MobilityDB-OpenTripPlanner : Workshop

COLLABORATORS			
	TITLE : MobilityDB-OpenTripPlanner : Workshop		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Mehdi Maazouz	September 7, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Abstract

This workshop is about generating trajectories using OpenTripPlanner and connect MobilityDB on those trajectories. This workshop is open source and its code is available on Github The vilualization will be done by Qgis

MobilityDB-OpenTripPlanner is developed by the student Maazouz Mehdi in the framework of his thesis in computer science in the Université Libre de Bruxelles (ULB) under the direction of Prof. Esteban Zimányi.

This is the workshop for MobilityDB-OpenTripPlanner. Feel free to use this material any way you like.

Chapter 1

Introduction

This workshop aims to show you how to connect OpenTripPlanner and MobilityDB and to display the generated trips on Qgis. The generated trips are multi-modal and focus on means of transport obtained via GTFS data.

In this document you will find the tools used for the realization of this workshop as well as the different steps for the creation and visualization of the trips.

1.1 Environment

This workshop was run on a Ubuntu 18.04 LTS Linux machine, 8GB Ram, Intel Core I5-5300U CPU and 512GB SSD. In addition, the following tools made this workshop possible

- Java 11 (OpenTripPlanner 2 requires Java 11 or later)
- PostgreSQL 13.6
- Postgis 2.5.5
- MobilityDB 1.0
- Python 3.7
  - module psycopg2 (installed via PyPi)
- QGis 3.22 Białowieża and its module
  - Move (please follow instructions on github)

Chapter 2

Setting up the project

2.1 Getting the project

Please go to the official github page of the project and download the various files there, or simply clone the project. Here is the command to clone the repository

```
$ git clone https://github.com/MobilityDB/MobilityDB-OpenTripPlanner
```

This project contains several files, here is a brief overview :

- make_doc.sh to generate the worksop's documentation.
- workshop/ contains pregenerated versions of workshop's documentation.
- doc/ contains the necesseray files to generate this workshop.
- src/ contains the sql and python files that will allow the generation of MobilityTrips.
- otpData/ contains the files necessary for the WEB OTP server to function properly. These files are :
  - brussels.pbf corresponds to the brussels.osm file reduced
  - stib-gtfs.zip corresponds to the GTFS data of the Stib covering the months of March 2022 to April 2022
  - tec-gtfs.zip corresponds to the GTFS data of the TEC covering the months of March 2022 to May 2022
  - build-config.json contains options and parameters that are taken into account during the graph building process

2.2 Prepare Database

For the moment, we will use the OSM map of Brussels. The data was fetched via the OpenStreetMap API (Overpass API) and then reduced into a zip file. The result can be found in the root of the project in the file ./brussels.zip. Please extract this file in order to get ./brussels.osm file. Then, open a console

```
$ createdb -h localhost -p 5432 -U dbowner brusselsOTP
-- replace localhost with your database host, 5432 with your port,
-- and dbowner with your database use
```

this command line allows the creation of the brusselsOTP database that we will use throughout this workshop

MobilityDB-OpenTripPlanner is a project that depends on Postgis, MobilityDB as well as Hstore. This is why we need to add these extensions to our database

```
-- create needed extensions
$ psql -h localhost -p 5432 -U dbowner -d brusselsOTP  -c 'CREATE EXTENSION hstore'

$ psql -h localhost -p 5432 -U dbowner -d brusselsOTP  -c 'CREATE EXTENSION postgis'
$ psql -h localhost -p 5432 -U dbowner -d brusselsOTP  -c 'CREATE EXTENSION MobilityDB '
-- adds the PostGIS and the MobilityDB extensions to the database
```

The last two lines can be executed in a single line by using the `CASCADE` parameter which install Postgis extension before installing MobilityDB extension .

```
$ psql -h localhost -p 5432 -U dbowner -d brusselsOTP  -c 'CREATE EXTENSION MobilityDB  ↩
    CASCADE'
```

This last command will allow us to create a topology for the city of Brussels by creating two tables, namely `ways` as well as `ways_vertices_pgr`

```
$ osm2pgrouting -W 'password' -h localhost -p 5432 -U dbowner -f brussels.osm -d  ↩
    brusselsOTP
-- replace password by your dbowner's password
```

the table `ways_vertices_pgr` brings us more than 116000 nodes located in the Brussels area. It is from these points that we will choose our source nodes and target nodes.

