MobilityDB Data Generator Workshop

COLLABORATORS

	TITLE : MobilityDB Data Generator Workshop	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Mahmoud SAKR and Esteban ZIMÁNYI	June 16, 2020	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

List of Figures

Abstract

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

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

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

Chapter 1

Generating Realistic Trajectory Datasets

1.1 Introduction

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

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

The data generator simulates as many cars and as many simulation days as needed. It models people trips using their cars to and from work during the week as well as some additional trips at evenings or weekends. The simulation uses multiple ideas to be close to reality, including:

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

The generator is written in PL/pgSQL, so that it will be easy to include own simulation rules. 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.

1.2 Contents

This module covers the following topics:

- A quick start using the generator
- Exploring the generated data
- Customizing the generator to your city
- Tuning the generator parameters
- Hacking the generator, and changing the simulation scenario
- Creating a graph to be used for the generator

1.3 Tools

- MobilityDB, hence PostgreSQL and PostGIS. The installation instruction can be found here.
- pgRouting. The installation instruction can be found here.

1.4 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.

```
CREATE EXTENSION MobilityDB CASCADE;
CREATE EXTENSION pgRouting;
```

Here we will use the OSM map of Brussels. In the next sections, we will explain how to use other maps. To download the map using the Overpass API, write the following in a terminal:

```
CITY="brussels"
BBOX="4.22,50.75,4.5,50.92"
wget --progress=dot:mega -O "$CITY.osm"
    "http://www.overpass-api.de/api/xapi?*[bbox=${BBOX}][@meta]"
```

or download the map from any OSM server.

To reduce the size of the OSM file:

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

The resulting file brussels.osm is also provided in the data section of this workshop. The data from the Overpass API is by default in Spherical Mercator (SRID 3857), so it is good for calculating distances. Next load the map and convert it into a routable format suitable for pgRouting.

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

The configuration file mapconfig_brussels.xml tells osm2pgrouting about the speed limits of the different road types. During the conversion, osm2pgrouting transforms the data into WGS84 (SRID 4326).

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 realisitc, we want to load a map of the administrative regoins of Brussuls (called communes), and feed the simulator with real statistics about the population, and the number of jobs 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 -d brussels brussels.osm
psql -d brussels -f brussels_preparedata.sql
```

Finally run the generator

```
psql -d brussels -f workweek_datagenerator.sql
psql -d brussels -c 'select workweek_generate()'
```

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

```
NOTICE:  ------------------------------------------------------------------
NOTICE:  Starting the work week data generator with Scale Factor 0.005
NOTICE:  ------------------------------------------------------------------
NOTICE:  Parameters:
```

```
NOTICE:  ------------
NOTICE:  No. of Cars = 141, No. of Days = 2, Start day = 2000-01-03
NOTICE:  Optimization = Fastest Path, Disturb data = f
...
```

The generator will take about xx minutes. It will generate trajectories, according to the default parameters, for 141 cars over 2 days Jan 3rd and 4th, 2000.

1.5 Exploring the Generated Data

1.6 Customizing the Generator to Your City

1.7 Tuning the Generator Parameters

1.8 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. 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. 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 services the customers in the order of the list. We assume that the scheduling of the deliveries to clients by the vehicles is done by another system and takes into account the availability of the customers in a time slot of a day and the time needed to service 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);
```

The following procedure generates the data for a number of vehicles and a number of days starting at a given day. The last two arguments correspond to the parameters `P_PATH_MODE` and `P_DISTURB_DATA`.

```
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 servicing a customer
  serviceTime 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 service time in [10, 60] min using a bounded Gaussian distribution
      serviceTime = random_boundedgauss(10, 60) * interval '1 min';
      t = t + serviceTime;
    END LOOP;
  END IF;
END;
$$ LANGUAGE plpgsql STRICT;
```

Figure 1.1 and Figure 1.2 show visualizations of the data generated for the deliveries scenario.

