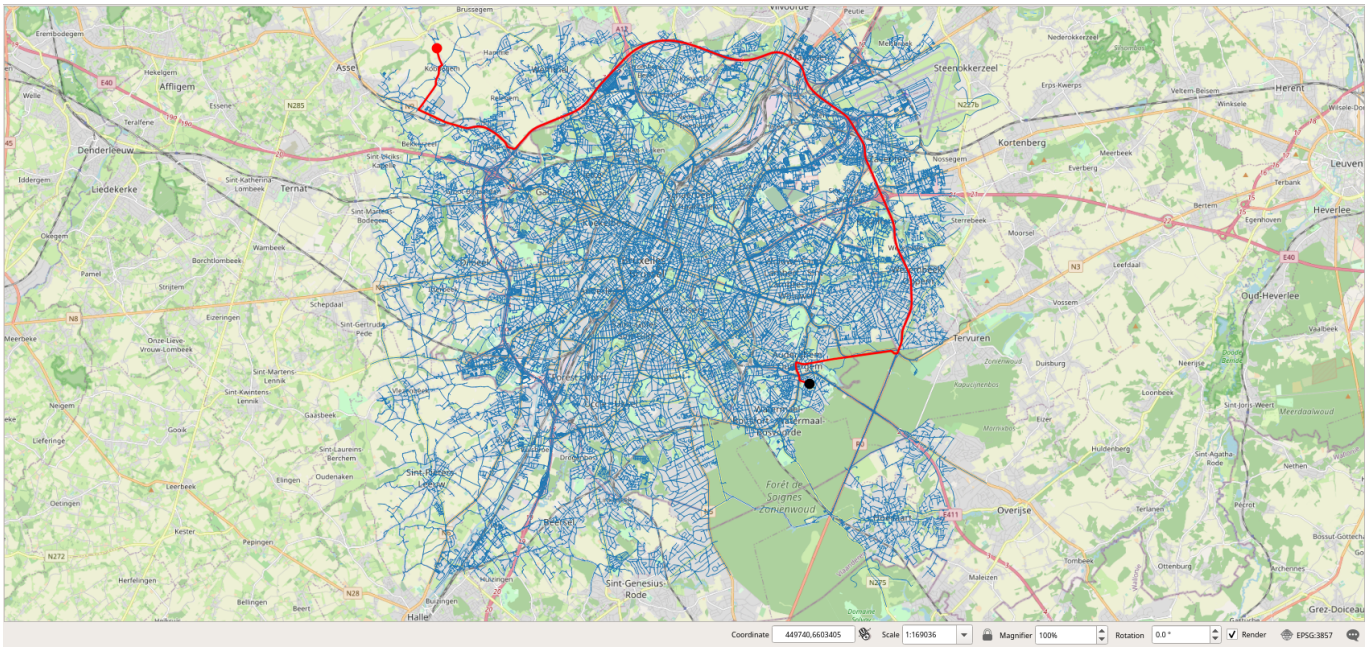

Figure 5.2: Visualization of the longest trip.

We can obtain some statistics about the average speed in Km/h of all the trips as follows.

```
SELECT MIN(twavg(speed(Trip))) * 3.6, MAX(twavg(speed(Trip))) * 3.6,
  AVG(twavg(speed(Trip))) * 3.6
FROM Trips;

12.0104235772758  58.0295976075198  34.0267418203071
```

A possible visualization that we could envision is to use gradients to show how the edges of the network are used by the trips. We start by determining how many trips traversed each of the edges of the network as follows.

```
CREATE TABLE HeatMap AS
SELECT  E.id, E.geom, count(*)
FROM Edges E, Trips T
WHERE st_intersects(E.geom, T.trajectory)
GROUP BY E.id, E.geom;
```

This is an expensive query since it took 42 min in my laptop. In order to display unused edges in our visualization we need to add them to the table with a count of 0.

```
INSERT INTO HeatMap
SELECT E.id, E.geom, 0 FROM Edges E WHERE E.id NOT IN (
  SELECT id FROM HeatMap );
```

We need some basic statistics about the attribute `count` in order to define the gradients.

```
SELECT min(count), max(count), round(avg(count),3), round(stddev(count),3) FROM HeatMap;
-- 0 204 4.856 12.994
```

Although the maximum value is 204, the average and the standard deviation are, respectively, around 5 and 13.

In order to display in QGIS the edges of the network with a gradient according to the attribute `count`, we use the following expression.

```
ramp_color('RdGy', scale_linear(count, 0, 10, 0, 1))
```

The `scale_linear` function transforms the value of the attribute `count` into a value in [0,1], as stated by the last two parameters. As stated by the two other parameters 0 and 10, which define the range of values to transform, we decided to assign a full red color to an edge as soon as there are at least 10 trips that traverse the edge. The `ramp_color` function states the gradient to be used for the display, in our case from blue to red. The usage of this expression in QGIS is shown in Figure 5.3 and the resulting visualization is shown in Figure 5.4.

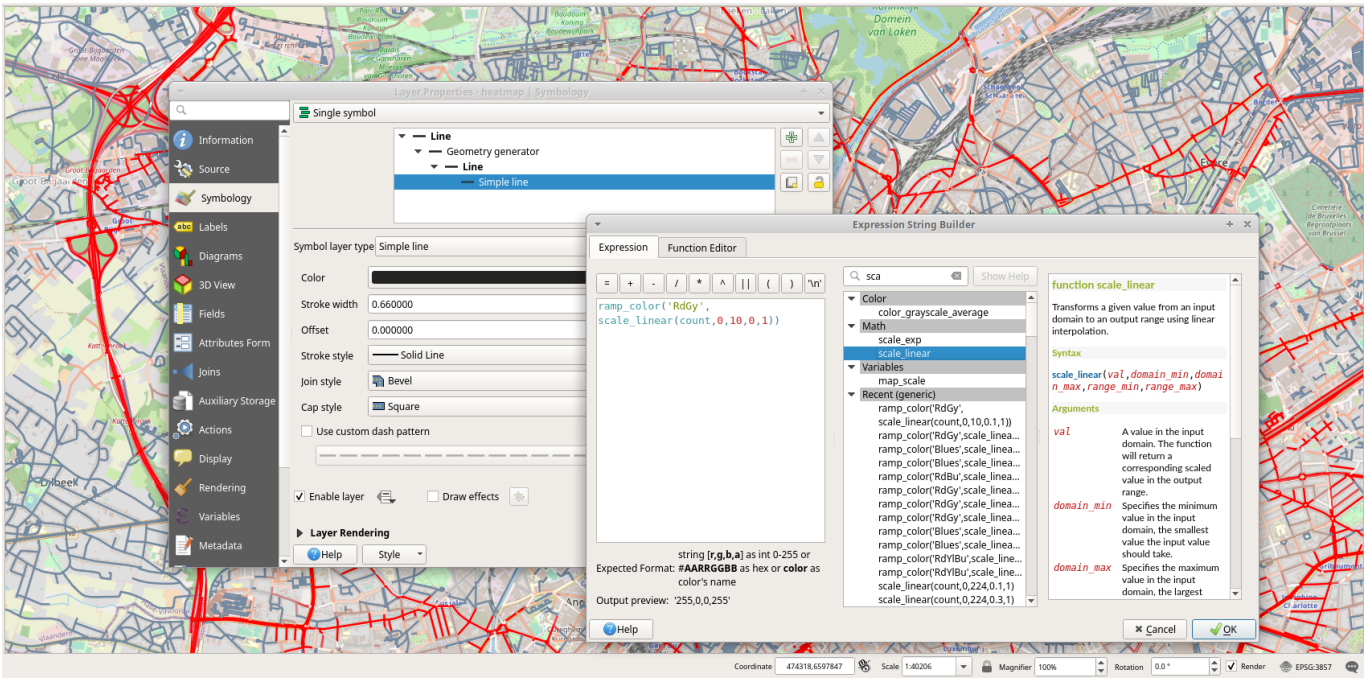

Figure 5.3: Assigning in QGIS a gradient color from blue to red according to the value of the attribute `count`.

