BerlinMOD Benchmark on MobilityDB

COLLABORATORS

	TITLE : BerlinMOD Benchmark on MobilityDB	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Esteban Zimányi	August 3, 2020	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

List of Figures

Abstract

MobilityDB is an extension to the PostgreSQL object-relational database system and its spatial extension PostGIS. It allows temporal and spatio-temporal objects to be stored in the database, that is, objects whose attribute values and/or location evolves in time. This document shows an implementation of the BerlinMOD benchmark 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

It starts with a tutorial introducing MobilityDB based on BerlinMOD data, continues by explaining how to generate realistic trajectory datasets of arbitrary size, and concludes by explaining how to run the BerlinMOD benchmark on MobilityDB.

MobilityDB is open source and its code is available on Github. MobilityDB is developed by the Computer & Decision Engineering Department of the Université Libre de Bruxelles (ULB) under the direction of Prof. Esteban Zimányi. ULB is an OGC Associate Member.

Chapter 1

MobilityDB Tutorial

To illustrate the capabilities of MobilityDB, we give an example use case that loads, explores, and query mobility data. The data used is based on the BerlinMOD benchmark for moving object databases and is available as a ZIP file.

1.1 Installation

For this tutorial we will use a Docker image containing MobilityDB and all its dependencies (including PostgreSQL and PostGIS). The container has a default database called `mobilitydb` with the MobilityDB extension installed where `user = pw = docker`. This presupposes that you have installed Docker into your computer. In that case, you can run the following command.

```
docker pull codewit/mobilitydb
docker volume create mobilitydb_data
docker run --name "mobilitydb" -d -p 25432:5432 -v mobilitydb_data:/var/lib/postgresql
  codewit/mobilitydb
```

In the above commands

- `docker pull` downloads the Docker image of `mobilitydb`. If the image has been downloaded before, this checks whether a more recent image has been published in the docker repository, and downloads it. It is better to call this command every time, to ensures that you have the latest most up-to-date version of this image.
- `docker volume create mobilitydb_data` creates a volume container on the host, that we will use to persist the PostgreSQL database files outside of the MobilityDB container. You need to run this command only once, during the first use of the image
- `docker run --name=mobilitydb` tells Docker our new container will be named `mobilitydb`.
- `-d` runs the container in the background (detached mode).
- `-p 25432:5432` maps TCP port 5432 in the container to port 25432 on the Docker host (to prevent potential conflicts with any local database instance you may have). This is required because the PostgreSQL database server in the container listens for connections on port 5432 by default.
- `-v mobilitydb_data:/var/lib/postgresql` tells the container filesystem to mount the `mobilitydb_data` volume that we have just created to the path /var/lib/postgresql. This means that any database objects that the container saves or creates (by default in `/var/lib/postgresql`) will instead be persisted in the `mobilitydb_data` directory, which is stored in the host. This options ensures that your data will not be lost when the container is removed.
- `codewit/mobilitydb` tells Docker to pull the docker image with that name from Docker Hub.

Now we can launch any PostgreSQL administrative front-end to start using MobilityDB. Two traditional ones are the command-line tool `psql` and the graphical tool `pgAdmin`. We can launch `psql` as follows.

```
docker exec -t -i mobilitydb psql -h localhost -p 5432 -d mobilitydb -U docker
```

In the above command

- `docker exec -t -i mobilitydb psql` tells Docker to allocate a pseudo-TTY, to keep STDIN open, and to execute in the container `mobilitydb` the command `psql`.
- `-h localhost -p 5432 -d mobilitydb -U docker` tells `psql`, respectively, the database server host, the server port, the database name, and the user name.

Note that you will be prompted to provide the password, which is also `docker`.

In order to launch `pgAdmin`, there are two options to create a connection. The first option is to set the host to the localhost (127.0.0.1), and the port to the mapped one on the host, as per the `docker run` command. In this example the port is 25432. Now we can launch `pgAdmin` and establish a new connection to the docker container. This is done as shown in Figure 1.1.

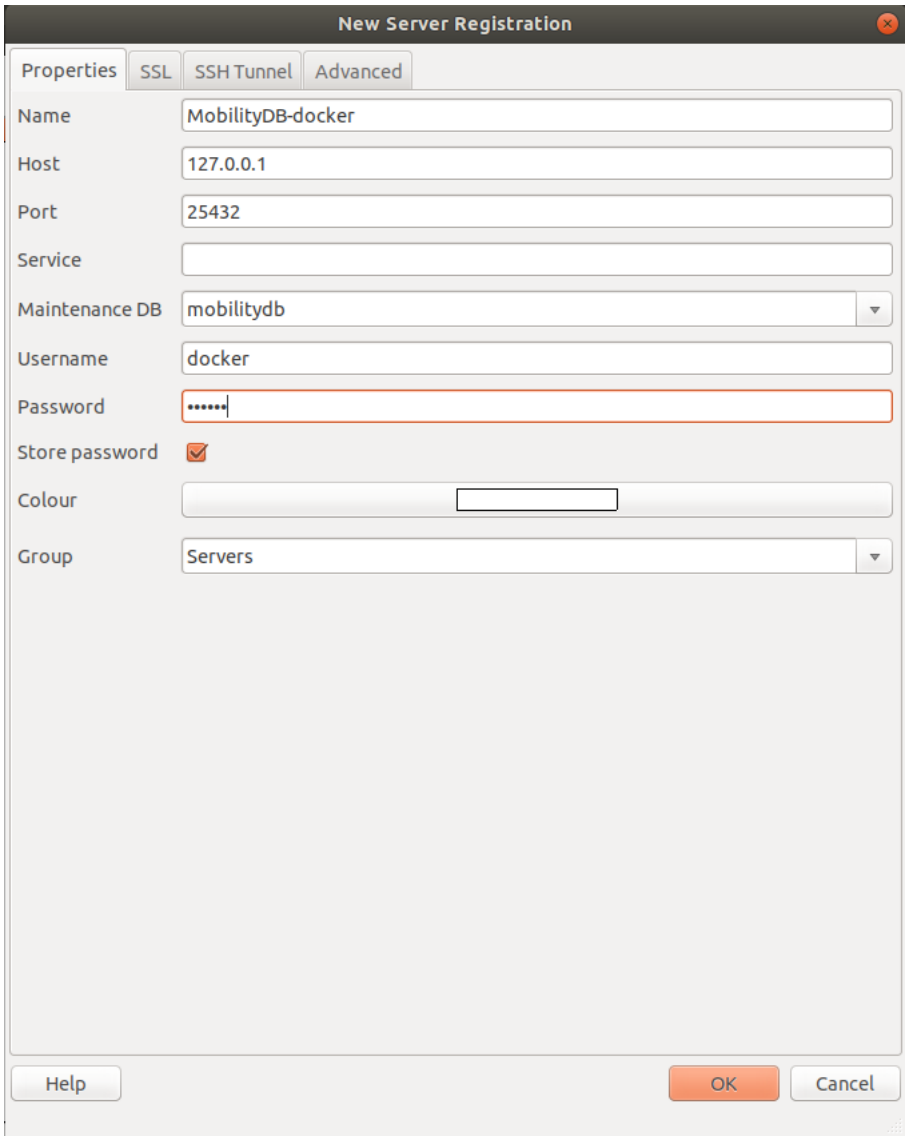


Figure 1.1: Configuration of a connection to the docker image in pgAdmin.

The second option is to know the IP address used by docker container with the following command.

```
docker-machine ip
192.168.99.101
```

Notice that the address obtained in your computer may be different from the one above. Now we can launch `pgAdmin` and establish a new connection to the docker container. This is done as shown in Figure 1.1. The second option is to set the host to the localhost (127.0.0.1), and the port to the mapped one on the host, as per the `docker run` command. In this example the port would be 25432.

Now you can use pgAdmin to query the mobilitydb database, as will be further explained in the following sections. Here are few more docker commands that you will eventually need:

```
docker stop "mobilitydb"
docker start "mobilitydb"
docker rm "mobilitydb"
```

The above commands

- `docker stop` shuts down the docker container. You need to issue this command, for example, if you need to re-start the host.
- `docker start` launches back the docker container. You need to issue this command, for example, after re-starting the host.
- `docker rm` removes/deletes docker container. You need to issue this command, for example, if you need to `docker pull` a more recent MobilityDB image. If the databases are stored in a `docker volume` as explained above, it will still be available after downloading and running the new image.

1.2 Loading the Data

The ZIP file with the data for this tutorial contains a set of CSV files as follows:

- `datamcar.csv` with fields `Moid`, `Licence`, `Type`, and `Model` contains the vehicle descriptions (without position history).
- `trips.csv` with fields `Moid`, `Tripid`, `Pos_x`, `Pos_y`, and `Instant` contains vehicles movements and pauses.
- `queryinstants.csv` with fields `Id` and `Instant` contains timestamps used for queries.
- `queryperiods.csv` with fields `Id`, `Begin`, and `End` contains periods used for the queries.
- `querypoints.csv` with fields `Id`, `Pos_x`, and `Pos_y` contains points used for queries.
- `queryregions.csv` with fields `Id`, `SegNo`, `Xstart`, `Ystart`, `Xend`, and `Yend` contains the polygons used for queries.

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

```
unzip berlinmod_data.zip
```

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

In the following, we can use the `mobilitydb` database provided in the container. This database has already installed the MobilityDB extension. Alternatively, you may use another database. In that case, you can install the MobilityDB extension in your database by using the following command.

```
CREATE EXTENSION MobilityDB CASCADE;
```

By using `CASCADE` we load the required PostGIS extension prior to loading MobilityDB.

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

```
CREATE TABLE Cars (
  CarId integer PRIMARY KEY,
  Licence varchar(32),
  Type varchar(32),
  Model varchar(32)
);
CREATE TABLE TripsInput (
  CarId integer REFERENCES Cars,
  TripId integer,
  Lon float,
  Lat float,
  T timestamptz,
  PRIMARY KEY (CarId, TripId, T)
);
CREATE TABLE Instants (
  InstantId integer PRIMARY KEY,
  Instant timestamptz
);
CREATE TABLE Periods (
  PeriodId integer PRIMARY KEY,
  Tstart TimestampTz,
  Tend TimestampTz,
  Period period
);
CREATE TABLE Points (
  PointId integer PRIMARY KEY,
  PosX double precision,
  PosY double precision,
  Geom Geometry(Point)
);
CREATE TABLE RegionsInput
(
  RegionId integer,
  SegNo integer,
  XStart double precision,
  YStart double precision,
  XEnd double precision,
  YEnd double precision,
  PRIMARY KEY (RegionId, SegNo)
);
CREATE TABLE Regions (
  RegionId integer PRIMARY KEY,
  Geom Geometry(Polygon)
);
CREATE TABLE Trips
(
  CarId integer NOT NULL,
  TripId integer NOT NULL,
  Trip tgeompoint,
  PRIMARY KEY (CarId, TripId),
  FOREIGN KEY (CarId) REFERENCES Cars (CarId)
);
```

We created one table for each CSV file renaming attributes for better readability. In addition, we created a table `Regions` in order to assemble all segments composing a region into a single geometry and a table `Trips` in order to assemble all points composing a trip into a single temporal point.