1.9 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

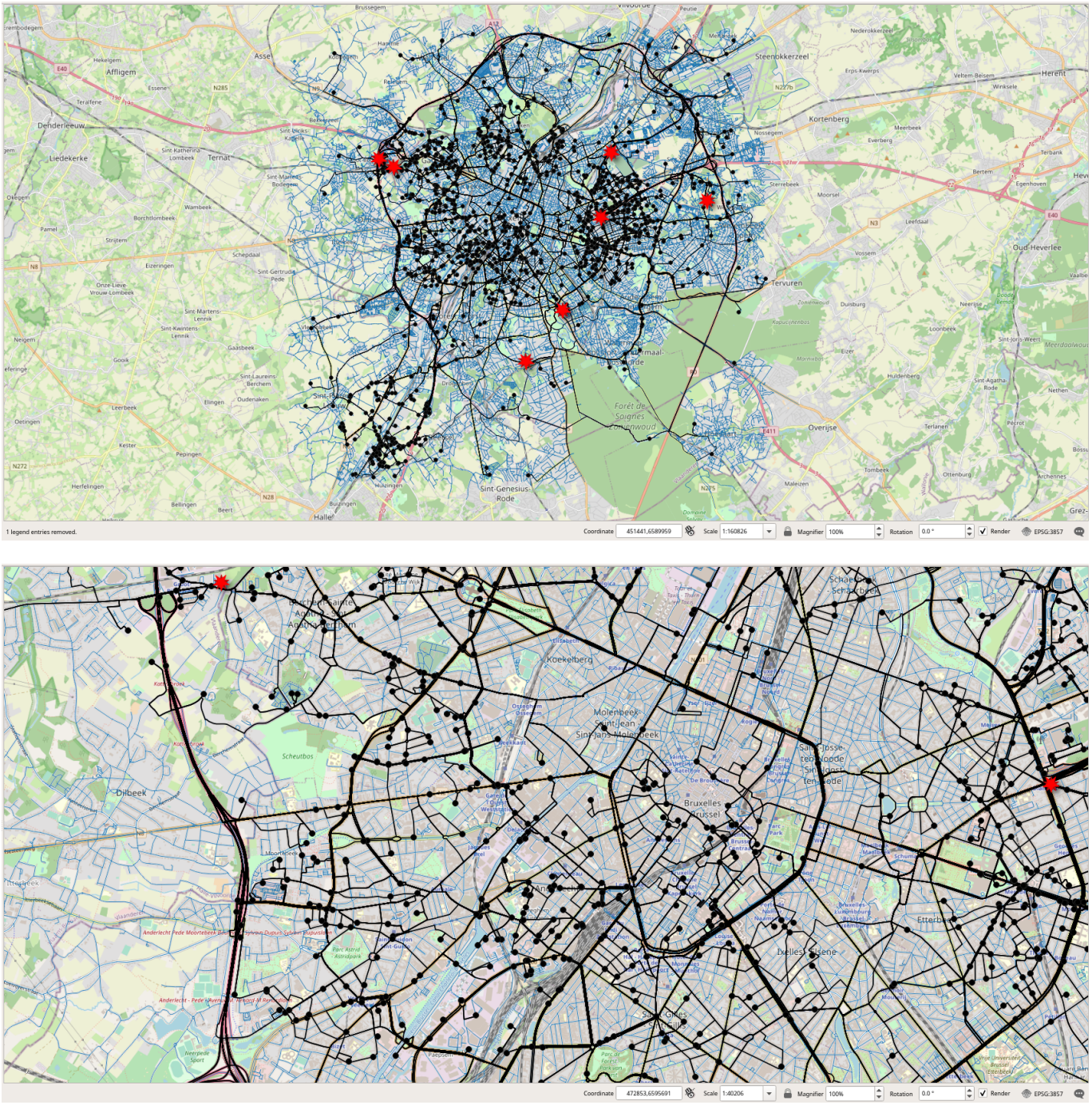

Figure 1.1: Visualization of the data generated for the deliveries scenario. The road network is shown with blue lines, the warehouses are shown with a red star, the routes taken by the deliveries are shown with black lines, and the location of the customers with black points.