Another possible visualization is to use gradients to show the speed used by the trips to traverse the edges of the network. As the maximum speed of edges varies from 20 to 120 Km/h, what would be interesting to compare is the speed of the trips at an edge with respect to the maximum speed of the edge. For this we issue the following query.

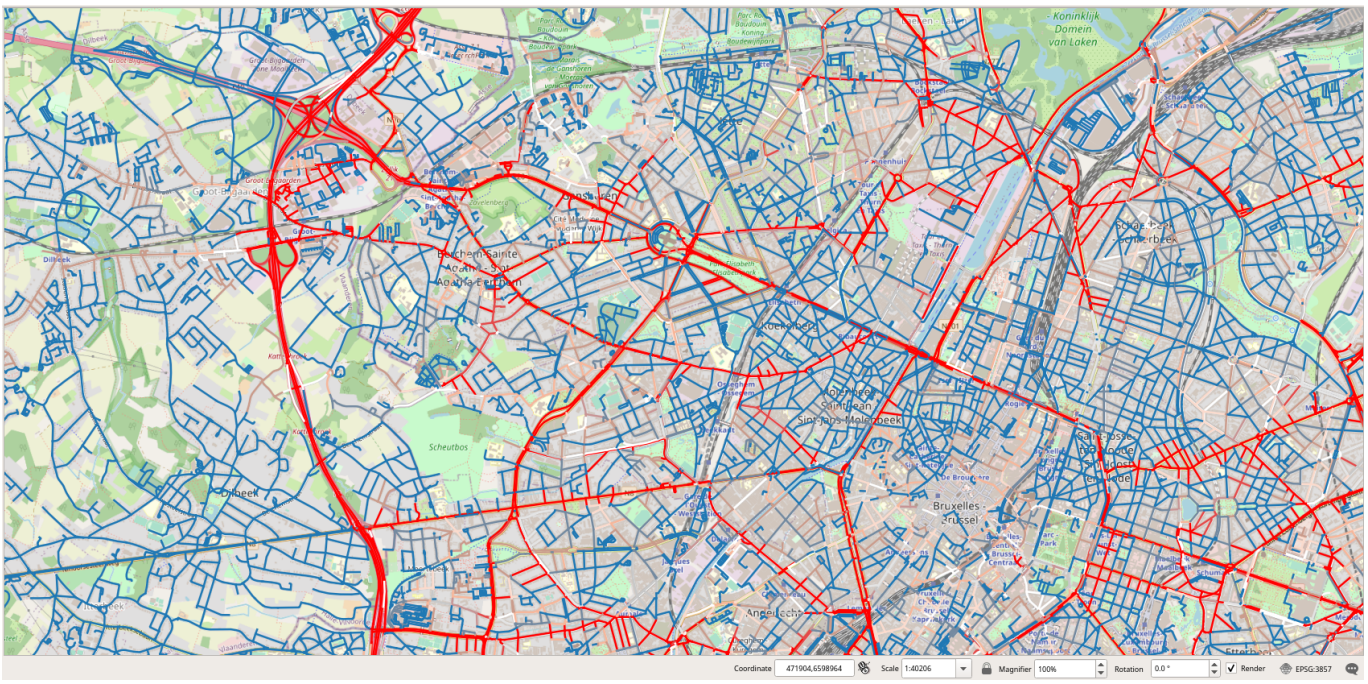

Figure 5.4: Visualization of the edges of the graph according to the number of trips that traversed the edges.

```
DROP TABLE IF EXISTS EdgeSpeed;
CREATE TABLE EdgeSpeed AS
SELECT P.edge, twavg(speed(atGeometry(T.trip, ST_Buffer(P.geom, 0.1)))) * 3.6 AS twavg
FROM Trips T, Paths P
WHERE T.source = P.start_vid AND T.target = P.end_vid AND P.edge > 0
ORDER BY P.edge;
```

This is an even more expensive query than the previous one since it took more than 2 hours in my laptop. Given a trip and an edge, the query restricts the trip to the geometry of the edge and computes the time-weighted average of the speed. Notice that the `ST_Buffer` is used to cope with the floating-point precision. After that we can compute the speed map as follows.

```
CREATE TABLE SpeedMap AS
WITH Temp AS (
  SELECT edge, avg(twavg) FROM EdgeSpeed GROUP BY edge
)
SELECT id, maxspeed_forward AS maxspeed, geom, avg, avg / maxspeed_forward AS perc
FROM Edges E, Temp T
WHERE E.id = T.edge;
```

Figure 5.5 shows the visualization of the speed map without and with the base map.

5.7 Customizing the Generator to Your City

In order to customize the generator to a particular city the only thing we need is to define a bounding box that will be used to download the data from OSM. There are many ways to obtain such a bounding box, and a typical way to proceed is to use one of the multiple online services that allows one to visually define a bounding box over a map. Figure 5.6 shows how we can define the bounding box around Barcelona using the web site bboxfinder.

After obtaining the bounding box, we can proceed as we stated in Section 5.4. We create a new database `barcelona`, then add both PostGIS, MobilityDB, and pgRouting to it.

Figure 5.5: Visualization of the edges of the graph according to the speed of trips that traversed the edges.

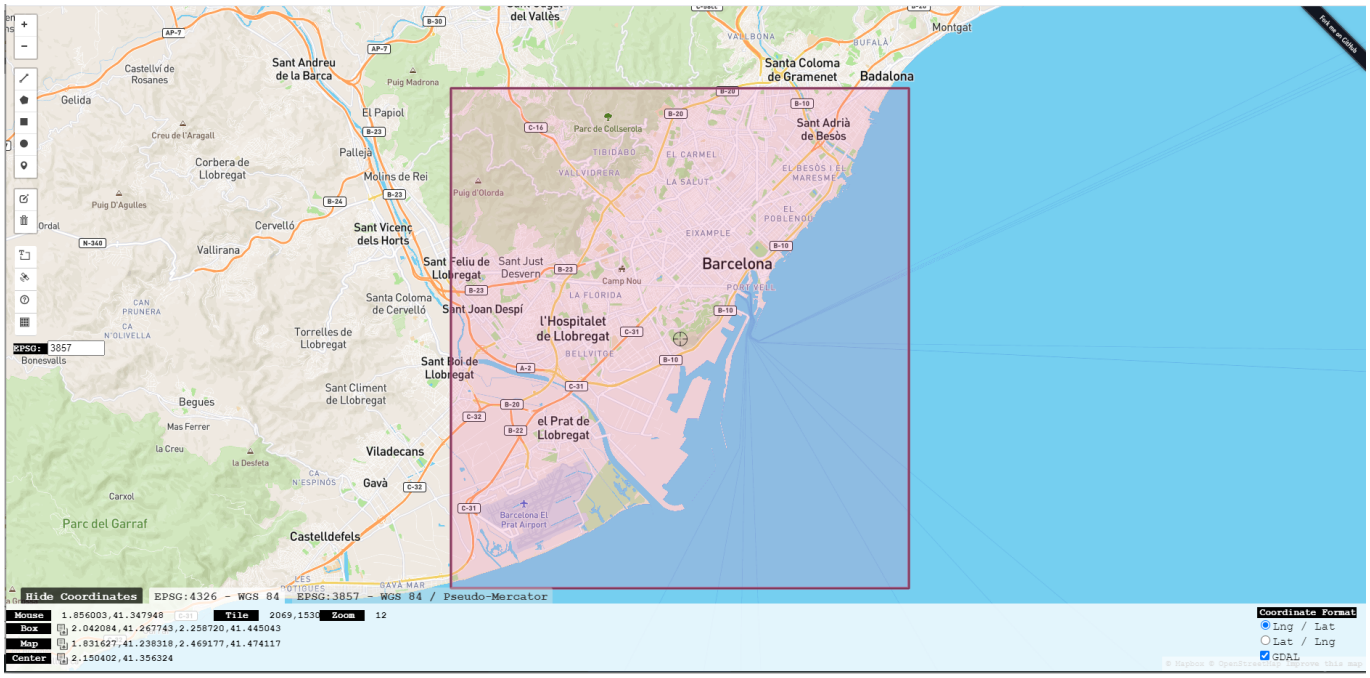

Figure 5.6: Defining the bounding box for obtaining OSM data from Barcelona.

```
CREATE EXTENSION mobilitydb CASCADE;
CREATE EXTENSION pgRouting;
```