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

```
COPY Cars(CarId, Licence, Type, Model) FROM '/home/mobilitydb/data/datamcar.csv'
  DELIMITER  ',' CSV HEADER;
COPY TripsInput(CarId, TripId, Lon, Lat, T) FROM '/home/mobilitydb/data/trips.csv'
```

```
  DELIMITER  ',' CSV HEADER;
COPY Instants(InstantId, Instant) FROM '/home/mobilitydb/data/queryinstants.csv'
  DELIMITER  ',' CSV HEADER;
COPY Periods(PeriodId, Tstart, Tend) FROM '/home/mobilitydb/data/queryperiods.csv'
  DELIMITER  ',' CSV HEADER;
UPDATE Periods
SET Period = period(Tstart, Tend);
COPY Points(PointId, PosX, PosY) FROM '/home/mobilitydb/data/querypoints.csv'
  DELIMITER  ',' CSV HEADER;
UPDATE Points
SET Geom = ST_Transform(ST_SetSRID(ST_MakePoint(PosX, PosY), 4326), 5676);
COPY RegionsInput(RegionId, SegNo, XStart, YStart, XEnd, YEnd) FROM
  '/home/mobilitydb/data/queryregions.csv' DELIMITER  ',' CSV HEADER;
```

The following query is used to load table `Regions` from the data in table `RegionsInput`.

```
INSERT INTO Regions (RegionId, Geom)
WITH RegionsSegs AS
(
  SELECT RegionId, SegNo,
    ST_Transform(ST_SetSRID(St_MakeLine(ST_MakePoint(XStart, YStart),
    ST_MakePoint(XEnd, YEnd)), 4326), 5676) AS Geom
  FROM RegionsInput
)
SELECT RegionId, ST_Polygon(ST_LineMerge(ST_Union(Geom ORDER BY SegNo)), 5676) AS Geom
FROM RegionsSegs
GROUP BY RegionId;
```

The following query is used to load table `Trips` from the data in table `TripsInput`.

```
INSERT INTO Trips
SELECT CarId, TripId, tgeompointseq(array_agg(tgeompointinst(
  ST_Transform(ST_SetSRID(ST_MakePoint(Lon,Lat), 4326), 5676), T) ORDER BY T))
FROM TripsInput
GROUP BY CarId, TripId;
```

There are a lot of nested functions, so reading from the innermost:

- Function `ST_MakePoint` construct a point from the `Lon` and `Lat` values.
- Function `ST_SetSRID` sets the SRID of the point to 4326, that is, to the standard WGS 84 GPS coordinates.
- Function `ST_Transform` transforms the spherical GPS coordinates to plannar coordinates fitted for Germany.
- Function `tgeompointinst` gets the point and the time values to create a temporal point of instant duration.
- Function `array_agg` collects in an array all temporal points of a given car and a given trip (as specified by the `GROUP BY` clause) and sort them by time (as specified by the `ORDER BY` clause)
- Function `tgeompointseq` gets the array of temporal points and construct a temporal point of sequence duration.

Finally, we create indexes on traditional, spatial, temporal or spatiotemporal attributes as well as views to select a subset of the rows from the corresponding tables. This can be done as follows.

```
CREATE UNIQUE INDEX Cars_CarId_Idx ON Cars USING btree(CarId);
CREATE INDEX Instants_Instant_Idx ON Instants USING btree(Instant);
CREATE INDEX Periods_Period_Idx ON Periods USING gist(Period);
CREATE INDEX Points_Geom_Idx ON Points USING gist(Geom);
CREATE INDEX Regions_Geom_Idx ON Regions USING gist(Geom);
CREATE INDEX Trips_CarId_idx ON Trips USING btree(CarId);
CREATE UNIQUE INDEX Trips_pkey_idx ON Trips USING btree(CarId, TripId);
CREATE INDEX Trips_gist_idx ON Trips USING gist(trip);
```

```
CREATE VIEW Instants1 AS SELECT * FROM Instants LIMIT 10;
CREATE VIEW Periods1 AS SELECT * FROM Periods LIMIT 10;
CREATE VIEW Points1 AS SELECT * FROM Points LIMIT 10;
CREATE VIEW Regions1 AS SELECT * FROM Regions LIMIT 10;
CREATE VIEW Trips1 AS SELECT * FROM Trips LIMIT 100;
```

1.3 Loading the Data in Partitioned Tables

PostgreSQL provides partitioning mechanisms so that large tables can be split in smaller physical tables. This may result in increased performance when querying and manipulating large tables. We will split the `Trips` table given in the previous section using list partitioning, where each partitition will contain all the trips that start at a particular date. For doing this, we use the procedure given next for automatically creating the partitions according to a date range.

```
CREATE OR REPLACE FUNCTION create_partitions_by_date(TableName TEXT, StartDate DATE,
  EndDate DATE)
RETURNS void AS $$
DECLARE
  d DATE;
  PartitionName TEXT;
BEGIN
  IF NOT EXISTS
    (SELECT 1
     FROM information_schema.tables
     WHERE table_name = lower(TableName))
  THEN
    RAISE EXCEPTION 'Table % does not exist', TableName;
  END IF;
  IF StartDate >= EndDate THEN
    RAISE EXCEPTION 'The start date % must be before the end date %', StartDate, EndDate;
  END IF;
  d = StartDate;
  WHILE d <= EndDate
  LOOP
    PartitionName = TableName || '_' || to_char(d, 'YYYY_MM_DD');
    IF NOT EXISTS
      (SELECT 1
       FROM information_schema.tables
       WHERE  table_name = lower(PartitionName))
    THEN
      EXECUTE format('CREATE TABLE %s PARTITION OF %s FOR VALUES IN (''%s'');',
        PartitionName, TableName, to_char(d, 'YYYY-MM-DD'));
      RAISE NOTICE 'Partition % has been created', PartitionName;
    END IF;
    d = d + '1 day'::interval;
  END LOOP;
  RETURN;
END
$$ LANGUAGE plpgsql;
```

In order to partition table `Trips` by date we need to add an addition column `TripDate` to table `TripsInput`.

```
ALTER TABLE TripsInput ADD COLUMN TripDate DATE;
UPDATE TripsInput T1
SET TripDate = T2.TripDate
FROM (SELECT DISTINCT TripId, date_trunc('day', MIN(T) OVER (PARTITION BY TripId))
  AS TripDate FROM TripsInput) T2
WHERE T1.TripId = T2.TripId;
```

Notice that the `UPDATE` statement above takes into account the fact that a trip may finish at a day later than the starting day.

The following statements create table `Trips` partitioned by date and the associated partitions.

```
CREATE TABLE Trips
(
  CarId integer NOT NULL,
  TripId integer NOT NULL,
  TripDate date,
  Trip tgeompoint,
  Traj geometry,
  PRIMARY KEY (CarId, TripId, TripDate),
  FOREIGN KEY (CarId) REFERENCES Cars (CarId)
) PARTITION BY LIST(TripDate);

SELECT create_partitions_by_date('Trips', (SELECT MIN(TripDate) FROM TripsInput),
  (SELECT MAX(TripDate) FROM TripsInput));
```

To see the partitions that have been created automatically we can use the following statement.

```
SELECT I.inhrelid::regclass AS child
FROM   pg_inherits I
WHERE  i.inhparent = 'trips'::regclass;
```

In our case this would result in the following output.

```
"trips_2007_05_27"
"trips_2007_05_28"
"trips_2007_05_29"
"trips_2007_05_30"
```

We modify the query that loads table `Trips` from the data in table `TripsInput` as follows.

```
INSERT INTO Trips
SELECT CarId, TripId, TripDate, tgeompointseq(array_agg(tgeompointinst(
  ST_Transform(ST_SetSRID(ST_MakePoint(Lon,Lat), 4326), 5676), T) ORDER BY T))
FROM TripsInput
GROUP BY CarId, TripId, TripDate;
```

Then, we can define the indexes and the views on the table `Trips` as shown in the previous section.

An important advantange of the partitioning mechanism in PostgreSQL is that the constraints and the indexes defined on the `Trips` table are propagated to the partitions as shown next.

```
INSERT INTO Trips VALUES (1, 10, '2007-05-30', NULL);

ERROR:  duplicate key value violates unique constraint "trips_2007_05_30_pkey"
DETAIL:  Key (carid, TripId, tripdate)=(1, 10, 2007-05-30) already exists.

EXPLAIN SELECT COUNT(*) from Trips where Trip && period '[2007-05-28, 2007-05-29]';

"Aggregate  (cost=59.95..59.96 rows=1 width=8)"
"  ->  Append  (cost=0.14..59.93 rows=8 width=0)"
"        ->  Index Scan using trips_2007_05_27_trip_idx on trips_2007_05_27  (cost ↩
    =0.14..8.16 rows=1 width=0)"
"              Index Cond: (trip && 'STBOX T((,,2007-05-28 00:00:00+00),(,,2007-05-29  ↩
    00:00:00+00))'::stbox)"
"        ->  Index Scan using trips_2007_05_28_trip_idx on trips_2007_05_28  (cost ↩
    =0.27..8.29 rows=1 width=0)"
"              Index Cond: (trip && 'STBOX T((,,2007-05-28 00:00:00+00),(,,2007-05-29  ↩
    00:00:00+00))'::stbox)"
"        ->  Index Scan using trips_2007_05_29_trip_idx on trips_2007_05_29  (cost ↩
    =0.27..8.29 rows=1 width=0)"
```

```
"              Index Cond: (trip && 'STBOX T((,,2007-05-28 00:00:00+00),(,,2007-05-29  ↩
    00:00:00+00))'::stbox)"
[...]
```

1.4 Exploring the Data

In order to visualize the data with traditional tools such as QGIS we add to table `Trip` a column `Traj` of type `geometry` containing the trajectory of the trips.

```
ALTER TABLE Trips ADD COLUMN traj geometry;
UPDATE Trips
SET Traj = trajectory(Trip);
```

The visualization of the trajectories in QGIS is given in Figure 1.2. In the figure red lines correspond to the trajectories of moving cars, while yellow points correspond to the position of stationary cars. In order to know the total number of trips as well as the number of moving and stationary trips we can issue the following queries.

```
SELECT count(*) FROM Trips;
-- 1797
SELECT count(*) FROM Trips WHERE GeometryType(Traj) = 'POINT';
-- 969
SELECT count(*) FROM Trips WHERE GeometryType(Traj) = 'LINESTRING';
-- 828
```

We can also determine the spatiotemporal extent of the data using the following query.

```
SELECT extent(Trip) from Trips
-- "STBOX T((2983189.5, 5831006.5,2007-05-27 00:00:00+02),
  (3021179.8, 5860883,2007-05-31 00:00:00+02))"
```

We continue investigating the data set by computing the maximum number of concurrent trips over the whole period

```
SELECT maxValue(tcount(Trip)) FROM Trips;
-- 141
```

the average sampling rate

```
SELECT AVG(timespan(Trip)/numInstants(Trip)) FROM Trips;
-- "03:43:01.695539"
```

and the total travelled distance in kilometers of all trips:

```
SELECT SUM(length(Trip)) / 1e3 as TotalLengthKm FROM Trips;
-- 10074.8123345527
```

Now we want to know the average duration of a trip.

```
SELECT AVG(timespan(Trip)) FROM Trips;
--"07:31:57.195325"
```

This average duration is too long. To investigate more we use the following query

```
SELECT length(Trip) / 1e3, timespan(Trip) FROM Trips ORDER BY duration;
```

The query shows very many trips with zero length and a duration of more than one day. That would imply that there are stationary trips, representing parking overnight and even over the weekend. The previous query can hence be refined as follows:

```
SELECT AVG(timespan(Trip)/numInstants(Trip)) FROM Trips WHERE length(Trip) > 0;
-- "00:00:01.861784"
```