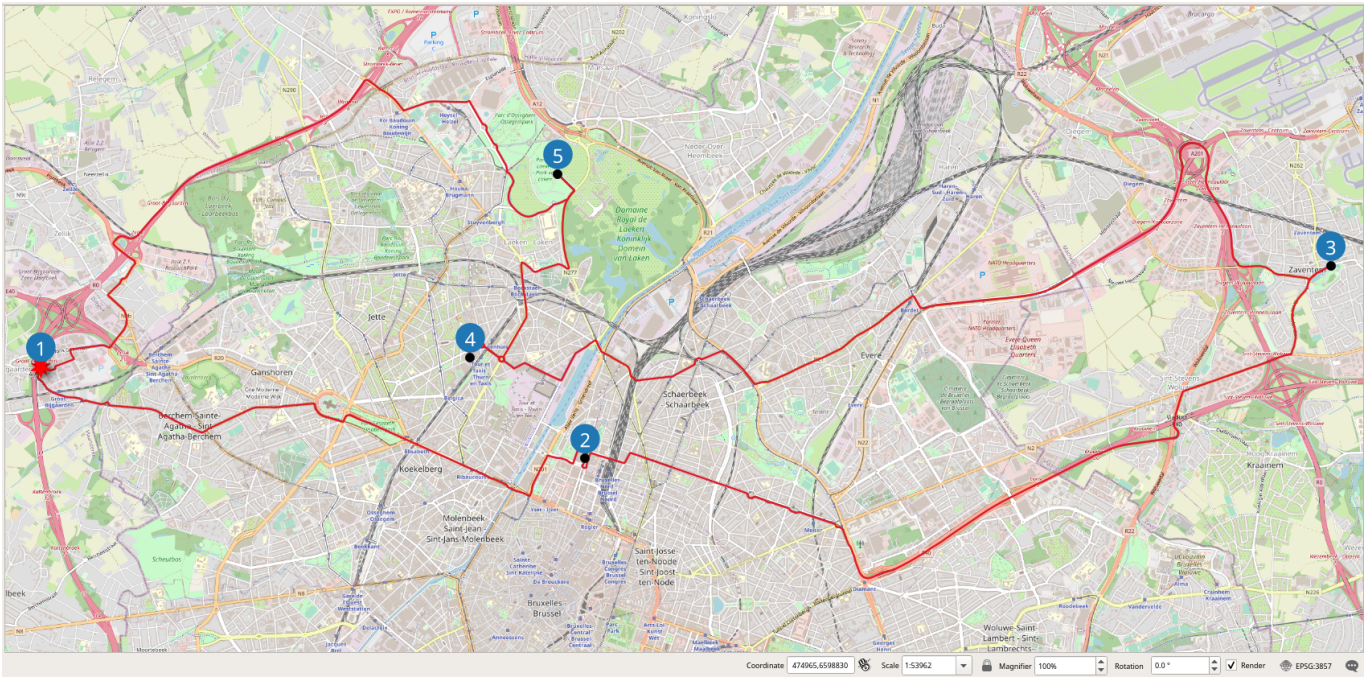

Figure 1.2: Visualization of the deliveries of one vehicle during one day. A delivery trip starts and ends at a warehouse and services several customers, four in this case.

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.

1.9.1 Creating the Graph using SQL

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;
-- SELECT 46234
-- Query returned successfully in 2 secs 857 msec.