Then, we download the OSM data from Barcelona using the Overpass API by writing the following in a terminal:

```
CITY="barcelona"
BBOX="2.042084,41.267743,2.258720,41.445043"
wget --progress=dot:mega -O "$CITY.osm" "http://www.overpass-api.de/api/xapi?*[bbox=${BBOX ↩
    }][@meta]"
```

We can optionally reduce the size of the OSM file as follows

```
sed -r "s/version=\"[0-9]+\" timestamp=\"[^\"]+\" changeset=\"[0-9]+\" uid=\"[0-9]+\" user ↩
    =\"[^\"]+\"//g" barcelona.osm -i.org
```

Finally, we load the map and convert it into a routable format suitable for pgRouting as follows.

```
osm2pgrouting -f barcelona.osm --dbname barcelona -c mapconfig_brussels.xml
```

5.8 Tuning the Generator Parameters

Multiple parameters can be used to tune the generator according to your needs. We describe next these parameters.

A first set of primary parameters determine the global behaviour of the generator. These parameters can also be set by a corresponding optional argument when calling the function `berlinmod_generate`.

- `P_SCALE_FACTOR: float`: Main parameter that determines the size of the data generated. Default value: 0.005. Corresponding optional argument: `scaleFactor`. By default, the scale factor determine the number of vehicles and the number of days they are observed as follows:

```
noVehicles int = round((2000 * sqrt(P_SCALE_FACTOR))::numeric, 0)::int;
noDays int = round((sqrt(P_SCALE_FACTOR) * 28)::numeric, 0)::int;
```

For example, for a scale factor of 1.0, the number of vehicles and the number of days will be, respectively, 2000 and 28. Alternatively, you can manually set the number of vehicles or the number of days using the optional arguments `noVehicles` and `noDays`, which are both integers.

- `P_START_DAY: date`: The day the observation starts. Default value: Monday 2020-01-06. Corresponding optional argument: `startDay`.
- `P_PATH_MODE: text`: Method for selecting a path between source and target nodes. Possible values are `'Fastest Path'` (default) and `'Shortest Path'`. Corresponding optional argument: `pathMode`.
- `P_NODE_CHOICE: text`: Method for selecting home and work nodes. Possible values are `'Network Based'` for chosing the nodes with a uniform distribution among all nodes (default) and `'Region Based'` to use the population and number of enterprises statistics in the `Regions` tables. Corresponding optional argument: `nodeChoice`.
- `P_DISTURB_DATA: boolean`: Determine whether imprecision is added to the data generated. Possible values are false (no imprecision, default) and true (disturbed data). Corresponding optional argument: `disturbData`.
- `P_MESSAGES: text`: Quantity of messages shown describing the generation process. Possible values are `'minimal'`, `'mediummmm'`, `'verbose'`, and `'debug'`. Corresponding optional argument: `messages`.
- `P_TRIP_GENERATION: text`: Determine the language used to generate the trips. Possible values are `'C'` (default) and `'SQL'`. Corresponding optional argument: `tripGeneration`.

For example, possible calls of the `berlinmod_generate` function setting values for the parameters are as follows.

```
-- Use all default values
SELECT berlinmod_generate();
-- Set the scale factor and use all other default values
SELECT berlinmod_generate(scaleFactor := 2.0);
-- Set the number of vehicles and number of days
SELECT berlinmod_generate(noVehicles := 10, noDays := 10);
```

Another set of parameters determining the global behaviour of the generator are given next.

- `P_RANDOM_SEED: float`: Seed for the random generator used to ensure deterministic results. Default value: 0.5.
- `P_NEIGHBOURHOOD_RADIUS: float`: Radius in meters defining a node neigbourhood. Default value: 3000.0.
- `P_SAMPLE_SIZE: int`: Size for sample relations. Default value: 100.
- `P_VEHICLE_TYPES: text[]`: Set of vehicle types. Default value: `{"passenger", "bus", "truck"}`.
- `P_VEHICLE_MODELS: text[]`: Set of vehicle models. Default value:

```
{"Mercedes-Benz", "Volkswagen", "Maybach", "Porsche", "Opel", "BMW", "Audi", "Acabion",
"Borgward", "Wartburg", "Sachsenring", "Multicar"}
```

- `P_PGROUTING_BATCH_SIZE: int`: Number of paths sent in a batch to pgRouting. Default value: 1e5 .

Another set of paramaters determine how the trips are created out of the paths.

- `P_EPSILON_SPEED: float`: Minimum speed in Km/h that is considered as a stop and thus only an accelaration event can be applied. Default value: 1.0.
- `P_EPSILON: float`: Minimum distance in the units of the coordinate system that is considered as zero. Default value: 0.0001.

- `P_EVENT_C: float`: The probability of a stop or a deceleration event is proportional to `P_EVENT_C / maxspeed`. Default value: 1.0
- `P_EVENT_P: float`: The probability for an event to be a stop. The complement 1.0 - `P_EVENT_P` is the probability for an event to be a deceleration. Default value: 0.1
- `P_EVENT_LENGTH: float`: Sampling distance in meters at which an acceleration, deceleration, or stop event may be generated. Default value: 5.0.
- `P_EVENT_ACC: float`: Constant speed in Km/h that is added to the current speed in an acceleration event. Default value: 12.0.
- `P_DEST_STOPPROB: float`: Probabilities for forced stops at crossings depending on the road type. It is defined by a transition matrix where lines and columns are ordered by side road (S), main road (M), freeway (F). The OSM highway types must be mapped to one of these categories in the function `berlinmod_roadCategory`. Default value:

```
{{0.33, 0.66, 1.00}, {0.33, 0.50, 0.66}, {0.10, 0.33, 0.05}}
```

- `P_DEST_EXPMU: float`: Mean waiting time in seconds using an exponential distribution. Increasing/decreasing this parameter allows us to slow down or speed up the trips. Could be think of as a measure of network congestion. Given a specific path, fine-tuning this parameter enable us to obtain an average travel time for this path that is the same as the expected travel time computed by a routing service such as, e.g., Google Maps. Default value: 1.0.
- `P_GPS_TOTALMAXERR: float` and `P_GPS_STEPMAXERR: float`: Parameters for simulating measuring errors. They are only required when the parameter `P_DISTURB_DATA` is true. They are, respectively, the maximum total deviation from the real position and maximum deviation per step, both in meters. Default values: 100.0 and 1.0.

5.9 Changing the Simulation Scenario

In this workshop, we have used until now the BerlinMOD scenario, which models the trajectories of persons going from home to work in the morning and returning back from work to home in the evening during the week days, with one possible leisure trip during the weekday nights and two possible leisure trips in the morning and in the afternoon of the weekend days. In this section, we devise another scenario for the data generator. This scenario corresponds to a home appliance shop that has several warehouses located in various places of the city. From each warehouse, the deliveries of appliances to customers are done by vehicles belonging to the warehouse. Although this scenario is different than BerlinMOD, many things can be reused and adapted. For example, home nodes can be replaced by warehouse locations, leisure destinations can be replaced by customer locations, and in this way many functions of the BerlinMOD SQL code will work directly. This is a direct benefit of having the simulation code written in SQL, so it will be easy to adapt to other scenarios. We describe next the needed changes.

Each day of the week excepted Sundays, deliveries of appliances from the warehouses to the customers are organized as follows. Each warehouse has several vehicles that make the deliveries. To each vehicle is assigned a list of customers that must be delivered during a day. A trip for a vehicle starts and ends at the warehouse and make the deliveries to the customers in the order of the list. Notice that in a real-world situation, the scheduling of the deliveries to clients by the vehicles requires to take into account the availability of the customers in a time slot of a day and the time needed to make the delivery of the previous customers in the list.

We describe next the main steps in the generation of the deliveries scenario.