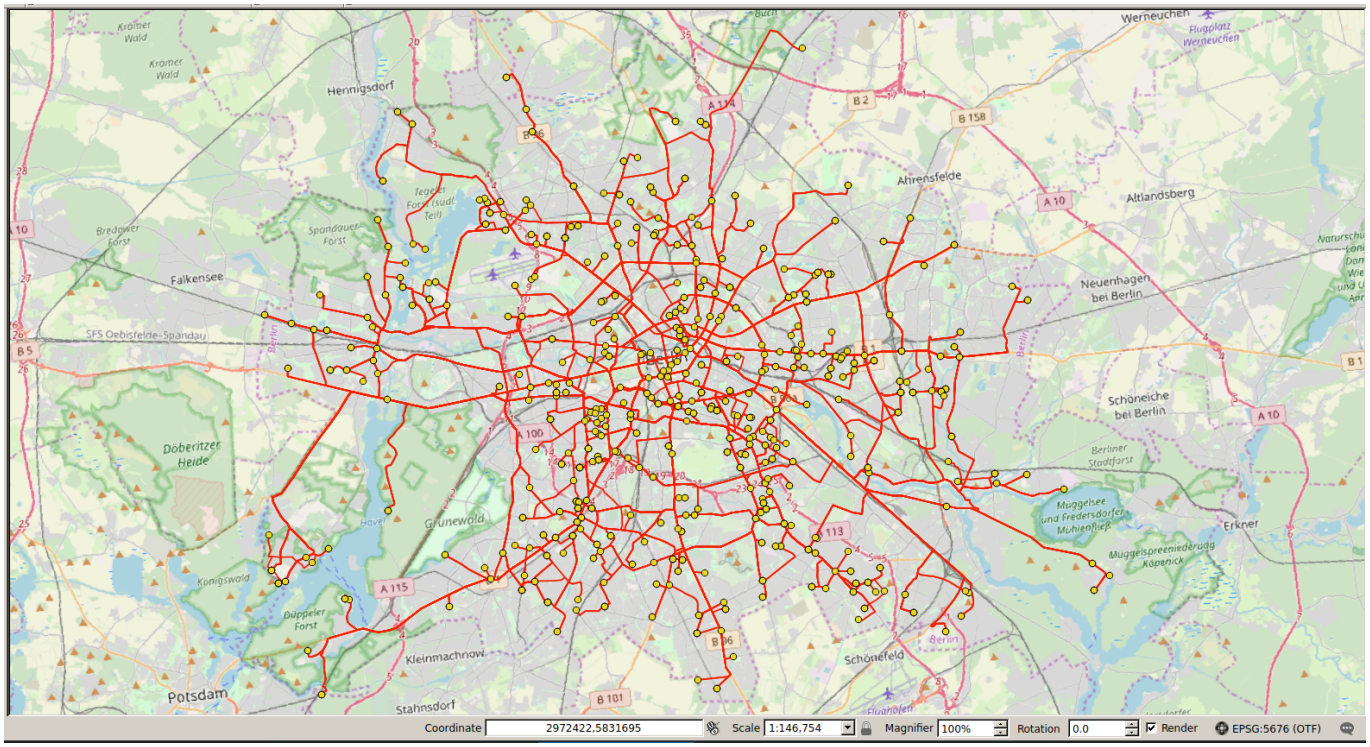


Figure 1.2: Visualization of the trajectories of the trips in QGIS.

The following query produces a histogram of trip length.

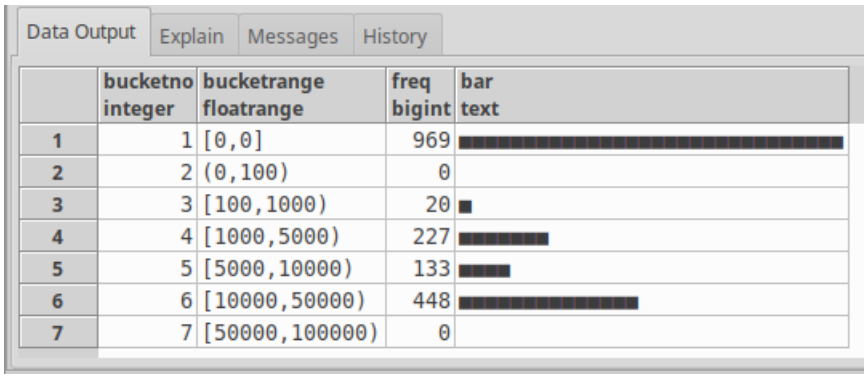
```
WITH buckets (bucketNo, bucketRange) AS (
  SELECT 1, floatrange '[0, 0]' UNION
  SELECT 2, floatrange '(0, 100)' UNION
  SELECT 3, floatrange '[100, 1000)' UNION
  SELECT 4, floatrange '[1000, 5000)' UNION
  SELECT 5, floatrange '[5000, 10000)' UNION
  SELECT 6, floatrange '[10000, 50000)' UNION
  SELECT 7, floatrange '[50000, 100000)' ),
histogram AS (
  SELECT bucketNo, bucketRange, count(TripId) as freq
  FROM buckets left outer join trips on length(trip) <@ bucketRange
  GROUP BY bucketNo, bucketRange
  ORDER BY bucketNo, bucketRange
)
SELECT bucketNo, bucketRange, freq,
  repeat('■', ( freq::float / max(freq) OVER () * 30 )::int ) AS bar
FROM histogram;
```

The result of the above query is given in Figure 1.3.

1.5 Querying the Data

We discuss next four categories of queries: range queries, distance queries, temporal aggregate queries, and nearest-neighbor queries[1].

[1] A web interface to explore the temporal types on a database containing the BerlinMOD benchmark data generated at scale 0.005 is available at the address here.



Data Output | Explain | Messages | History

	bucketno integer	bucketrange floatrange	freq bigint	bar text
1	1	[0,0]	969	
2	2	(0,100)	0	
3	3	[100,1000)	20	
4	4	[1000,5000)	227	
5	5	[5000,10000)	133	
6	6	[10000,50000)	448	
7	7	[50000,100000)	0	

Figure 1.3: Result of the query building an histogram of trip lengths.

1.5.1 Range Queries

The queries in this category restrict Trips with respect to a spatial, temporal, or spatio-temporal point or range. In the examples, the spatial points and ranges are given, respectively, in tables `Points` and `Regions`, while temporal points and ranges are given, respectively, in tables `Instants` and `Periods`.

1. List the cars that have passed at a region from `Regions`.

```
SELECT DISTINCT R.RegionId, T.CarId
FROM Trips T, Regions R
WHERE ST_Intersects(trajectory(T.Trip), R.Geom)
ORDER BY R.RegionId, T.CarId;
```

   This is a spatial range query. The query verifies that the trajectory of the car intersects the region. PostGIS performs an implicit bounding box comparison `trajectory(T.Trip) && R.Geom` using the spatial index on table `Regions` when executing the predicate `ST_Intersects`.

2. List the cars that were within a region from `Regions` during a period from `Periods`.

```
SELECT R.RegionId, P.PeriodId, T.CarId
FROM Trips T, Regions R, Periods P
WHERE T.Trip && stbox(R.Geom, P.Period) AND
  _intersects(atPeriod(T.Trip, P.Period), R.Geom)
ORDER BY R.RegionId, P.PeriodId, T.CarId;
```

   This is a spatio-temporal range query. The query performs a bounding box comparison with the `&&` operator using the spatio-temporal index on table `Trips`. After that, the query verifies that the location of the car during the period intersects the region. Notice that the predicate `_intersects` is used instead of `intersects` to avoid an implicit index test with the bounding box comparison `atPeriod(Trip, P.Period) && R.Geom` is performed using the spatio-temporal index.

3. List the pairs of cars that were both located within a region from `Regions` during a period from `Periods`.

```
SELECT DISTINCT T1.CarId AS CarId1, T2.CarId AS CarId2, R.RegionId, P.PeriodId
FROM Trips T1, Trips T2, Regions R, Periods P
WHERE T1.CarId < T2.CarId AND T1.Trip && stbox(R.Geom, P.Period) AND
  T2.Trip && stbox(R.Geom, P.Period) AND
  _intersects(atPeriod(T1.Trip, P.Period), R.Geom) AND
  _intersects(atPeriod(T2.Trip, P.Period), R.Geom)
ORDER BY T1.CarId, T2.CarId, R.RegionId, P.PeriodId;
```

   This is a spatio-temporal range join query. The query selects two trips of different cars and performs bounding box comparisons of each trip with a region and a period using the spatio-temporal index of the `Trips` table. The query then verifies that both cars were located within the region during the period.

4. List the first time at which a car visited a point in `Points`.

```
SELECT T.CarId, P.PointId, MIN(startTimestamp(atValue(T.Trip,P.Geom))) AS Instant
FROM Trips T, Points P
WHERE ST_Contains(trajectory(T.Trip), P.Geom)
GROUP BY T.CarId, P.PointId;
```

The query selects a trip and a point and verifies that the car passed by the point by testing that the trajectory of the trip contains the point. Notice that PostGIS will perform the bounding box containment `trajectory(T.Trip) ~ P.Geom` using the spatial index on table `Points` before executing `ST_Contains`. Then, the query projects the trip to the point with the `atValue` function, get the first timestamp of the projected trip with the `startTimestamp` function, and applies the traditional `MIN` aggregate function for all trips of the car and the point.

1.5.2 Temporal Aggregate Queries

There are three common types of temporal aggregate queries.

- Instant temporal aggregate queries in which, from a conceptual perspective, the traditional aggregate function is applied at each instant.
- Window temporal aggregate queries (also known as cumulative queries), which, given a time interval w, compute the value of the aggregate at a time instant t from the values during the time period [t-w, t].
- Span temporal aggregate queries, which, first, split the time line into predefined intervals independently of the target data, and then, for each of these intervals, aggregate the data that overlap the interval.

5. Compute how many cars were active at each period in `Periods`.

```
SELECT P.PeriodID, COUNT(*), TCOUNT(atPeriod(T.Trip, P.Period))
FROM Trips T, Periods P
WHERE T.Trip && P.Period
GROUP BY P.PeriodID
ORDER BY P.PeriodID;
```

This an instant temporal aggregate query. For each period, the query projects the trips to the given period and applies the temporal count to the projected trips. The condition in the `WHERE` clause is used for filtering the trips with the spatio-temporal index on table `Trips`.

6. For each region in `Regions`, give the window temporal count of trips with a 10-minute interval.

```
SELECT R.RegionID, WCOUNT(atGeometry(T.Trip, R.Geom), interval '10 min')
FROM Trips T, Regions R
WHERE T.Trip && R.Geom
GROUP BY R.RegionID
HAVING WCOUNT(atGeometry(T.Trip, R.Geom), interval '10 min') IS NOT NULL
ORDER BY R.RegionID;
```

This is a window temporal aggregate query. Suppose that we are computing pollution levels by region. Since the effect of a car passing at a location lasts some time interval, this is a typical case for window aggregates. For each region, the query computes the spatial projection of the trips to the given region and apply the window temporal count to the projected trips. The condition in the `WHERE` clause is used for filtering the trips with the spatio-temporal index. The condition in the `HAVING` clause is used for removing regions that do not intersect with any trip.

7. Count the number of trips that were active during each hour in May 29, 2007.

```
WITH TimeSplit(Period) AS (
  SELECT period(H, H + interval '1 hour')
  FROM generate_series(timestamptz '2007-05-29 00:00:00',
    timestamptz '2007-05-29 23:00:00', interval '1 hour') AS H )
SELECT Period, COUNT(*)
```

```
FROM TimeSplit S, Trips T
WHERE S.Period && T.Trip AND atPeriod(Trip, Period) IS NOT NULL
GROUP BY S.Period
ORDER BY S.Period;
```

This is a span temporal aggregate query. The query defines the intervals to consider in the `TimeSplit` temporary table. For each of these intervals, the main query applies the traditional count function for counting the trips that overlap the interval.

1.5.3 Distance queries

The queries in this category deal with either the distance travelled by a single object or the distance between two objects. The complexity of the latter queries depend, on the one hand, on whether the reference objects are static or moving, and on the other, on whether the operation required is either the minimum distance ever or the temporal distance computed at each instant.

8. List the overall traveled distances of the cars during the periods from `Periods`.

```
SELECT T.CarId, P.PeriodId, P.Period,
  SUM(length(atPeriod(T.Trip, P.Period))) AS Distance
FROM Trips T, Periods P
WHERE T.Trip && P.Period
GROUP BY T.CarId, P.PeriodId, P.Period
ORDER BY T.CarId, P.PeriodId;
```

The query performs a bounding box comparison with the `&&` operator using the spatio-temporal index on the `Trips` table. It then projects the trip to the period, computes the length of the projected trip, and sum the lengths of all the trips of the same car during the period.

9. List the minimum distance ever between each car and each point from `Points`.

```
SELECT T.CarId, P.PointId, MIN(trajectory(T.Trip) <-> P.Geom) AS MinDistance
FROM Trips T, Points P
GROUP BY T.CarId, P.PointId
ORDER BY T.CarId, P.PointId;
```

The query projects the trip to the spatial dimension with the `trajectory` function and computes the traditional distance between the trajectory of the trip and the point. The traditional minimum function is then applied for computing the minimum distance between all trips of the car and the point.