2.3 Prepare OpenTripPlanner server

In order to contact the OpenTripPlanner API, we will set up a local web server .

The .jar file `otp-2.1.0-SNAPSHOT-shaded.jar` is located at the root of the project and corresponds to OpenTripPlanner 2.1 (actually, it corresponds to a modified version of OpenTripPlanner 2.1 which offers a better compatibility with GBFS format, see README.md for more information)

When it is executed, it will use the files in otpData/ to create the graph. By default, when the graph is created, the transit network will contain the GTFS data of the Tec and the Stib. If you want to add a GTFS file, please put it in the otpData/ folder and add its name in the build-config.json file like this.

- "gtfs" : "tec-gtfs.zip|stib-gtfs.zip|`my-gtfs.zip`"

For example if we remove the GTFS file tec-gtfs.zip in our build-config.json we've got :

- "gtfs" : "stib-gtfs.zip"

Please be careful, for OpenTripPlanner to detect a GTFS file, its name must end in .zip and must contain the letters. Now all we have to do is launch our server.

```
-- at the root of the project:
$ java -Xmx4G -jar otp-2.0.0-shaded.jar   --build --save ./otpData/
```

The parameter `-Xmx4G` means that you reserve 4GB of memory in order to build the graph containing our OSM and GTFS data from Stib and Tec. If you decide to use only OSM and Stib data, the parameter `-Xmx2G` would be sufficient. The parameter`--build` means the graph is build and `--save` means we store the graph onto disk. To start the server

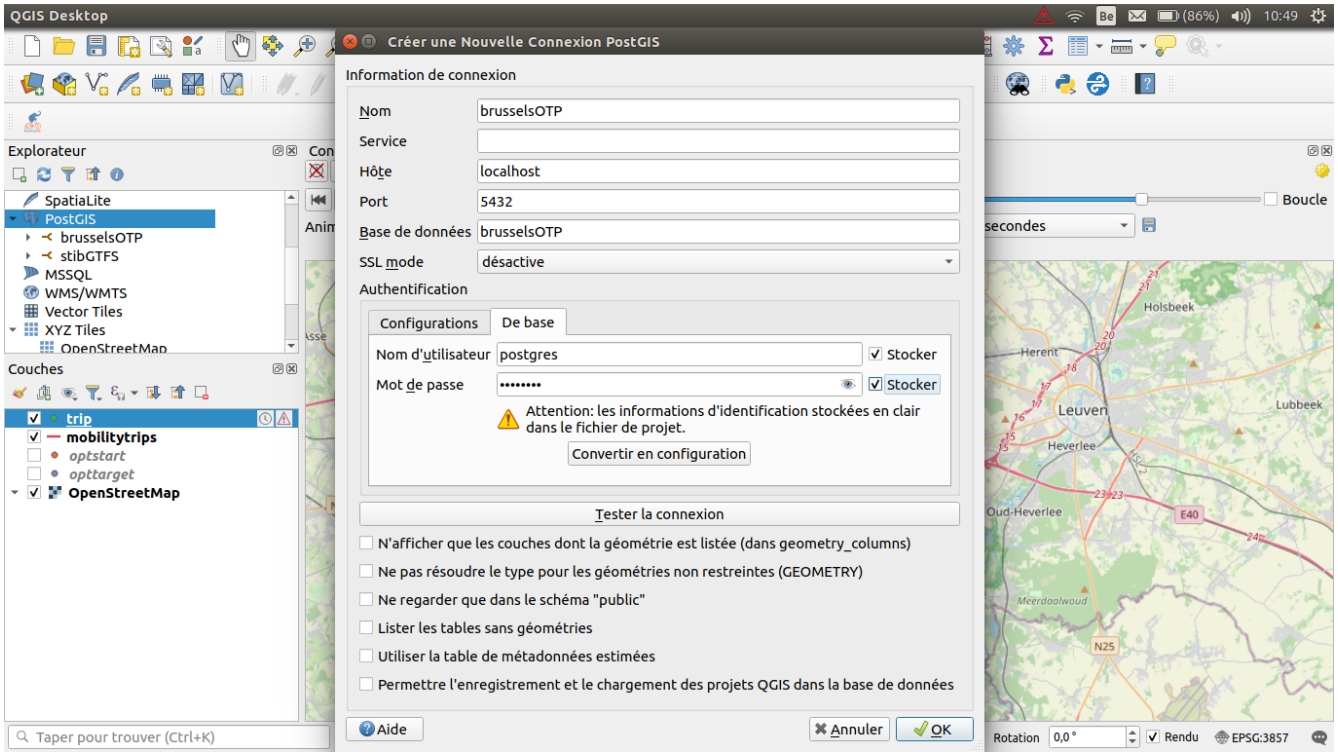
```
-- at the root of the project:
$ java -jar otp-2.0.0-shaded.jar   --load --serve ./otpData/
```