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 MyEdges;
CREATE TABLE MyEdges(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 MyEdges(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 MyEdges_id_idx ON MyEdges USING BTREE(id);
CREATE INDEX MyEdges_geom_index ON MyEdges 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 MyEdges 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 MyEdges 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 MyEdges 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 computed implied one way restriction as follows.

```
UPDATE MyEdges 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 MyEdges 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 MyNodes;
CREATE TABLE MyNodes 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 MyEdges')
),
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 MyNodes_id_idx ON MyNodes USING BTREE(id);
CREATE INDEX MyNodes_geom_idx ON MyNodes 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 MyEdges SET source = NULL, target = NULL;

UPDATE MyEdges E SET
  source = N1.id, target = N2.id
FROM MyNodes N1, MyNodes 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 MyEdges 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 MyEdges;
-- 81073
SELECT count(*) FROM Ways_vertices_pgr;
-- 66832
SELECT count(*) FROM MyNodes;
-- 45494
```

As can be seen, we have reduced the size of the graph. This can also be shown in Figure 1.3, 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.

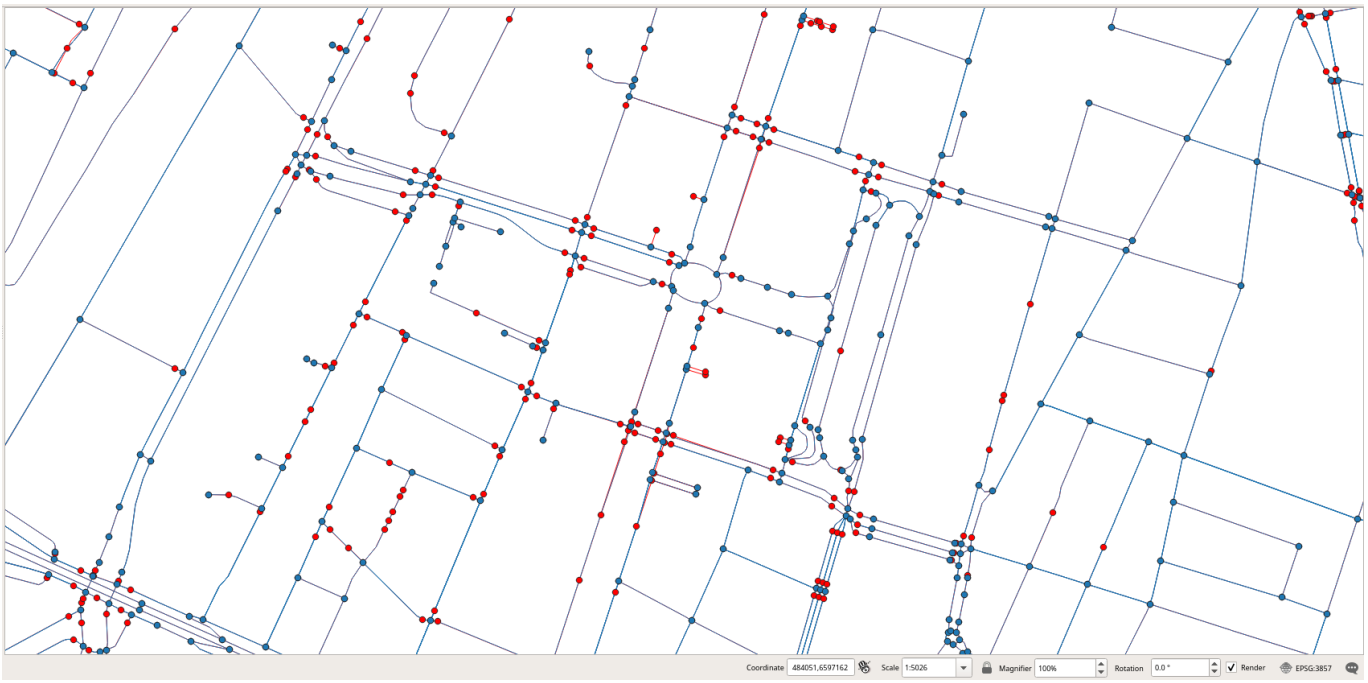

Figure 1.3: Comparison of the nodes obtained (in blue) with those obtained by osm2pgrouting (in red).