We start by generating the `Warehouse` table. Each warehouse is located at a random node of the network.

```
DROP TABLE IF EXISTS Warehouse;
CREATE TABLE Warehouse(warehouseId int, nodeId bigint, geom geometry(Point));

FOR i IN 1..noWarehouses LOOP
  INSERT INTO Warehouse(warehouseId, nodeId, geom)
  SELECT i, id, geom
  FROM Nodes N
  ORDER BY id LIMIT 1 OFFSET random_int(1, noNodes);
END LOOP;
```

We create a relation `Vehicle` with all vehicles and the associated warehouse. Warehouses are associated to vehicles in a round-robin way.

```
DROP TABLE IF EXISTS Vehicle;
CREATE TABLE Vehicle(vehicleId int, warehouseId int, noNeighbours int);

INSERT INTO Vehicle(vehicleId, warehouseId)
SELECT id, 1 + ((id - 1) % noWarehouses)
FROM generate_series(1, noVehicles) id;
```

We then create a relation `Neighbourhood` containing for each vehicle the nodes with a distance less than the parameter `P_NEIGHBOURHOOD_RADIUS` to its warehouse node.

```
DROP TABLE IF EXISTS Neighbourhood;
CREATE TABLE Neighbourhood AS
SELECT ROW_NUMBER() OVER () AS id, V.vehicleId, N2.id AS Node
FROM Vehicle V, Nodes N1, Nodes N2
WHERE V.warehouseId = N1.id AND ST_DWithin(N1.Geom, N2.geom, P_NEIGHBOURHOOD_RADIUS);

CREATE UNIQUE INDEX Neighbourhood_id_idx ON Neighbourhood USING BTREE(id);
CREATE INDEX Neighbourhood_vehicleId_idx ON Neighbourhood USING BTREE(VehicleId);

UPDATE Vehicle V SET
  noNeighbours = (SELECT COUNT(*) FROM Neighbourhood N WHERE N.vehicleId = V.vehicleId);
```

We create next the `DeliveryTrip` and `Destinations` tables that contain, respectively, the list of source and destination nodes composing the delivery trip of a vehicle for a day, and the list of source and destination nodes for all vehicles.

```
DROP TABLE IF EXISTS DeliveryTrip;
CREATE TABLE DeliveryTrip(vehicle int, day date, seq int, source bigint, target bigint,
  PRIMARY KEY (vehicle, day, seq));
DROP TABLE IF EXISTS Destinations;
CREATE TABLE Destinations(id serial, source bigint, target bigint);
-- Loop for every vehicle
FOR i IN 1..noVehicles LOOP
  -- Get the warehouse node and the number of neighbour nodes
  SELECT W.node, V.noNeighbours INTO warehouseNode, noNeigh
  FROM Vehicle V, Warehouse W WHERE V.id = i AND V.warehouse = W.id;
  day = startDay;
  -- Loop for every generation day
  FOR j IN 1..noDays LOOP
    -- Generate delivery trips excepted on Sunday
    IF date_part('dow', day) <> 0 THEN
      -- Select a number of destinations between 3 and 7
      SELECT random_int(3, 7) INTO noDest;
      sourceNode = warehouseNode;
      FOR k IN 1..noDest + 1 LOOP
        IF k <= noDest THEN
          targetNode = berlinmod_selectDestNode(i, noNeigh, noNodes);
        ELSE
          targetNode = warehouseNode;
        END IF;
        IF targetNode IS NULL THEN
          RAISE EXCEPTION '    Destination node cannot be NULL';
        END IF;
        -- Keep the start and end nodes of each subtrip
        INSERT INTO DeliveryTrip VALUES (i, day, k, sourceNode, targetNode);
        INSERT INTO Destinations(source, target) VALUES (sourceNode, targetNode);
        sourceNode = targetNode;
      END LOOP;
    END IF;
    day = day + 1 * interval '1 day';
```

```
  END LOOP;
END LOOP;
```

For every vehicle and every day which is not Sunday we proceed as follows. We randomly chose a number between 3 and 7 destinations and call the function `berlinmod_selectDestNode` we have seen in previous sections for determining these destinations. This function choses either one node in the neighbourhood of the warehouse of the vehicle with 80% probability or a node from the complete graph with 20% probability. Then, the sequence of source and destination couples starting in the warehouse, visiting sequentially the clients to deliver and returning to the warehouse are added to the tables `DeliveryTrip` and `Destinations`.

Next, we compute the paths between all source and target nodes that are in the `Destinations` table. Such paths are generated by pgRouting and stored in the `Paths` table.

```
DROP TABLE IF EXISTS Paths;
CREATE TABLE Paths(seq int, path_seq int, start_vid bigint, end_vid bigint,
  node bigint, edge bigint, cost float, agg_cost float,
  -- These attributes are filled in the subsequent update
  geom geometry, speed float, category int);

-- Select query sent to pgRouting
IF pathMode = 'Fastest Path' THEN
  query1_pgr = 'SELECT id, source, target, cost_s AS cost, reverse_cost_s as reverse_cost  ↩
      FROM edges';
ELSE
  query1_pgr = 'SELECT id, source, target, length_m AS cost, length_m * sign(reverse_cost_s ↩
      ) as reverse_cost FROM edges';
END IF;
-- Get the total number of paths and number of calls to pgRouting
SELECT COUNT(*) INTO noPaths FROM (SELECT DISTINCT source, target FROM Destinations) AS T;
noCalls = ceiling(noPaths / P_PGROUTING_BATCH_SIZE::float);
FOR i IN 1..noCalls LOOP
  query2_pgr = format('SELECT DISTINCT source, target FROM Destinations ORDER BY source,  ↩
      target LIMIT %s OFFSET %s',
    P_PGROUTING_BATCH_SIZE, (i - 1) * P_PGROUTING_BATCH_SIZE);
  INSERT INTO Paths(seq, path_seq, start_vid, end_vid, node, edge, cost, agg_cost)
  SELECT * FROM pgr_dijkstra(query1_pgr, query2_pgr, true);
END LOOP;

UPDATE Paths SET geom =
    -- adjusting directionality
    CASE
      WHEN node = E.source THEN E.geom
      ELSE ST_Reverse(E.geom)
    END,
    speed = maxspeed_forward,
    category = berlinmod_roadCategory(tag_id)
  FROM Edges E WHERE E.id = edge;

CREATE INDEX Paths_start_vid_end_vid_idx ON Paths USING BTREE(start_vid, end_vid);
```

After creating the `Paths` table, we set the query to be sent to pgRouting depending on whether we have want to compute the fastest or the shortest paths between two nodes. The generator uses the parameter `P_PGROUTING_BATCH_SIZE` to determine the maximum number of paths we compute in a single call to pgRouting. This parameter is set to 10,000 by default. Indeed, there is limit in the number of paths that pgRouting can compute in a single call and this depends in the available memory of the computer. Therefore, we need to determine the number of calls to pgRouting and compute the paths by calling the function `pgr_dijkstra`. Finally, we need to adjust the directionality of the geometry of the edges depending on which direction a trip traverses the edges, and set the speed and the category of the edges.