10. List the minimum temporal distance between each pair of cars.

```
SELECT T1.CarId AS Car1Id, T2.CarId AS Car2Id, MIN(T1.Trip <-> T2.Trip) AS MinDistance
FROM Trips T1, Trips T
WHERE T1.CarId < T2.CarId AND period(T1.Trip) && period(T2.Trip)
GROUP BY T1.CarId, T2.CarId
ORDER BY T1.CarId, T2.CarId;
```

The query selects two trips `T1` and `T2` from different cars that were both traveling during a common period of time, computes the temporal distance between the trips, and then computes the temporal minimum distance between all trips of the two cars. The query uses the spatio-temporal index to filter the pairs of trips that were both traveling during a common period of time.

11. List the nearest approach time, distance, and shortest line between each pair of trips.

```
SELECT T1.CarId AS Car1Id, T1.TripId AS Trip1Id, T2.CarId AS Car2Id,
  T2.TripId AS Trip2Id, period(NearestApproachInstant(T1.Trip, T2.Trip)) AS Time,
  NearestApproachDistance(T1.Trip, T2.Trip) AS Distance,
  ShortestLine(T1.Trip, T2.Trip) AS Line
FROM Trips1 T1, Trips1 T
WHERE T1.CarId < T2.CarId AND period(T1.Trip) && period(T2.Trip)
ORDER BY T1.CarId, T1.TripId, T2.CarId, T2.TripId;
```

This query shows similar functionality as that provided by the PostGIS functions `ST_ClosestPointOfApproach` and `ST_DistanceCPA`. The query selects two trips `T1` and `T2` from different cars that were both traveling during a common period of time and computes the required results.

12. List when and where a pairs of cars have been at 10 m or less from each other.

```
SELECT T1.CarId AS CarId1, T2.CarId AS CarId2, atPeriodSet(T1.Trip,
  period(atValue(tdwithin(T1.Trip, T2.Trip, 10.0), TRUE))) AS Position
FROM Trips T1, Trips T
WHERE T1.CarId < T2.CarId AND T1.Trip && expandSpatial(T2.Trip, 10) AND
  atPeriodSet(T1.Trip, period(atValue(tdwithin(T1.Trip, T2.Trip, 10.0), TRUE)))
  IS NOT NULL
ORDER BY T1.CarId, T2.CarId, Position;
```

The query performs for each pair of trips `T1` and `T2` of different cars a bounding box comparison with the `&&` operator using the spatio-temporal index on the `Trips` table, where the bounding box of `T2` is expanded by 10 m. Then, the `period` expression computes the periods during which the cars were within 10 m. from each other and the `atPeriodSet` function projects the trips to those periods. Notice that the expression `tdwithin(T1.Trip, T2.Trip, 10.0)` is conceptually equivalent to `dwithin(T1.Trip, T2.Trip) #<= 10.0`. However, in this case the spatio-temporal index cannot be used for filtering values.

1.5.4 Nearest-Neighbor Queries

There are three common types of nearest-neighbor queries in spatial databases.

- k-nearest-neighbor (kNN) queries find the k nearest points to a given point.
- Reverse k-nearest-neighbor (RkNN) queries find the points that have a given point among their k nearest-neighbors.
- Given two sets of points P and Q, aggregate nearest-neighbor (ANN) queries find the points from P that have minimum aggregated distance to all points from Q.

The above types of queries are generalized to temporal points. However, the complexity of these queries depend on whether the reference object and the candidate objects are static or moving. In the examples that follow we only consider the nontemporal version of the nearest-neighbor queries, that is, the one in which the calculation is performed on the projection of temporal points on the spatial dimension. The temporal version of the nearest-neighbor queries remains to be done.

13. For each trip from `Trips`, list the three points from `Points` that have been closest to that car.

```
WITH TripsTraj AS (
  SELECT *, trajectory(Trip) AS Trajectory FROM Trips )
SELECT T.CarId, P1.PointId, P1.Distance
FROM TripsTraj T CROSS JOIN LATERAL (
SELECT P.PointId, T.Trajectory <-> P.Geom AS Distance
FROM Points P
ORDER BY Distance LIMIT 3 ) AS P1
ORDER BY T.TripId, T.CarId, P1.Distance;
```

This is a nearest-neighbor query with moving reference objects and static candidate objects. The query above uses PostgreSQL's lateral join, which intuitively iterates over each row in a result set and evaluates a subquery using that row as a parameter. The query starts by computing the trajectory of the trips in the temporary table `TripsTraj`. Then, given a trip `T` in the outer query, the subquery computes the traditional distance between the trajectory of `T` and each point `P`. The `ORDER BY` and `LIMIT` clauses in the inner query select the three closest points. PostGIS will use the spatial index on the `Points` table for selecting the three closest points.

14. For each trip from `Trips`, list the three cars that are closest to that car

```
SELECT T1.CarId AS CarId1, C2.CarId AS CarId2, C2.Distance
FROM Trips T1 CROSS JOIN LATERAL (
SELECT T2.CarId, minValue(T1.Trip <-> T2.Trip) AS Distance
FROM Trips T2
WHERE T1.CarId < T2.CarId AND period(T1.Trip) && period(T2.Trip)
ORDER BY Distance LIMIT 3 ) AS C2
ORDER BY T1.CarId, C2.CarId;
```

This is a nearest-neighbor query where both the reference and the candidate objects are moving. Therefore, it is not possible to proceed as in the previous query to first project the moving points to the spatial dimension and then compute the traditional distance. Given a trip `T1` in the outer query, the subquery computes the temporal distance between `T1` and a trip `T2` of another car different from the car from `T1` and then computes the minimum value in the temporal distance. Finally, the `ORDER BY` and `LIMIT` clauses in the inner query select the three closest cars.

15. For each trip from `Trips`, list the points from `Points` that have that car among their three nearest neighbors.

```
WITH TripsTraj AS (
  SELECT *, trajectory(Trip) AS Trajectory FROM Trips ),
PointTrips AS (
  SELECT P.PointId, T2.CarId, T2.TripId, T2.Distance
  FROM Points P CROSS JOIN LATERAL (
  SELECT T1.CarId, T1.TripId, P.Geom <-> T1.Trajectory AS Distance
  FROM TripsTraj T
  ORDER BY Distance LIMIT 3 ) AS T2 )
SELECT T.CarId, T.TripId, P.PointId, PT.Distance
FROM Trips T CROSS JOIN Points P JOIN PointTrips PT
ON T.CarId = PT.CarId AND T.TripId = PT.TripId AND P.PointId = PT.PointId
ORDER BY T.CarId, T.TripId, P.PointId;
```

This is a reverse nearest-neighbor query with moving reference objects and static candidate objects. The query starts by computing the corresponding nearest-neighbor query in the temporary table `PointTrips` as it is done in Query 13. Then, in the main query it verifies for each trip `T` and point `P` that both belong to the `PointTrips` table.

16. For each trip from `Trips`, list the cars having the car of the trip among the three nearest neighbors.

```
WITH TripDistances AS (
  SELECT T1.CarId AS CarId1, T1.TripId AS TripId1, T3.CarId AS CarId2,
    T3.TripId AS TripId2, T3.Distance
  FROM Trips T1 CROSS JOIN LATERAL (
  SELECT T2.CarId, T2.TripId, minValue(T1.Trip <-> T2.Trip) AS Distance
  FROM Trips T
  WHERE T1.CarId < T2.CarId AND period(T1.Trip) && period(T2.Trip)
  ORDER BY Distance LIMIT 3 ) AS T3 )
SELECT T1.CarId, T1.TripId, T2.CarId, T2.TripId, TD.Distance
FROM Trips T1 JOIN Trips T2 ON T1.CarId < T2.CarId
  JOIN TripDistances TD ON T1.CarId = TD.CarId1 AND T1.TripId = TD.TripId1 AND
  T2.CarId = TD.CarId2 AND T2.TripId = TD.TripId
ORDER BY T1.CarId, T1.TripId, T2.CarId, T2.TripId;
```

This is a reverse nearest-neighbor query where both the reference and the candidate objects are moving. The query starts by computing the corresponding nearest-neighbor query in the temporary table `TripDistances` as it is done in Query 14. Then, in the main query it verifies for each pair of trips `T1` and `T2` that both belong to the `TripDistances` table.

17. For each group of ten disjoint cars, list the point(s) from `Points`, having the minimum aggregated distance from the given group of ten cars during the given period.

```
WITH Groups AS (
  SELECT ((ROW_NUMBER() OVER (ORDER BY C.CarId))-1)/10 + 1 AS GroupId, C.CarId
  FROM Cars C ),
SumDistances AS (
  SELECT G.GroupId, P.PointId,
```

```
    SUM(ST_Distance(trajectory(T.Trip), P.Geom)) AS SumDist
  FROM Groups G, Points P, Trips T
  WHERE T.CarId = G.CarId
  GROUP BY G.GroupId, P.PointId )
SELECT S1.GroupId, S1.PointId, S1.SumDist
FROM SumDistances S
WHERE S1.SumDist <= ALL (
  SELECT SumDist
  FROM SumDistances S
  WHERE S1.GroupId = S2.GroupId )
  ORDER BY S1.GroupId, S1.PointId;
```

This is an aggregate nearest-neighbor query. The temporary table `Groups` splits the cars in groups where the `GroupId` column takes the values from 1 to total number of groups. The temporary table `SumDistances` computes for each group `G` and point `P` the sum of the distances between a trip of a car in the group and the point. The main query then selects for each group in table `SumDistances` the points(s) that have the minimum aggregated distance.

Chapter 2

Generating Realistic Trajectory Datasets

2.1 Introduction

Do you need an arbitrarily large trajectory dataset to tests your ideas? This chapter 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 leisure 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.
- Drivers will slow down in curves
- Trips between home and work do not include additional destinations
- Leisure trips start and end at home locations and include multiple destinations

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.

2.2 Contents

This chapter 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

2.3 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.

2.4 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 and work nodes, transforms data to SRID 3857, 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, messages := 'medium')" 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'`, `'verbose'`, or `'debug'`.

Figure 2.1 shows a visualization of the trips generated.

2.5 Exploring the Generated Data

Now use a PostgreSQL client such as psql or pgAdmin to explore the properties of the generated trajecotries. 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;

1686  "618:34:23.478239"  20546.31859281626
```

We continue by further analyzing the duration of all the trips

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

"00:00:29.091033" "01:13:21.225514" "00:22:02.365486"
```

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'
```

Figure 2.1: 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.

```
    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;

"leisure_weekday"    558    "00:00:29.091033" "00:57:30.195709" "00:10:59.118318"
"work_home"          564    "00:02:04.159342" "01:13:21.225514" "00:27:33.424924"
"home_work"          564    "00:01:57.456419" "01:11:44.551344" "00:27:25.145454"
```

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;

0.2731400585134866  53.76566616928331 12.200901777206806
```

As can be seen the longest trip is more than 56 Km long. Let's visualize one of these long trips.

```
SELECT vehicle, seq, source, target, round(length(Trip)::numeric / 1e3, 3),
  startTimestamp(Trip), timespan(Trip)
FROM Trips
WHERE  length(Trip) > 50000 LIMIT 1;

90  1 23078 11985 53.766  "2020-06-01 08:46:55.487+02"  "01:10:10.549413"
```

We can then visualize this trip in PostGIS. As can be seen, in Figure 2.2, the home and the work nodes of the vehicle are located at two extremities in Brussels.