1.9.2 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 use the following procedure.

```
CREATE OR REPLACE FUNCTION MergeRoads()
RETURNS void AS $$
DECLARE
  i integer = 1;
  cnt integer;
BEGIN
  -- Create tables
  DROP TABLE IF EXISTS TempRoads;
  CREATE TABLE TempRoads AS
  SELECT *, '{}'::bigint[] AS path
  FROM Roads;
  CREATE INDEX TempRoads_geom_idx ON TempRoads USING GIST(geom);
  DROP TABLE IF EXISTS MergeRoads;
  CREATE TABLE MergeRoads(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 MergeRoads;
    INSERT INTO MergeRoads
    SELECT E1.osm_id AS osm_id1, E2.osm_id AS osm_id2, ST_LineMerge(ST_Union(E1.geom, E2. ↩
        geom)) AS geom
    FROM TempRoads E1, Roads E2
    WHERE E1.osm_id <> E2.osm_id AND E1.highway = E2.highway AND
      E1.oneway = E2.oneway AND ST_Intersects(E1.geom, E2.geom) AND
      ST_EndPoint(E1.geom) =  ST_StartPoint(E2.geom)
      AND NOT EXISTS (
        SELECT * FROM Roads E3 WHERE osm_id NOT IN (
          SELECT osm_id FROM DeletedRoads) AND
        E3.osm_id <> E1.osm_id AND E3.osm_id <> E2.osm_id AND
        ST_Intersects(E3.geom, ST_StartPoint(E2.geom)))
      AND geometryType(ST_LineMerge(ST_Union(E1.geom, E2.geom))) = 'LINESTRING'
      AND NOT St_Equals(ST_LineMerge(ST_Union(E1.geom, E2.geom)), E1.geom);
    -- Exit if there is no more roads to extend
    SELECT count(*) INTO cnt FROM MergeRoads;
    RAISE INFO 'Extended % roads', cnt;
    EXIT WHEN cnt = 0;
    -- Extend the geometries
    UPDATE TempRoads R SET
      geom = M.geom,
      path = R.path || osm_id2
    FROM MergeRoads M
    WHERE R.osm_id = M.osm_id1;
    -- Keep track of redundant roads
    INSERT INTO DeletedRoads
    SELECT osm_id2 FROM MergeRoads
    WHERE osm_id2 NOT IN (SELECT osm_id FROM DeletedRoads);
  END LOOP;
  -- Delete redundant roads
  DELETE FROM TempRoads R USING DeletedRoads M
  WHERE R.id = M.id;
  -- Drop tables
  DROP TABLE MergeRoads;
```

```
  DROP TABLE DeletedRoads;
END; $$
LANGUAGE PLPGSQL;

SELECT MergeRoads();
```

The procedure starts by creating a table `TempRoads` obtained by adding a column `path` to the table `Roads` 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 `MergeRoads` 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 set the `one_way` attribute as we did it previously.

```
ALTER TABLE TempRoads ADD COLUMN one_way integer;
UPDATE TempRoads R
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;

-- Implied one_way
UPDATE TempRoads R
SET one_way = 1
WHERE R.oneway IS NULL AND
  (R.junction = 'roundabout' OR R.highway = 'motorway');
```

We are now ready to create a new set of roads from which we can construct the graph.

```
DROP TABLE IF EXISTS Roads1;
CREATE TABLE Roads1 AS
```

```
SELECT ROW_NUMBER() OVER () AS id, osm_id || path AS osm_id,
  admin_level, bridge, cutting, highway, junction, name, one_way,
  operator, ref, route, surface, toll, tracktype, tunnel, width, geom
FROM TempRoads;

CREATE INDEX Roads1_geom_idx ON Roads1 USING GIST(geom);
```