The following procedure generates the trips for a number of vehicles and a number of days starting at a given day. The last argument correspond to the Boolean parameter `P_DISTURB_DATA` that determines whether simulated GPS errors are added to the trips.

```
DROP FUNCTION IF EXISTS deliveries_createTrips;
CREATE FUNCTION deliveries_createTrips(noVehicles int, noDays int, startDay Date,
  disturbData boolean)
RETURNS void LANGUAGE plpgsql STRICT AS $$
DECLARE
  -- Loops over the days for which we generate the data
  day date;
  -- 0 (Sunday) to 6 (Saturday)
  weekday int;
  -- Loop variables
  i int; j int;
BEGIN
  DROP TABLE IF EXISTS Trips;
  CREATE TABLE Trips(vehicle int, day date, seq int, source bigint,
    target bigint, trip tgeompoint,
    -- These columns are used for visualization purposes
    trajectory geometry, sourceGeom geometry,
    PRIMARY KEY (vehicle, day, seq));
  day = startDay;
  FOR i IN 1..noDays LOOP
    SELECT date_part('dow', day) into weekday;
    -- 6: saturday, 0: sunday
    IF weekday <> 0 THEN
      FOR j IN 1..noVehicles LOOP
        PERFORM deliveries_createDay(j, day, disturbData);
      END LOOP;
    END IF;
    day = day + 1 * interval '1 day';
  END LOOP;
  -- Add geometry attributes for visualizing the results
  UPDATE Trips SET sourceGeom = (SELECT geom FROM Nodes WHERE id = source);
  RETURN;
END; $$
```

As can be seen, this procedure simply loops for each day (excepted Sundays) and for each vehicle and calls the function `deliveries_createDay` which is given next.

```
DROP FUNCTION IF EXISTS deliveries_createDay;
CREATE FUNCTION deliveries_createDay(vehicId int, aDay date, disturbData boolean)
RETURNS void LANGUAGE plpgsql STRICT AS $$
DECLARE
  -- Current timestamp
  t timestamptz;
  -- Start time of a trip to a destination
  startTime timestamptz;
  -- Number of trips in a delivery (number of destinations + 1)
  noTrips int;
  -- Loop variable
  i int;
  -- Time delivering a customer
  deliveryTime interval;
  -- Warehouse identifier
  warehouseNode bigint;
  -- Source and target nodes of one subtrip of a delivery trip
  sourceNode bigint; targetNode bigint;
  -- Path betwen start and end nodes
  path step[];
  -- Trip obtained from a path
  trip tgeompoint;
BEGIN
  -- 0: sunday
```

```
  IF date_part('dow', aDay) <> 0 THEN
    -- Start delivery
    t = aDay + time '07:00:00' + createPauseN(120);
    -- Get the number of trips (number of destinations + 1)
    SELECT count(*) INTO noTrips
    FROM DeliveryTrip D
    WHERE D.vehicle = vehicId AND D.day = aDay;
    FOR i IN 1..noTrips LOOP
      -- Get the source and destination nodes of the trip
      SELECT source, target INTO sourceNode, targetNode
      FROM DeliveryTrip D
      WHERE D.vehicle = vehicId AND D.day = aDay AND D.seq = i;
      -- Get the path
      SELECT array_agg((geom, speed, category) ORDER BY path_seq) INTO path
      FROM Paths P
      WHERE start_vid = sourceNode AND end_vid = targetNode AND edge > 0;
      IF path IS NULL THEN
        RAISE EXCEPTION 'The path of a trip cannot be NULL';
      END IF;
      startTime = t;
      trip = create_trip(path, t, disturbData);
      IF trip IS NULL THEN
        RAISE EXCEPTION 'A trip cannot be NULL';
      END IF;
      INSERT INTO Trips VALUES (vehicId, aDay, i, sourceNode, targetNode,
        trip, trajectory(trip));
      t = endTimestamp(trip);
      -- Add a delivery time in [10, 60] min using a bounded Gaussian distribution
      deliveryTime = random_boundedgauss(10, 60) * interval '1 min';
      t = t + deliveryTime;
    END LOOP;
  END IF;
END;
$$ LANGUAGE plpgsql STRICT;
```

We first set the start time of a delivery trip by adding to 7 am a random non-zero duration of 120 minutes using a uniform distribution. Then, for every couple of source and destination nodes to be visited in the trip, we call the function `create_trip` that we have seen previously to generate the trip, wich is then inserted into the `Trips` table. Finally, we add a delivery time between 10 and 60 minutes using a bounded Gaussian distribution before starting the trip to the next customer or the return trip to the warehouse.

Figure 5.7 and Figure 5.8 show visualizations of the data generated for the deliveries scenario.

5.10 Creating a Graph from Input Data

In this workshop, we have used until now the network topology obtained by osm2pgrouting. However, in some circumstances it is necessary to build the network topology ourselves, for example, when the data comes from other sources than OSM, such as data from an official mapping agency. In this section we show how to build the network topology from input data. We import Brussels data from OSM into a PostgreSQL database using osm2pgsql. Then, we construct the network topology using SQL so that the resulting graph can be used with pgRouting. We show two approaches for doing this, depending on whether we want to keep the original roads of the input data or we want to merge roads when they have similar characteristics such as road type, direction, maximum speed, etc. At the end, we compare the two networks obtained with the one obtained by osm2pgrouting.

5.10.1 Creating the Graph

As we did at the beginning of this chapter, we load the OSM data from Brussels into PostgreSQL with the following command.

```
osm2pgsql --create --database brussels --host localhost brussels.osm
```

Figure 5.7: Visualization of the data generated for the deliveries scenario. The road network is shown with blue lines, the warehouses are shown with a red star, the routes taken by the deliveries are shown with black lines, and the location of the customers with black points.

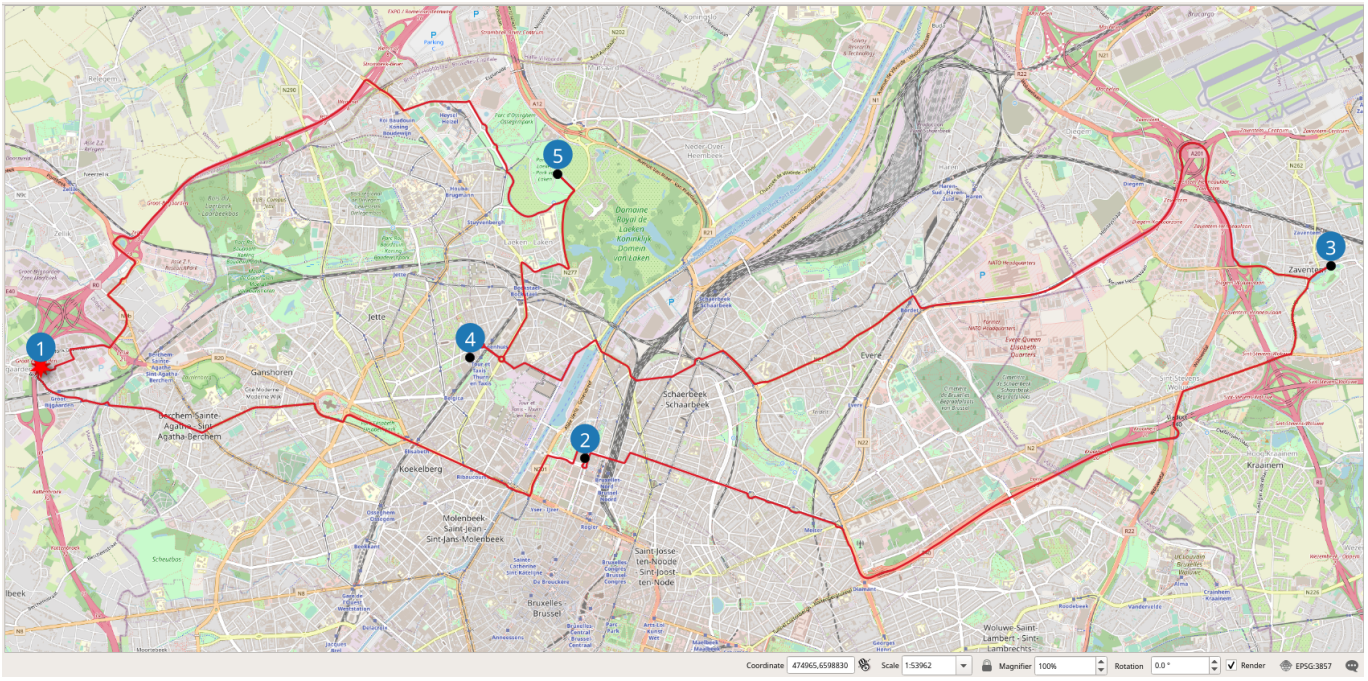

Figure 5.8: Visualization of the deliveries of one vehicle during one day. A delivery trip starts and ends at a warehouse and make the deliveries to several customers, four in this case.