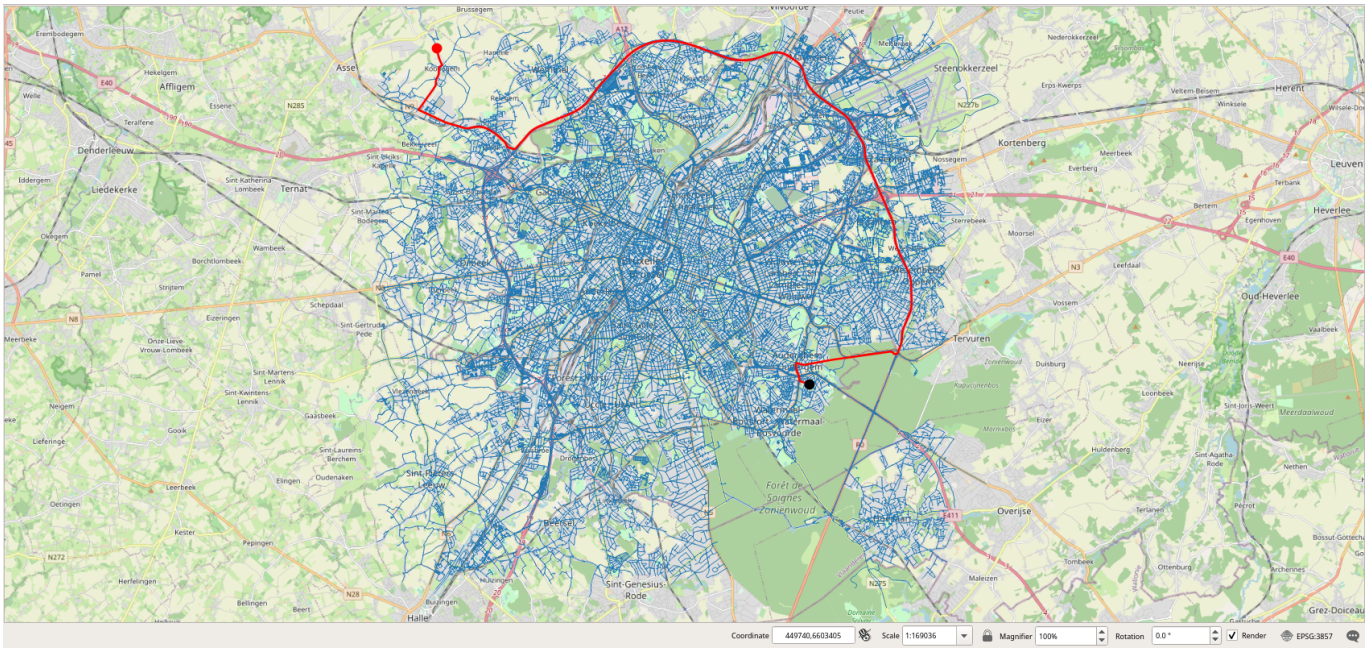


Figure 2.2: Visualization of a long trip.

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;

14.211962789552468  53.31779380411017 31.32438581663778
```

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 2.3 and the resulting visualization is shown in Figure 2.4.

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.

```
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 2.5 shows the visualization of the speed map without and with the base map.

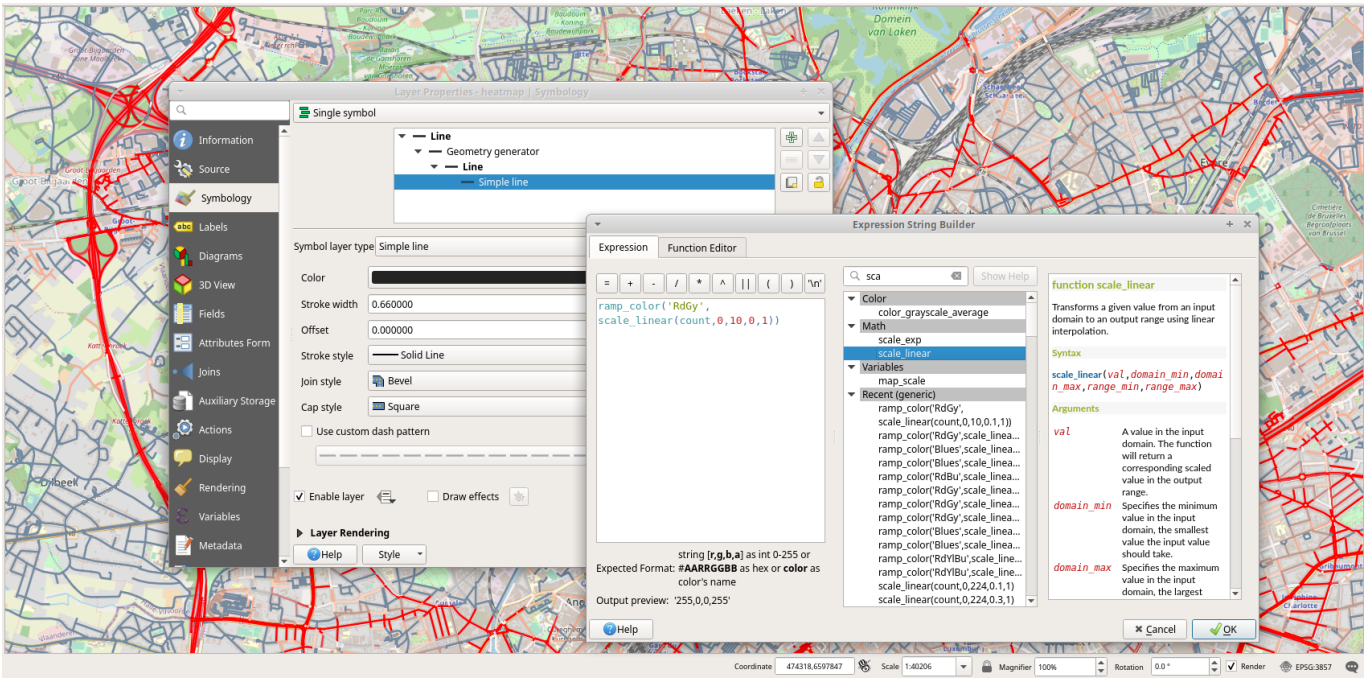


Figure 2.3: Assigning in QGIS a gradient color from blue to red according to the value of the attribute `count`.

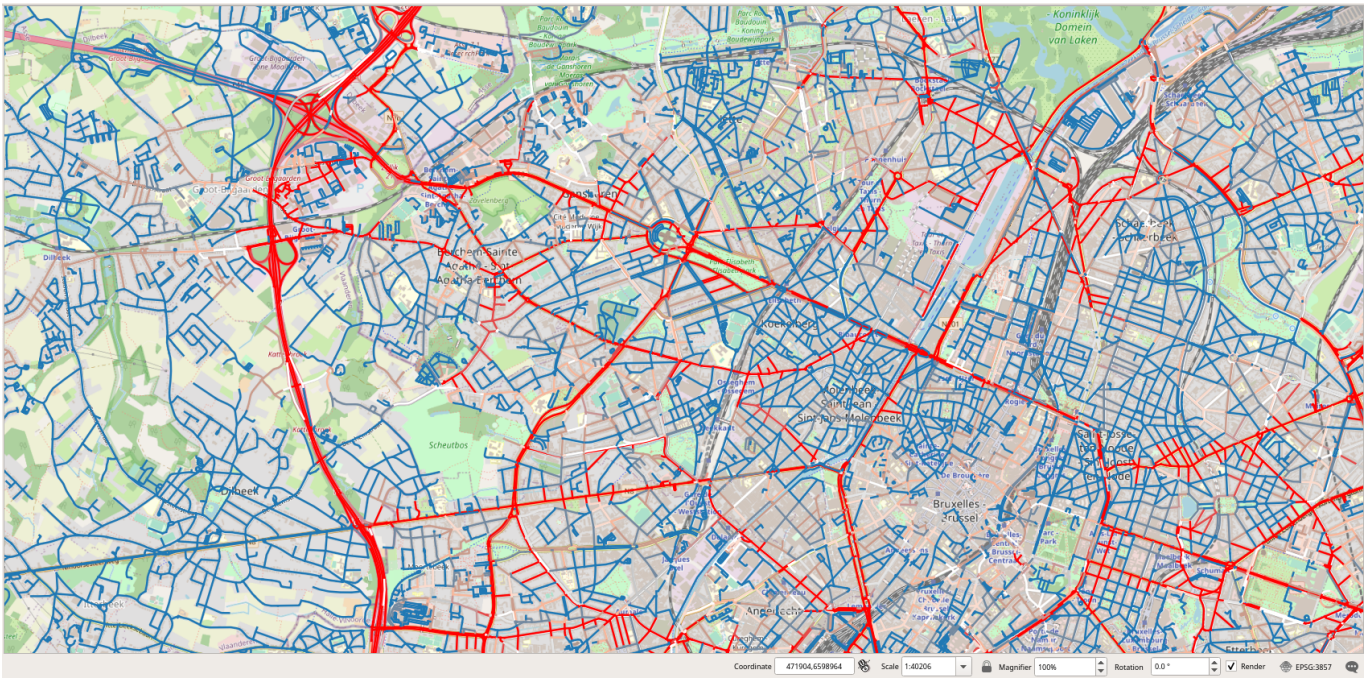


Figure 2.4: Visualization of the edges of the graph according to the number of trips that traversed the edges.

Figure 2.5: Visualization of the edges of the graph according to the speed of trips that traversed the edges.

2.6 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 2.8. 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
  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`.

2.7 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 2.6 shows how we can define the bounding box around Barcelona using the web site bboxfinder.

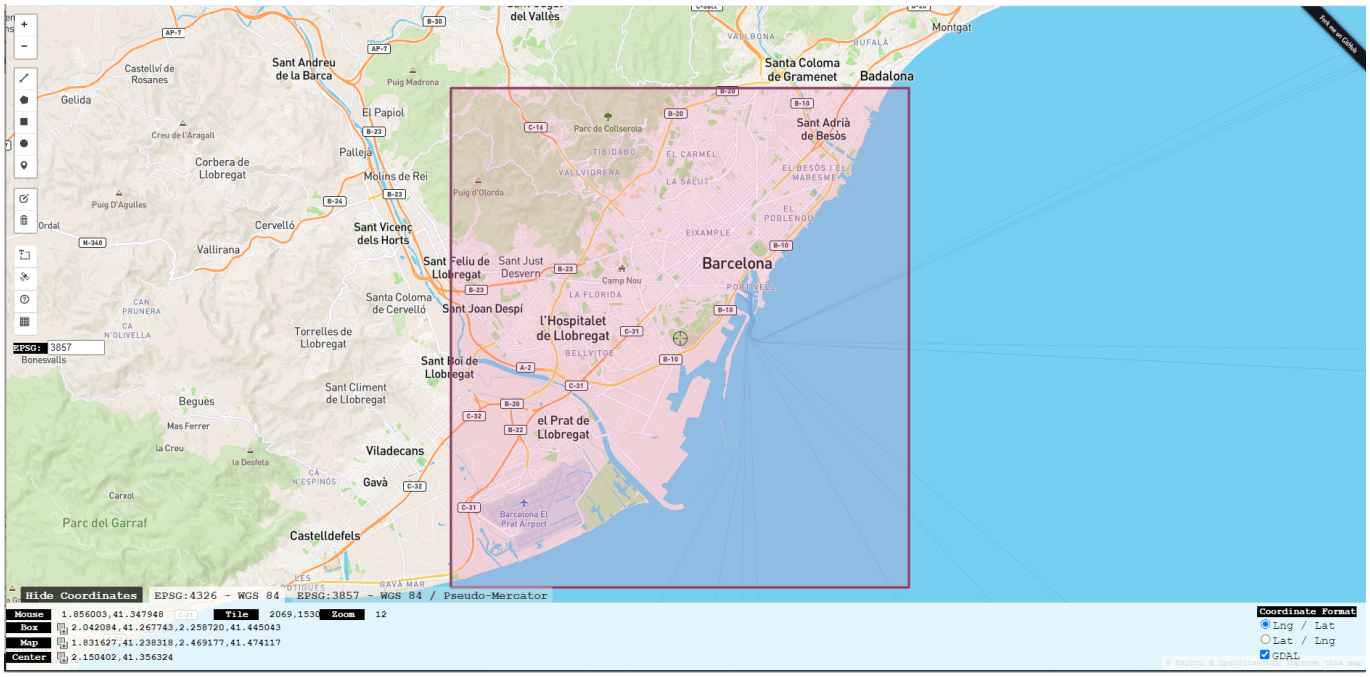


Figure 2.6: Defining the bounding box for obtaining OSM data from Barcelona.

After obtaining the bounding box, we can proceed as we stated in Section 2.4. We create a new database `barcelona`, then add both PostGIS, MobilityDB, and pgRouting to it.

```
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
```

2.8 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.

2.9 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 2.7 and Figure 2.8 show visualizations of the data generated for the deliveries scenario.

2.10 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.