`--load` means we load the graph just built before and `--serve` simply means we run an OTP API server

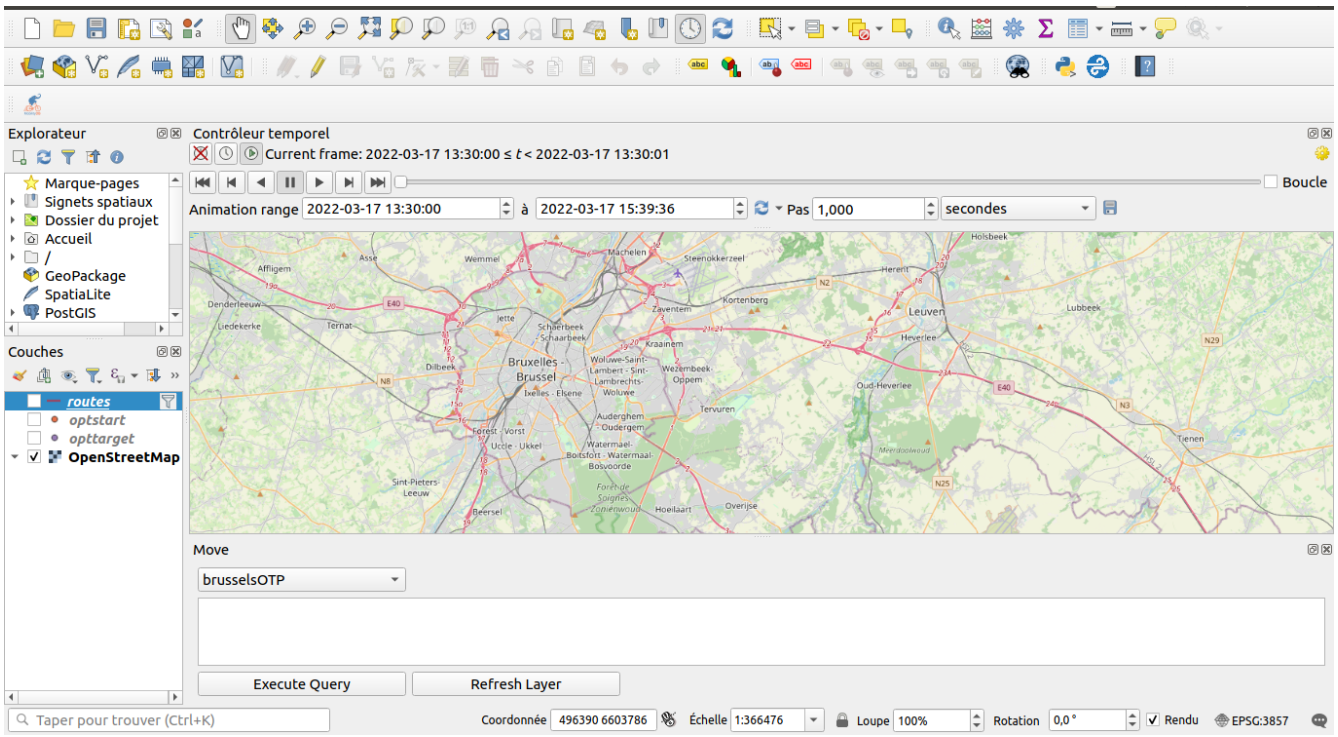
2.4 Prepare QGIS

To display the trips, the Qgis tool is used. Instructions for downloading and installing it can be found at this address

When connecting Qgis with your database, please enable basic authentication. Otherwise, there is a risk that the Move module do not work properly



After installing Qgis and its Move module, you should be able to access to the below window. If you have any problems, please refer to the Move github page



Chapter 3

Generate MobilityDB trips

Once the various tools have been installed and configured. We can start generating MobilityDB trips from trips obtained via OpenTripPlanner.

3.1 A Multi-Modal Trip

First, we will generate a multi-modal trip for a person wanting to travel from a point A to a point B in Brussels. To do this, we will randomly generate a source and a target nodes that will be used in the calculation of the trip.

```
$ psql -h localhost -U postgres -d brusselsOTP  -f ./src/combination.sql -v o=1
-- -v o=1 means we want to generate 1 source node and 1 target node
```

We will now contact the OpenTripPlanner API to generate the trip, and parse it into a PostgreSQL database called `routes`. Please make sure you have started the OTP server before doing this.

```
-- in a console at the root of your project :
$ python ./src/manageOTP.py localhost brusselsOTP dbowner password

-- as usual, replace dbowner by your database user
-- replace password by your database user's password
```

Running this command will prompt you to enter parameters to better define the trip you want to create. Below are some examples of parameters you might want to enter. OpenTripPlanner is full of parameters, a exhaustive list can be found at this address

the desired format is presented as follows : `key=value`

Here is an example you could enter in the console when you run `manageOTP.py`

```
please enter the desired parameters ( key=value ) : arriveBy=true optimize=QUICK
```