The table `planet_osm_line` contains all linear features imported from OSM, in particular road data, but also many other features which are not relevant for our use case such as pedestrian paths, cycling ways, train ways, electric lines, etc. Therefore, we use the attribute `highway` to extract the roads from this table. We first create a table containing the road types we are interested in and associate to them a priority, a maximum speed, and a category as follows.

```
DROP TABLE IF EXISTS RoadTypes;
CREATE TABLE RoadTypes(id int PRIMARY KEY, type text, priority float, maxspeed float,
  category int);
INSERT INTO RoadTypes VALUES
(101, 'motorway', 1.0, 120, 1),
(102, 'motorway_link', 1.0, 120, 1),
(103, 'motorway_junction', 1.0, 120, 1),
(104, 'trunk', 1.05, 120, 1),
(105, 'trunk_link', 1.05, 120, 1),
(106, 'primary', 1.15, 90, 2),
(107, 'primary_link', 1.15, 90, 1),
(108, 'secondary', 1.5, 70, 2),
(109, 'secondary_link', 1.5, 70, 2),
(110, 'tertiary', 1.75, 50, 2),
(111, 'tertiary_link', 1.75, 50, 2),
(112, 'residential', 2.5, 30, 3),
(113, 'living_street', 3.0, 20, 3),
(114, 'unclassified', 3.0, 20, 3),
(115, 'service', 4.0, 20, 3),
(116, 'services', 4.0, 20, 3);
```

Then, we create a table that contains the roads corresponding to one of the above types as follows.

```
DROP TABLE IF EXISTS Roads;
CREATE TABLE Roads AS
SELECT osm_id, admin_level, bridge, cutting, highway, junction, name, oneway, operator,
  ref, route, surface, toll, tracktype, tunnel, width, way AS geom
FROM planet_osm_line
```

```
WHERE highway IN (SELECT type FROM RoadTypes);

CREATE INDEX Roads_geom_idx ON Roads USING GiST(geom);
```

We then create a table that contains all intersections between two roads as follows:

```
DROP TABLE IF EXISTS Intersections;
CREATE TABLE Intersections AS
WITH Temp1 AS (
  SELECT ST_Intersection(a.geom, b.geom) AS geom
  FROM Roads a, Roads b
  WHERE a.osm_id < b.osm_id AND ST_Intersects(a.geom, b.geom)
),
Temp2 AS (
  SELECT DISTINCT geom
  FROM Temp1
  WHERE geometrytype(geom) = 'POINT'
  UNION
  SELECT (ST_DumpPoints(geom)).geom
  FROM Temp1
  WHERE geometrytype(geom) = 'MULTIPOINT'
)
SELECT ROW_NUMBER() OVER () AS id, geom
FROM Temp2;

CREATE INDEX Intersections_geom_idx ON Intersections USING GIST(geom);
```

The temporary table `Temp1` computes all intersections between two different roads, while the temporary table `Temp2` selects all intersections of type point and splits the intersections of type multipoint into the component points with the function `ST_DumpPoints`. Finally, the last query adds a sequence identifier to the resulting intersections.

Our next task is to use the table `Intersections` we have just created to split the roads. This is done as follows.

```
DROP TABLE IF EXISTS Segments;
CREATE TABLE Segments AS
SELECT DISTINCT osm_id, (ST_Dump(ST_Split(R.geom, I.geom))).geom
FROM Roads R, Intersections I
WHERE ST_Intersects(R.Geom, I.geom);

CREATE INDEX Segments_geom_idx ON Segments USING GIST(geom);
```

The function `ST_Split` breaks the geometry of a road using an intersection and the function `ST_Dump` obtains the individual segments resulting from the splitting. However, as shown in the following query, there are duplicate segments with distinct `osm_id`.

```
SELECT S1.osm_id, S2.osm_id
FROM Segments S1, Segments S2
WHERE S1.osm_id < S2.osm_id AND st_intersects(S1.geom, S2.geom) AND
  ST_Equals(S1.geom, S2.geom);
-- 490493551  740404156
-- 490493551  740404157
```

We can remove those duplicates segments with the following query, which keeps arbitrarily the smaller `osm_id`.

```
DELETE FROM Segments S1
    USING Segments S2
WHERE S1.osm_id > S2.osm_id AND ST_Equals(S1.geom, S2.geom);
```

We can obtain some characteristics of the segments with the following queries.

```
SELECT DISTINCT geometrytype(geom) FROM Segments;
-- "LINESTRING"
```

```
SELECT min(ST_NPoints(geom)), max(ST_NPoints(geom)) FROM Segments;
-- 2  283
```

Now we are ready to obtain a first set of nodes for our graph.

```
DROP TABLE IF EXISTS TempNodes;
CREATE TABLE TempNodes AS
WITH Temp(geom) AS (
  SELECT ST_StartPoint(geom) FROM Segments UNION
  SELECT ST_EndPoint(geom) FROM Segments
)
SELECT ROW_NUMBER() OVER () AS id, geom
FROM Temp;

CREATE INDEX TempNodes_geom_idx ON TempNodes USING GIST(geom);
```

The above query select as nodes the start and the end points of the segments and assigns to each of them a sequence identifier. We construct next the set of edges of our graph as follows.

```
DROP TABLE IF EXISTS Edges;
CREATE TABLE Edges(id bigint, osm_id bigint, tag_id int, length_m float, source bigint,
  target bigint, cost_s float, reverse_cost_s float, one_way int, maxspeed float,
  priority float, geom geometry);
INSERT INTO Edges(id, osm_id, source, target, geom, length_m)
SELECT ROW_NUMBER() OVER () AS id, S.osm_id, N1.id AS source, N2.id AS target, S.geom,
  ST_Length(S.geom) AS length_m
FROM Segments S, TempNodes N1, TempNodes N2
WHERE ST_Intersects(ST_StartPoint(S.geom), N1.geom) AND
  ST_Intersects(ST_EndPoint(S.geom), N2.geom);

CREATE UNIQUE INDEX Edges_id_idx ON Edges USING BTREE(id);
CREATE INDEX Edges_geom_index ON Edges USING GiST(geom);
```

The above query connects the segments obtained previously to the source and target nodes. We can verify that all edges were connected correctly to their source and target nodes using the following query.

```
SELECT count(*) FROM Edges WHERE source IS NULL OR target IS NULL;
-- 0
```

Now we can fill the other attributes of the edges. We start first with the attributes `tag_id`, `priority`, and `maxspeed`, which are obtained from the table `RoadTypes` using the attribute `highway`.

```
UPDATE Edges E
SET tag_id = T.id, priority = T.priority, maxspeed = T.maxSpeed
FROM Roads R, RoadTypes T
WHERE E.osm_id = R.osm_id AND R.highway = T.type;
```

We continue with the attribute `one_way` according to the semantics stated in the OSM documentation.

```
UPDATE Edges E
SET one_way = CASE
  WHEN R.oneway = 'yes' OR R.oneway = 'true' OR R.oneway = '1' THEN 1 -- Yes
  WHEN R.oneway = 'no' OR R.oneway = 'false' OR R.oneway = '0' THEN 2 -- No
  WHEN R.oneway = 'reversible' THEN 3 -- Reversible
  WHEN R.oneway = '-1' OR R.oneway = 'reversed' THEN -1 -- Reversed
  WHEN R.oneway IS NULL THEN 0 -- Unknown
  END
FROM Roads R
WHERE E.osm_id = R.osm_id;
```

We compute the implied one way restriction based on OSM documentation as follows.

```
UPDATE Edges E
SET one_way = 1
FROM Roads R
WHERE E.osm_id = R.osm_id AND R.oneway IS NULL AND
  (R.junction = 'roundabout' OR R.highway = 'motorway');
```

Finally, we compute the cost and reverse cost in seconds according to the length and the maximum speed of the edge.

```
UPDATE Edges E SET
  cost_s = CASE
    WHEN one_way = -1 THEN - length_m / (maxspeed / 3.6)
    ELSE length_m / (maxspeed / 3.6)
    END,
  reverse_cost_s = CASE
    WHEN one_way = 1 THEN - length_m / (maxspeed / 3.6)
    ELSE length_m / (maxspeed / 3.6)
    END;
```

Our last task is to compute the strongly connected components of the graph. This is necessary to ensure that there is a path between every couple of arbritrary nodes in the graph.

```
DROP TABLE IF EXISTS Nodes;
CREATE TABLE Nodes AS
WITH Components AS (
  SELECT * FROM pgr_strongComponents(
    'SELECT id, source, target, length_m AS cost, '
    'length_m * sign(reverse_cost_s) AS reverse_cost FROM Edges')
),
LargestComponent AS (
  SELECT component, count(*) FROM Components
  GROUP BY component ORDER BY count(*) DESC LIMIT 1
),
Connected AS (
  SELECT geom
  FROM TempNodes N, LargestComponent L, Components C
  WHERE N.id = C.node AND C.component = L.component
)
SELECT ROW_NUMBER() OVER () AS id, geom
FROM Connected;

CREATE UNIQUE INDEX Nodes_id_idx ON Nodes USING BTREE(id);
CREATE INDEX Nodes_geom_idx ON Nodes USING GiST(geom);
```

The temporary table `Components` is obtained by calling the function `pgr_strongComponents` from pgRouting, the temporary table `LargestComponent` selects the largest component from the previous table, and the temporary table `Connected` selects all nodes that belong to the largest component. Finally, the last query assigns a sequence identifier to all nodes.