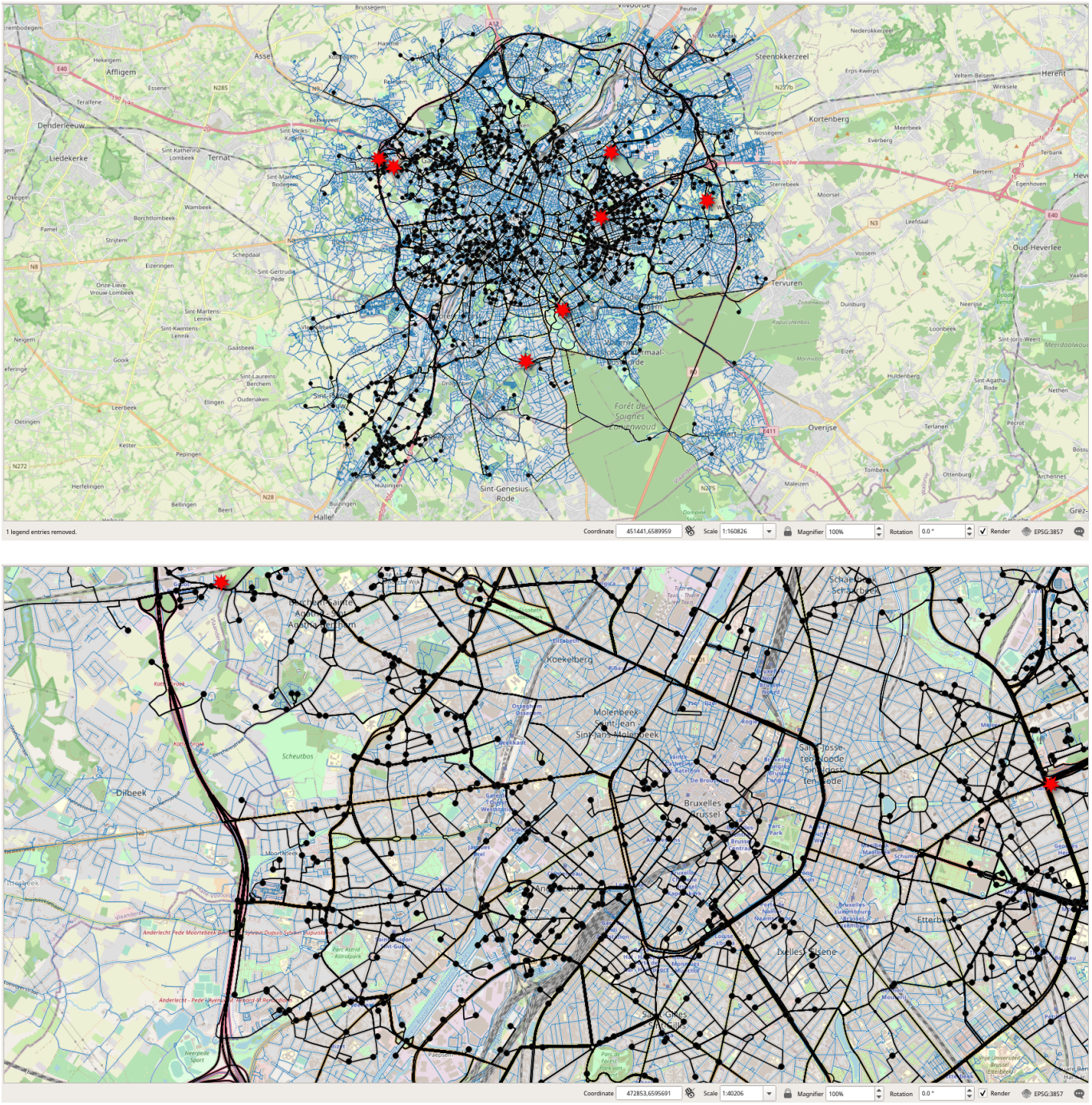


Figure 2.7: Visualization of the data generated for the deliveries scenario. The road network is shown with blue lines, the warehouses are shown with a red star, the routes taken by the deliveries are shown with black lines, and the location of the customers with black points.

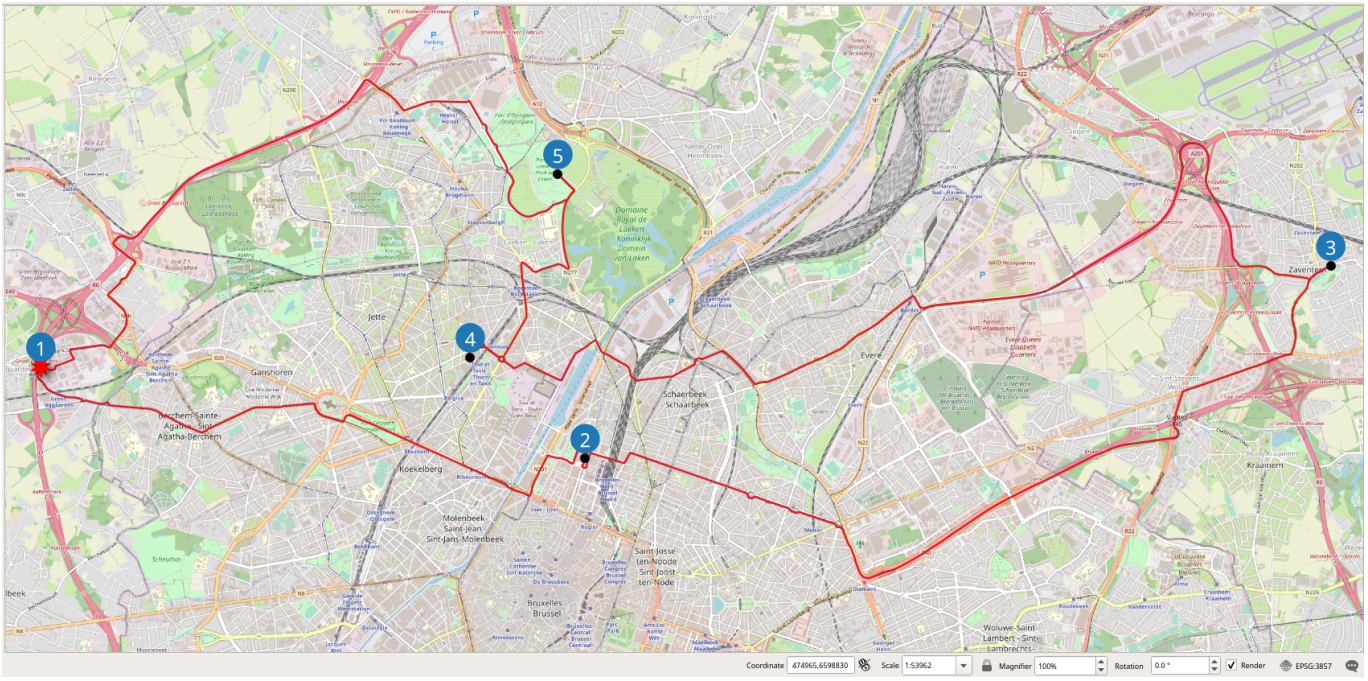


Figure 2.8: Visualization of the deliveries of one vehicle during one day. A delivery trip starts and ends at a warehouse and make the deliveries to several customers, four in this case.

2.10.1 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
```

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 2.9, 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.

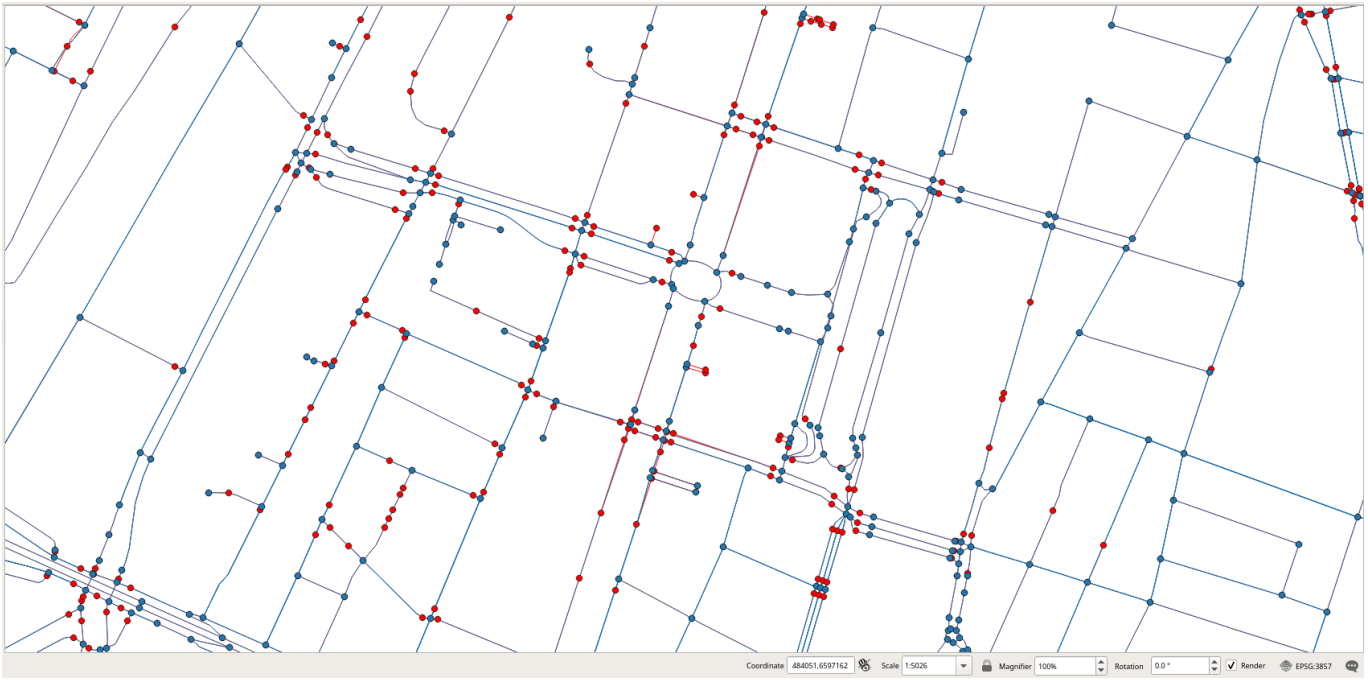


Figure 2.9: Comparison of the nodes obtained (in blue) with those obtained by osm2pgrouting (in red).

2.10.2 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 2.10.1 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 2.10 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.

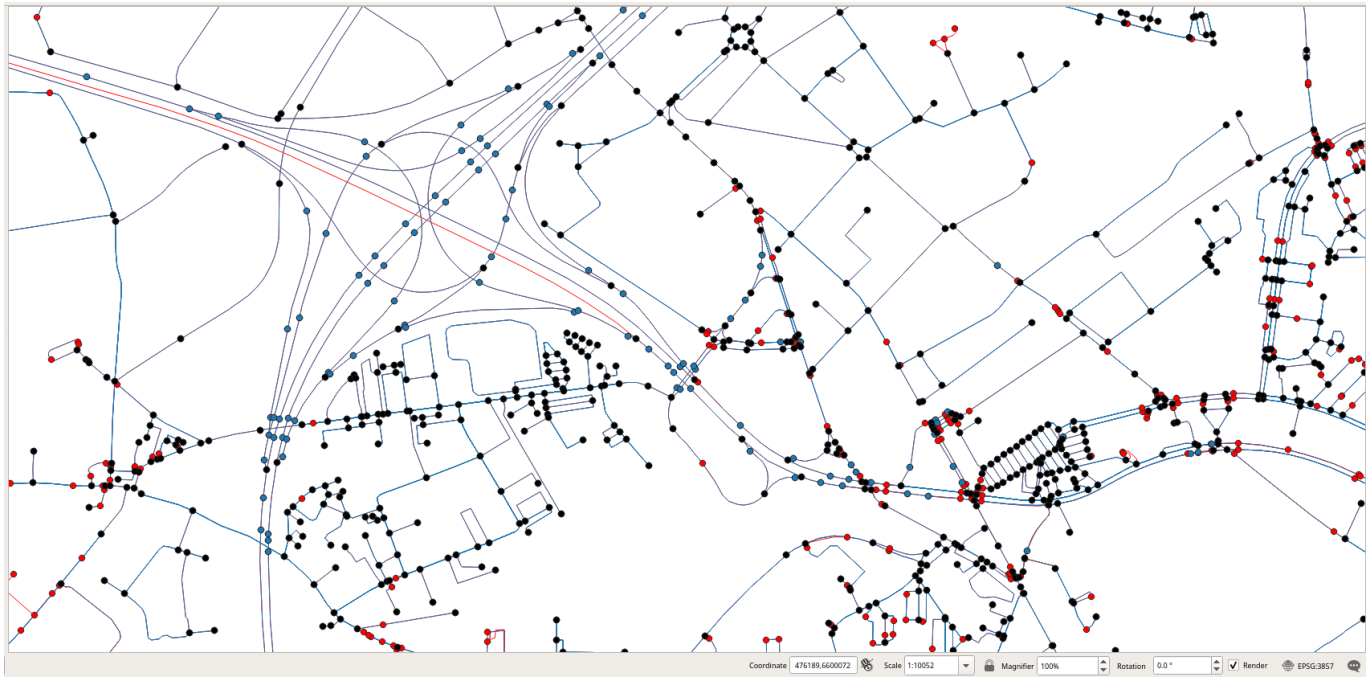


Figure 2.10: Comparison of the nodes obtained by contracting the graph (in black), before contraction (in blue), and those obtained by osm2pgrouting (in red).

Chapter 3

BerlinMOD Benchmark on MobilityDB

BerlinMOD is a standard benchmark for moving object DBMSs. It provides a data generator, pregenerated benchmark data for different scale factors, and set of queries of two types: 17 range-style queries (called BerlinMOD/R), and 9 nearest-neighbours queries (called BerlinMOD/NN). The MobilityDB tutorial presented in Chapter 1 and its associated data were based on BerlinMOD. However, its purpose was to show the capabilities of MobilityDB. In this chapter, we show how to load pregenerated BerlinMOD data on MobilityDB and how to express the 17 queries in BerlinMOD/R. Some of these queries were already presented in Chapter 1.

3.1 Loading the Data

The script for loading pregenerated data is available here.