We then proceed as we did in Section 1.9.1 to compute the set of nodes and the set of edges, now stored into tables `MyNodes1` and `MyEdges1`. In order to compare the two graphs we have obtained and the one obtained by osm2pgrouting we can issue the following queries.

```
SELECT count(*) FROM Ways;
-- 83017
SELECT count(*) FROM MyEdges;
-- 81073
SELECT count(*) FROM MyEdges1;
-- 77986
SELECT count(*) FROM Ways_vertices_pgr;
-- 66832
SELECT count(*) FROM MyNodes;
-- 45494
SELECT count(*) FROM MyNodes1;
-- 42156
```

Figure 1.4 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.

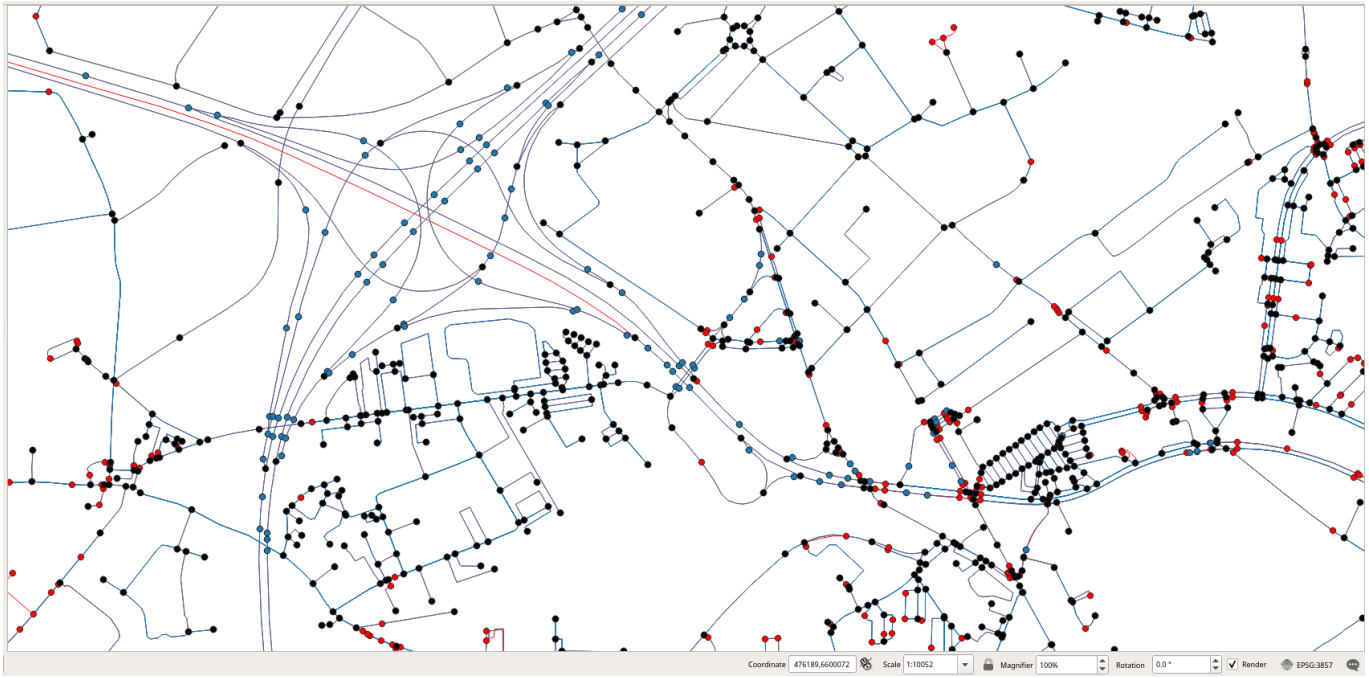

Figure 1.4: Comparison of the nodes obtained by contracting the graph (in black), before contraction (in blue), and those obtained by osm2pgrouting (in red).