Now that we computed the nodes of the graph, we need to link the edges with the identifiers of these nodes. This is done as follows.

```
UPDATE Edges SET source = NULL, target = NULL;

UPDATE Edges E SET
  source = N1.id, target = N2.id
FROM Nodes N1, Nodes N2
WHERE ST_Intersects(E.geom, N1.geom) AND ST_StartPoint(E.geom) = N1.geom AND
  ST_Intersects(E.geom, N2.geom) AND ST_EndPoint(E.geom) = N2.geom;
```

We first set the identifiers of the source and target nodes to NULL before connecting them to the identifiers of the node. Finally, we delete the edges whose source or target node has been removed.

```
DELETE FROM Edges WHERE source IS NULL OR target IS NULL;
-- DELETE 1080
```

In order to compare the graph we have just obtained with the one obtained by osm2pgrouting we can issue the following queries.

```
SELECT count(*) FROM Ways;
-- 83017
SELECT count(*) FROM Edges;
-- 81073
SELECT count(*) FROM Ways_vertices_pgr;
-- 66832
SELECT count(*) FROM Nodes;
-- 45494
```

As can be seen, we have reduced the size of the graph. This can also be shown in Figure 5.9, where the nodes we have obtained are shown in blue and the ones obtained by osm2pgrouting are shown in red. It can be seen that osm2pgrouting adds many more nodes to the graph, in particular, at the intersection of a road and a pedestrian crossing. Our method only adds nodes when there is an intersection between two roads. We will show in the next section how this network can still be optimized by removing unnecessary nodes and merging the corresponding edges.

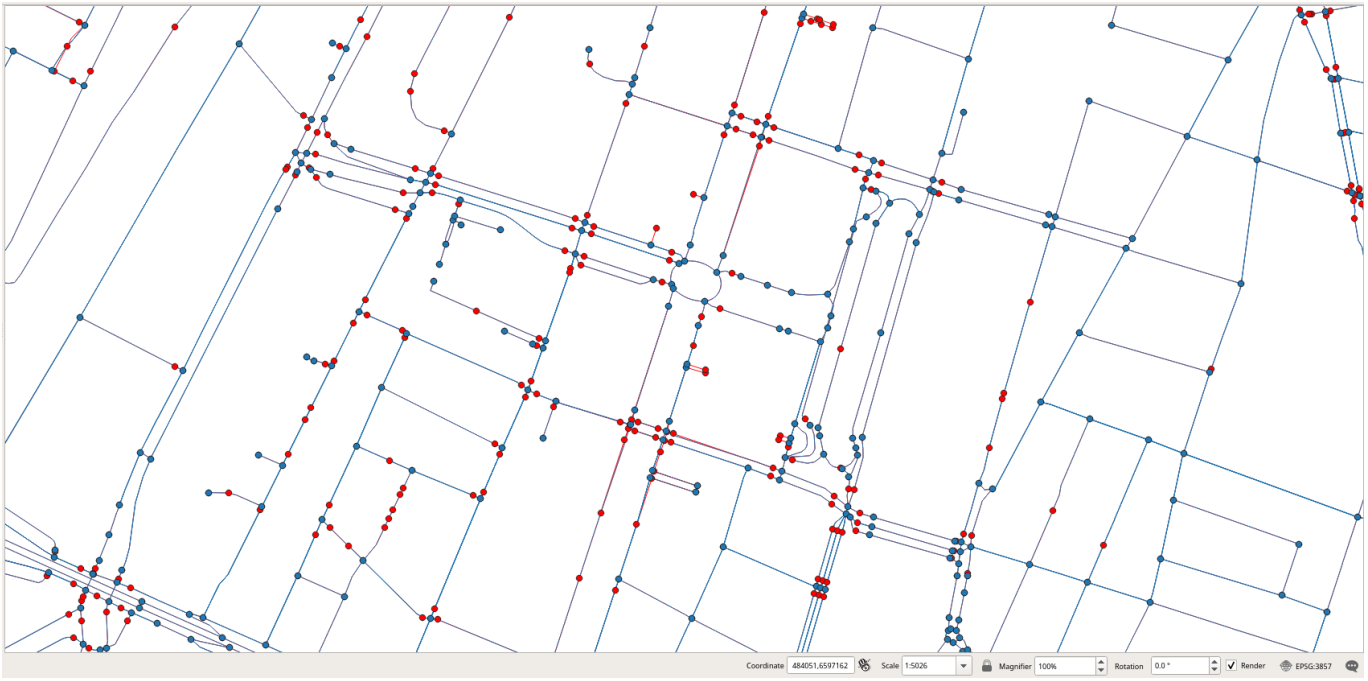

Figure 5.9: Comparison of the nodes obtained (in blue) with those obtained by osm2pgrouting (in red).

5.10.2 Linear Contraction of the Graph

We show next a possible approach to contract the graph. This approach corresponds to linear contraction provided by pgRouting although we do it differently by taking into account the type, the direction, and the geometry of the roads. For this, we get the initial roads to merge as we did previously but now we put them in a table `TempRoads`.

```
DROP TABLE IF EXISTS TempRoads;
CREATE TABLE TempRoads AS
SELECT osm_id, admin_level, bridge, cutting, highway, junction, name, oneway, operator,
  ref, route, surface, toll, tracktype, tunnel, width, way AS geom
FROM planet_osm_line
WHERE highway IN (SELECT type FROM RoadTypes);
```

```
-- SELECT 37045
CREATE INDEX TempRoads_geom_idx ON TempRoads USING GiST(geom);
```