```
------------------------------------------------------------------------------
-- Loads the BerlinMOD data in projected (2D) coordinates with SRID 5676
-- https://epsg.io/5676
------------------------------------------------------------------------------

DROP FUNCTION IF EXISTS berlinmod_load();
CREATE OR REPLACE FUNCTION berlinmod_load(scale_factor text DEFAULT '0.005',
  path text DEFAULT '/usr/local/BerlinMOD/')
RETURNS text AS $$
DECLARE
  fullpath text;
BEGIN
  fullpath = path || scale_factor || '/';
  DROP TABLE IF EXISTS streets;
  CREATE TABLE streets
  (
    StreetId integer,
    vmax integer,
    x1 double precision,
    y1 double precision,
    x2 double precision,
    y2 double precision,
    Geom geometry(LineString, 5676)
  );
  EXECUTE format('COPY streets(StreetId, vmax, x1, y1, x2, y2) FROM ''%sstreets.csv''
    DELIMITER '','' CSV HEADER', fullpath);
  UPDATE streets
  SET Geom = ST_Transform(ST_SetSRID(ST_MakeLine(ARRAY[ST_MakePoint(x1, y1),
    ST_MakePoint(x2, y2)]), 4326), 5676);

  DROP TABLE IF EXISTS Points CASCADE;
```

```
CREATE TABLE Points
(
  PointId integer,
  PosX double precision,
  PosY double precision,
  Geom geometry(Point, 5676)
);
EXECUTE format('COPY Points(PointId, PosX, PosY) FROM ''%squerypoints.csv''
  DELIMITER  '','' CSV HEADER', fullpath);
UPDATE Points
SET Geom = ST_Transform(ST_SetSRID(ST_MakePoint(PosX, PosY), 4326), 5676);

CREATE INDEX Points_geom_idx ON Points USING gist(Geom);

CREATE VIEW Points1(PointId, PosX, PosY, Geom) AS
SELECT PointId, PosX, PosY, Geom
FROM Points
LIMIT 10;

DROP TABLE IF EXISTS RegionsInput CASCADE;
CREATE TABLE RegionsInput
(
  RegionId integer,
  SegNo integer,
  XStart double precision,
  YStart double precision,
  XEnd double precision,
  YEnd double precision
);
EXECUTE format('COPY RegionsInput(RegionId, SegNo, XStart, YStart, XEnd, YEnd)
  FROM ''%squeryregions.csv'' DELIMITER  '','' CSV HEADER', fullpath);

DROP TABLE IF EXISTS Regions CASCADE;
CREATE TABLE Regions
(
  RegionId integer,
  Geom Geometry(Polygon, 5676)
);
INSERT INTO Regions (RegionId, Geom)
WITH RegionsSegs AS
(
  SELECT RegionId, SegNo, ST_Transform(ST_SetSRID(St_MakeLine(
    ST_MakePoint(XStart, YStart), ST_MakePoint(XEnd, YEnd)), 4326), 5676) AS Geom
  FROM RegionsInput
)
SELECT RegionId, ST_Polygon(ST_LineMerge(ST_Union(Geom ORDER BY SegNo)), 5676) AS Geom
FROM RegionsSegs
GROUP BY RegionId;

CREATE INDEX Regions_geom_idx ON Regions USING gist(Geom);

CREATE VIEW Regions1(RegionId, Geom) AS
SELECT RegionId, Geom
FROM Regions
LIMIT 10;

DROP TABLE IF EXISTS Instants CASCADE;
CREATE TABLE Instants
(
  InstantId integer,
  Instant timestamptz
);
```

```
EXECUTE format('COPY Instants(InstantId, Instant) FROM ''%squeryinstants.csv''
  DELIMITER  '','' CSV HEADER', fullpath);

CREATE INDEX Instants_instant_btree_idx ON Instants USING btree(instant);

CREATE VIEW Instants1(InstantId, Instant) AS
SELECT InstantId, Instant
FROM Instants
LIMIT 10;

DROP TABLE IF EXISTS Periods CASCADE;
CREATE TABLE Periods
(
  PeriodId integer,
  BeginP timestamp,
  EndP timestamp,
  Period period
);
EXECUTE format('COPY Periods(PeriodId, BeginP, EndP) FROM ''%squeryperiods.csv''
  DELIMITER  '','' CSV HEADER', fullpath);
UPDATE Periods
SET Period = period(BeginP,EndP);

CREATE INDEX Periods_Period_gist_idx ON Periods USING gist(Period);

CREATE VIEW Periods1(PeriodId, BeginP, EndP, Period) AS
SELECT PeriodId, BeginP, EndP, Period
FROM Periods
LIMIT 10;

DROP TABLE IF EXISTS Cars CASCADE;
CREATE TABLE Cars
(
  CarId integer primary key,
  Licence varchar(32),
  Type varchar(32),
  Model varchar(32)
);
EXECUTE format('COPY Cars(CarId, Licence, Type, Model) FROM ''%sdatamcar.csv''
  DELIMITER  '','' CSV HEADER', fullpath);

CREATE UNIQUE INDEX Cars_CarId_idx ON Cars USING btree(CarId);

DROP TABLE IF EXISTS Licences CASCADE;
CREATE TABLE Licences
(
  LicenceId integer,
  Licence varchar(8),
  CarId integer
);
EXECUTE format('COPY Licences(Licence, LicenceId) FROM ''%squerylicences.csv''
  DELIMITER  '','' CSV HEADER', fullpath);
UPDATE Licences Q
SET CarId = ( SELECT C.CarId FROM Cars C WHERE C.Licence = Q.Licence );

CREATE INDEX Licences_CarId_idx ON Licences USING btree(CarId);

CREATE VIEW Licences1(LicenceId, Licence, CarId) AS
SELECT LicenceId, Licence, CarId
FROM Licences
LIMIT 10;
```

```
CREATE VIEW Licences2(LicenceId, Licence, CarId) AS
SELECT LicenceId, Licence, CarId
FROM Licences
LIMIT 10 OFFSET 10;

DROP TABLE IF EXISTS TripsInput CASCADE;
CREATE TABLE TripsInput (
  CarId integer,
  TripId integer,
  TStart timestamp without time zone,
  TEnd timestamp without time zone,
  XStart double precision,
  YStart double precision,
  XEnd double precision,
  YEnd double precision,
  Geom geometry(LineString)
);
EXECUTE format('COPY TripsInput(CarId, TripId, TStart, TEnd, XStart, YStart, XEnd, YEnd)
  FROM ''%strips.csv'' DELIMITER  '','' CSV HEADER', fullpath);
UPDATE TripsInput
SET Geom = ST_Transform(ST_SetSRID(ST_MakeLine(ARRAY[ST_MakePoint(XStart, YStart),
  ST_MakePoint(XEnd, YEnd)]), 4326), 5676);

DROP TABLE IF EXISTS TripsInputInstants;
CREATE TABLE TripsInputInstants AS (
SELECT CarId, TripId, TStart, XStart, YStart,
  ST_Transform(ST_SetSRID(ST_MakePoint(XStart, YStart), 4326), 5676) as Geom
FROM TripsInput
UNION ALL
SELECT T1.CarId, T1.TripId, T1.TEnd, T1.XEnd, T1.YEnd,
  ST_Transform(ST_SetSRID(ST_MakePoint(T1.XEnd, T1.YEnd), 4326), 5676) as Geom
FROM TripsInput T1 INNER JOIN (
  SELECT CarId, TripId, max(TEnd) as MaxTend
  FROM TripsInput
  GROUP BY CarId, TripId
) T2 ON T1.CarId = T2.CarId AND T1.TripId = T2.TripId AND T1.TEnd = T2.MaxTend );
ALTER TABLE TripsInputInstants ADD COLUMN inst tgeompoint;
UPDATE TripsInputInstants
SET inst = tgeompointinst(Geom, TStart);

DROP TABLE IF EXISTS Trips CASCADE;
CREATE TABLE Trips (
  CarId integer NOT NULL,
  TripId integer NOT NULL,
  Trip tgeompoint,
  Traj geometry,
  PRIMARY KEY (CarId, TripId),
  FOREIGN KEY (CarId) REFERENCES Cars (CarId)
);
INSERT INTO Trips
  SELECT CarId, TripId, tgeompointseq(array_agg(inst ORDER BY TStart))
  FROM TripsInputInstants
  GROUP BY CarId, TripId;
UPDATE Trips
SET Traj = trajectory(Trip);

CREATE INDEX Trips_CarId_idx ON Trips USING btree(CarId);
CREATE UNIQUE INDEX Trips_pkey_idx ON Trips USING btree(CarId, TripId);
CREATE INDEX Trips_gist_idx ON Trips USING gist(trip);

DROP VIEW IF EXISTS Trips1;
CREATE VIEW Trips1 AS
```

```
  SELECT * FROM Trips LIMIT 100;

  -- Drop temporary tables
  DROP TABLE RegionsInput;
  DROP TABLE TripsInput;
  DROP TABLE TripsInputInstants;

  RETURN 'The End';
END;
$$ LANGUAGE 'plpgsql';
------------------------------------------------------------------------------
```

The script above creates a procedure to load pregenerated BerlinMOD data (in CSV format and WGS84 coordinates) at various scale factors. The procedure has two parameters: the scale factor and the directory where the CSV files are located. It supposes by default that the scale factor is `0.005` and that the CSV files are located in the directory `/usr/local/BerlinMOD/<scale factor>/`. Notice that the procedure creates GiST indexes for the tables. Alternatively, SP-GiST indexes could be used. The procedure can be called, for example, as follows.

```
SELECT berlinmod_load('0.05');
```

3.2 Loading the Data in Partitioned Tables

As we discussed in Chapter 1, partioning allows one to split a large table into smaller physical pieces. We show next how to modify the scripts given in the previous section to take advantage of partioning. We will partition the `Trips` table by date using list partitioning, where each partitition will contain all the trips that start at a particular date. We will use the procedure `create_partitions_by_date` shown in Chapter 1 for automatically creating the partitions according to the date range of the corresponding scale factor.