Then, we use the following procedure to merge the roads.

```
CREATE OR REPLACE FUNCTION mergeRoads()
RETURNS void LANGUAGE PLPGSQL AS $$
DECLARE
  i integer = 1;
  cnt integer;
BEGIN
  -- Create tables
  DROP TABLE IF EXISTS MergedRoads;
  CREATE TABLE MergedRoads AS
  SELECT *, '{}'::bigint[] AS path
  FROM TempRoads;
  CREATE INDEX MergedRoads_geom_idx ON MergedRoads USING GIST(geom);
  DROP TABLE IF EXISTS Merge;
  CREATE TABLE Merge(osm_id1 bigint, osm_id2 bigint, geom geometry);
  DROP TABLE IF EXISTS DeletedRoads;
  CREATE TABLE DeletedRoads(osm_id bigint);
  -- Iterate until no geometry can be extended
  LOOP
    RAISE INFO 'Iteration %', i;
    i = i + 1;
    -- Compute the union of two roads
    DELETE FROM Merge;
    INSERT INTO Merge
    SELECT R1.osm_id AS osm_id1, R2.osm_id AS osm_id2,
      ST_LineMerge(ST_Union(R1.geom, R2.geom)) AS geom
    FROM MergedRoads R1, TempRoads R2
    WHERE R1.osm_id <> R2.osm_id AND R1.highway = R2.highway AND
      R1.oneway = R2.oneway AND ST_Intersects(R1.geom, R2.geom) AND
      ST_EndPoint(R1.geom) =  ST_StartPoint(R2.geom)
      AND NOT EXISTS (
        SELECT * FROM TempRoads R3
        WHERE osm_id NOT IN (SELECT osm_id FROM DeletedRoads) AND
          R3.osm_id <> R1.osm_id AND R3.osm_id <> R2.osm_id AND
          ST_Intersects(R3.geom, ST_StartPoint(R2.geom)))
      AND geometryType(ST_LineMerge(ST_Union(R1.geom, R2.geom))) = 'LINESTRING'
      AND NOT St_Equals(ST_LineMerge(ST_Union(R1.geom, R2.geom)), R1.geom);
    -- Exit if there is no more roads to extend
    SELECT count(*) INTO cnt FROM Merge;
    RAISE INFO 'Extended % roads', cnt;
    EXIT WHEN cnt = 0;
    -- Extend the geometries
    UPDATE MergedRoads R SET
      geom = M.geom,
      path = R.path || osm_id2
    FROM Merge M
    WHERE R.osm_id = M.osm_id1;
    -- Keep track of redundant roads
    INSERT INTO DeletedRoads
    SELECT osm_id2 FROM Merge
    WHERE osm_id2 NOT IN (SELECT osm_id FROM DeletedRoads);
  END LOOP;
  -- Delete redundant roads
  DELETE FROM MergedRoads R USING DeletedRoads M
  WHERE R.osm_id = M.osm_id;
  -- Drop tables
  DROP TABLE Merge;
  DROP TABLE DeletedRoads;
```

```
  RETURN;
END; $$
```

The procedure starts by creating a table `MergedRoads` obtained by adding a column `path` to the table `TempRoads` created before. This column keeps track of the identifiers of the roads that are merged with the current one and is initialized to an empty array. It also creates two tables `Merge` and `DeletedRoads` that will contain, respectively, the result of merging two roads, and the identifiers of the roads that will be deleted at the end of the process. The procedure then iterates while there is at least one road that can be extended with the geometry of another one to which it connects to. More precisely, a road can be extended with the geometry of another one if they are of the same type and the same direction (as indicated by the attributes `highway` and `one_way`), the end point of the road is the start point of the other road, and this common point is not a crossing, that is, there is no other road that starts and this common point. Notice that we only merge roads if their resulting geometry is a linestring and we avoid infinite loops by verifying that the merge of the two roads is different from the original geometry. After that, we update the roads with the new geometries and add the identifier of the road used to extend the geometry into the `path` attribute and the `DeletedRoads` table. After exiting the loop, the procedure finishes by removing unnecessary roads.

The above procedure iterates 20 times for the largest segment that can be assembled, which is located in the ring-road around Brussels between two exits. It takes 15 minutes to execute in my laptop.

```
INFO:  Iteration 1
INFO:  Extended 3431 roads
INFO:  Iteration 2
INFO:  Extended 1851 roads
INFO:  Iteration 3
INFO:  Extended 882 roads
INFO:  Iteration 4
INFO:  Extended 505 roads
[...]
INFO:  Iteration 17
INFO:  Extended 3 roads
INFO:  Iteration 18
INFO:  Extended 2 roads
INFO:  Iteration 19
INFO:  Extended 1 roads
INFO:  Iteration 20
INFO:  Extended 0 roads
```

After we apply the above procedure to merge the roads, we are ready to create a new set of roads from which we can construct the graph.

```
CREATE TABLE Roads AS
SELECT osm_id || path AS osm_id,
  admin_level, bridge, cutting, highway, junction, name, oneway,
  operator, ref, route, surface, toll, tracktype, tunnel, width, geom
FROM MergedRoads;

CREATE INDEX Roads_geom_idx ON Roads USING GiST(geom);
```

Notice that now the attribute `osm_id` is an array of OSM identifiers (which are big integers), whereas in the previous section it was a single big integer.

We then proceed as we did in Section 5.10.1 to compute the set of nodes and the set of edges, which we will store now for comparison purposes into tables `Nodes1` and `Edges1`. We can issue the following queries to compare the two graphs we have obtained and the one obtained by osm2pgrouting .

```
SELECT count(*) FROM Ways;
-- 83017
SELECT count(*) FROM Edges;
-- 81073
SELECT count(*) FROM Edges1;
-- 77986
SELECT count(*) FROM Ways_vertices_pgr;
```

```
-- 66832
SELECT count(*) FROM Nodes;
-- 45494
SELECT count(*) FROM Nodes1;
-- 42156
```

Figure 5.10 shows the nodes for the three graphs, those obtained after contracting the graph are shown in black, those before contraction are shown in blue, and those obtained by osm2pgrouting are shown in red. The figure shows in particular how several segments of the ring-road around Brussels are merged together since the have the same road type, direction, and maximum speed, The figure also shows in read a road that was removed since it does not belong to the strongly connected components of the graph.

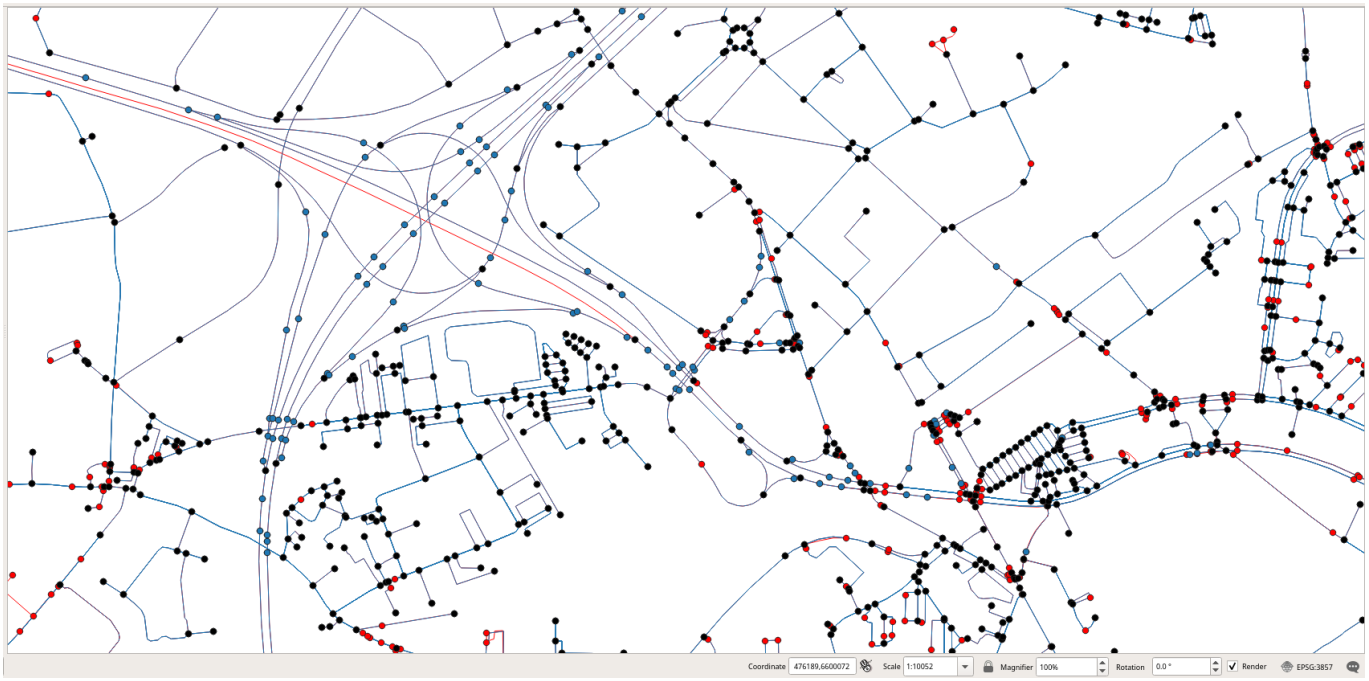

Figure 5.10: Comparison of the nodes obtained by contracting the graph (in black), before contraction (in blue), and those obtained by osm2pgrouting (in red).