```
[...]
DROP TABLE IF EXISTS TripsInput CASCADE;
CREATE TABLE TripsInput (
  CarId integer,
  TripId integer,
  TripDate date,
  TStart timestamp without time zone,
  TEnd timestamp without time zone,
  XStart double precision,
  YStart double precision,
  XEnd double precision,
  YEnd double precision,
  Geom geometry(LineString)
);
EXECUTE format('COPY TripsInput(CarId, TripId, TStart, TEnd, XStart, YStart, XEnd, YEnd)
  FROM ''%strips.csv'' DELIMITER  '','' CSV HEADER', fullpath);
UPDATE TripsInput
SET Geom = ST_Transform(ST_SetSRID(ST_MakeLine(ARRAY[ST_MakePoint(XStart, YStart),
    ST_MakePoint(XEnd, YEnd)]), 4326), 5676);
UPDATE TripsInput T1
SET TripDate = T2.TripDate
FROM (SELECT DISTINCT TripId, date_trunc('day', MIN(TStart) OVER
  (PARTITION BY TripId)) AS TripDate FROM TripsInput) T2
WHERE T1.TripId = T2.TripId;
[...]
DROP TABLE IF EXISTS Trips CASCADE;
CREATE TABLE Trips (
  CarId integer NOT NULL,
  TripId integer NOT NULL,
  TripDate date,
  Trip tgeompoint,
```

```
  Traj geometry,
  PRIMARY KEY (CarId, TripId, TripDate),
  FOREIGN KEY (CarId) REFERENCES Cars (CarId)
) PARTITION BY LIST(TripDate);

-- Create the partitions
SELECT MIN(TripDate), MAX(TripDate) INTO mindate, maxdate FROM TripsInputInstants;
PERFORM create_partitions_by_date('Trips', mindate, maxdate);

INSERT INTO Trips(CarId, TripId, TripDate, Trip)
SELECT CarId, TripId, TripDate, tgeompointseq(array_agg(inst ORDER BY TStart))
FROM TripsInputInstants
GROUP BY CarId, TripId, TripDate;
UPDATE Trips
SET Traj = trajectory(Trip);

CREATE INDEX Trips_CarId_idx ON Trips USING btree(CarId);
CREATE UNIQUE INDEX Trips_pkey_idx ON Trips USING btree(CarId, TripId, TripDate);
CREATE INDEX Trips_gist_idx ON Trips USING gist(trip);
[...]
```

With respect to the script given in the previous section, we need to add an additional column `TripDate` to the tables `TripsInput`, `TripsInputInstants` (not shown), and `Trips` that will be used for partitioning.

3.3 BerlinMOD/R Queries

The script for querying BerlinMOD data loaded in MobilityDB with the BerlinMOD/R queries is available here.

1. What are the models of the vehicles with licence plate numbers from `Licences`?

```
SELECT DISTINCT L.Licence, C.Model AS Model
FROM Cars C, Licences L
WHERE C.Licence = L.Licence;
```

2. How many vehicles exist that are passenger cars?

```
SELECT COUNT (Licence)
FROM Cars C
WHERE Type = 'passenger';
```

3. Where have the vehicles with licences from `Licences1` been at each of the instants from `Instants1`?

```
SELECT DISTINCT L.Licence, I.InstantId, I.Instant AS Instant,
  valueAtTimestamp(T.Trip, I.Instant) AS Pos
FROM Trips T, Licences1 L, Instants1 I
WHERE T.CarId = L.CarId AND valueAtTimestamp(T.Trip, I.Instant) IS NOT NULL
ORDER BY L.Licence, I.InstantId;
```

4. Which vehicles have passed the points from `Points`?

```
SELECT DISTINCT P.PointId, P.Geom, C.Licence
FROM Trips T, Cars C, Points P
WHERE T.CarId = C.CarId AND T.Trip && P.Geom
AND ST_Intersects(trajectory(T.Trip), P.Geom)
ORDER BY P.PointId, C.Licence;
```

5. What is the minimum distance between places, where a vehicle with a licence from `Licences1` and a vehicle with a licence from `Licences2` have been?

```
SELECT L1.Licence AS Licence1, L2.Licence AS Licence2,
  MIN(ST_Distance(trajectory(T1.Trip), trajectory(T2.Trip))) AS MinDist
FROM Trips T1, Licences1 L1, Trips T2, Licences2 L2
WHERE T1.CarId = L1.CarId AND T2.CarId = L2.CarId AND T1.CarId < T2.CarId
GROUP BY L1.Licence, L2.Licence
ORDER BY L1.Licence, L2.Licence;
```

6. What are the pairs of trucks that have ever been as close as 10m or less to each other?

```
SELECT DISTINCT C1.Licence AS Licence1, C2.Licence AS Licence2
FROM Trips T1, Cars C1, Trips T2, Cars C2
WHERE T1.CarId = C1.CarId AND T2.CarId = C2.CarId
AND T1.CarId < T2.CarId AND C1.Type = 'truck' AND C2.Type = 'truck'
AND T1.Trip && expandSpatial(T2.Trip, 10)
AND tdwithin(T1.Trip, T2.Trip, 10.0) ?= true
ORDER BY C1.Licence, C2.Licence;
```

7. What are the licence plate numbers of the passenger cars that have reached the points from `Points` first of all passenger cars during the complete observation period?

```
WITH Timestamps AS (
  SELECT DISTINCT C.Licence, P.PointId, P.Geom,
    MIN(startTimestamp(atValue(T.Trip,P.Geom))) AS Instant
  FROM Trips T, Cars C, Points1 P
  WHERE T.CarId = C.CarId AND C.Type = 'passenger'
  AND T.Trip && P.Geom AND ST_Intersects(trajectory(T.Trip), P.Geom)
  GROUP BY C.Licence, P.PointId, P.Geom
)
SELECT T1.Licence, T1.PointId, T1.Geom, T1.Instant
FROM Timestamps T1
WHERE T1.Instant <= ALL (
  SELECT T2.Instant
  FROM Timestamps T2
  WHERE T1.PointId = T2.PointId )
ORDER BY T1.PointId, T1.Licence;
```

8. What are the overall travelled distances of the vehicles with licence plate numbers from `Licences1` during the periods from `Periods1`?

```
SELECT L.Licence, P.PeriodId, P.Period, SUM(length(atPeriod(T.Trip, P.Period))) AS Dist
FROM Trips T, Licences1 L, Periods1 P
WHERE T.CarId = L.CarId AND T.Trip && P.Period
GROUP BY L.Licence, P.PeriodId, P.Period
ORDER BY L.Licence, P.PeriodId;
```

9. What is the longest distance that was travelled by a vehicle during each of the periods from `Periods`?

```
WITH Distances AS (
  SELECT P.PeriodId, P.Period, T.CarId, SUM(length(atPeriod(T.Trip, P.Period))) AS Dist
  FROM Trips T, Periods P
  WHERE T.Trip && P.Period
  GROUP BY P.PeriodId, P.Period, T.CarId
)
SELECT PeriodId, Period, MAX(Dist) AS MaxDist
FROM Distances
GROUP BY PeriodId, Period
ORDER BY PeriodId;
```

10. When and where did the vehicles with licence plate numbers from `Licences1` meet other vehicles (distance < 3m) and what are the latter licences?

```
WITH Values AS (
  SELECT DISTINCT L1.Licence AS QueryLicence, C2.Licence AS OtherLicence,
  atPeriodSet(T1.Trip, getTime(atValue(tdwithin(T1.Trip, T2.Trip, 3.0), TRUE))) AS Pos
  FROM Trips T1, Licences1 L1, Trips T2, Licences2 C2
  WHERE T1.CarId = L1.CarId AND T2.CarId = C2.CarId AND T1.CarId < T2.CarId
  AND expandSpatial(T1.Trip, 3) && expandSpatial(T2.Trip, 3)
  AND dwithin(T1.Trip, T2.Trip, 3.0)
)
SELECT QueryLicence, OtherLicence, array_agg(Pos ORDER BY startTimestamp(Pos)) AS Pos
FROM Values
GROUP BY QueryLicence, OtherLicence
ORDER BY QueryLicence, OtherLicence;
```

11. Which vehicles passed a point from `Points1` at one of the instants from `Instants1`?

```
SELECT P.PointId, P.Geom, I.InstantId, I.Instant, C.Licence
FROM Trips T, Cars C, Points1 P, Instants1 I
WHERE T.CarId = C.CarId AND T.Trip @> STBOX(P.Geom, I.Instant)
AND valueAtTimestamp(T.Trip, I.Instant) = P.Geom
ORDER BY P.PointId, I.InstantId, C.Licence;
```

12. Which vehicles met at a point from `Points1` at an instant from `Instants1`?

```
SELECT DISTINCT P.PointId, P.Geom, I.InstantId, I.Instant,
  C1.Licence AS Licence1, C2.Licence AS Licence2
FROM Trips T1, Cars C1, Trips T2, Cars C2, Points1 P, Instants1 I
WHERE T1.CarId = C1.CarId AND T2.CarId = C2.CarId AND T1.CarId < T2.CarId
AND T1.Trip @> STBOX(P.Geom, I.Instant) AND T2.Trip @> STBOX(P.Geom, I.Instant)
AND valueAtTimestamp(T1.Trip, I.Instant) = P.Geom
AND valueAtTimestamp(T2.Trip, I.Instant) = P.Geom
ORDER BY P.PointId, I.InstantId, C1.Licence, C2.Licence;
```

13. Which vehicles travelled within one of the regions from `Regions1` during the periods from `Periods1`?

```
SELECT DISTINCT R.RegionId, P.PeriodId, P.Period, C.Licence
FROM Trips T, Cars C, Regions1 R, Periods1 P
WHERE T.CarId = C.CarId AND T.trip && STBOX(R.Geom, P.Period)
AND ST_Intersects(trajectory(atPeriod(T.Trip, P.Period)), R.Geom)
ORDER BY R.RegionId, P.PeriodId, C.Licence;
```

14. Which vehicles travelled within one of the regions from `Regions1` at one of the instants from `Instants1`?

```
SELECT DISTINCT R.RegionId, I.InstantId, I.Instant, C.Licence
FROM Trips T, Cars C, Regions1 R, Instants1 I
WHERE T.CarId = C.CarId AND T.Trip && STBOX(R.Geom, I.Instant)
AND ST_Contains(R.Geom, valueAtTimestamp(T.Trip, I.Instant))
ORDER BY R.RegionId, I.InstantId, C.Licence;
```

15. Which vehicles passed a point from `Points1` during a period from `Periods1`?

```
SELECT DISTINCT PO.PointId, PO.Geom, PR.PeriodId, PR.Period, C.Licence
FROM Trips T, Cars C, Points1 PO, Periods1 PR
WHERE T.CarId = C.CarId AND T.Trip && STBOX(PO.Geom, PR.Period)
AND ST_Intersects(trajectory(atPeriod(T.Trip, PR.Period)),PO.Geom)
ORDER BY PO.PointId, PR.PeriodId, C.Licence;
```

16. List the pairs of licences for vehicles, the first from `Licences1`, the second from `Licences2`, where the corresponding vehicles are both present within a region from `Regions1` during a period from `QueryPeriod1`, but do not meet each other there and then.

```
SELECT P.PeriodId, P.Period, R.RegionId, L1.Licence AS Licence1, L2.Licence AS Licence2
FROM Trips T1, Licences1 L1, Trips T2, Licences2 L2, Periods1 P, Regions1 R
WHERE T1.CarId = L1.CarId AND T2.CarId = L2.CarId AND L1.Licence < L2.Licence
AND T1.Trip && STBOX(R.Geom, P.Period) AND T2.Trip && STBOX(R.Geom, P.Period)
AND ST_Intersects(trajectory(atPeriod(T1.Trip, P.Period)), R.Geom)
AND ST_Intersects(trajectory(atPeriod(T2.Trip, P.Period)), R.Geom)
AND tintersects(atPeriod(T1.Trip, P.Period), atPeriod(T2.Trip, P.Period)) %= FALSE
ORDER BY PeriodId, RegionId, Licence1, Licence2;
```

17. Which point(s) from `Points` have been visited by a maximum number of different vehicles?

```
WITH PointCount AS (
  SELECT P.PointId, COUNT(DISTINCT T.CarId) AS Hits
  FROM Trips T, Points P
  WHERE ST_Intersects(trajectory(T.Trip), P.Geom)
  GROUP BY P.PointId
)
SELECT PointId, Hits
FROM PointCount AS P
WHERE P.Hits = ( SELECT MAX(Hits) FROM PointCount );
```