You could found below some parameters you put enter :

- date=2022-03-10
- time=13:23
- mode=WALK,BIKE (represents transport modes to consider)
- wheelchair=true
- arriveBy=true (specifies that the given time is when we plan to arrive)
- maxWalkDistance=1000 (specifies the maximum distance in meters that you are willing to walk)

If you wish, you can press "Enter" directly. In this way, the default values will be taken into account. Note that the default `date` represents the current day, the default `time` represents the current time.

Please note that the baseline GTFS data covers the months of March 2022. If you are doing this workshop later, please enter a specific date that is taken into account by the GTFS data

For the sake of this workshop, we have changed the default value of the `mode`parameter. Originally `WALK`, it is now `WALK,TRANSIT`. The reason being that this workshop deals with multi-modal routing, so there is no point in using only your feet to do the trips. Moreover, we changed the default value `numItineraries=1` by `numItineraries=2` mainly because the default value only takes account the WALK trip.

In our case, when the prompt below appears, we click immediately on Enter

```
please enter the desired parameters ( key=value ) :
```

Now we just need to run one last sql script to generate the mobility trip.

```
-- in a console at the root of your project
$ psql -h localhost -U dbowner -d brusselsOTP  -f ./src/generateMobility_Trips.sql -v  ↩
    itineraries=false
-- We generate MobilityDB trips
```

We now have the mobilitytrips table containing our tgeompoints. As well as a table `stibtrip` containing our tgeompoints which represents the part by public transport, a table `waittrip` which represents the waiting for a transfer and a table `walktrip` which represents walking part.

Now let's go to Qgis and open a connection to our BrusselsOTP Database. Once done, click on the `optstart` and `opttarget` tables to display the source node and taget node. The generated nodes are random, so there is a good chance that you will have other nodes

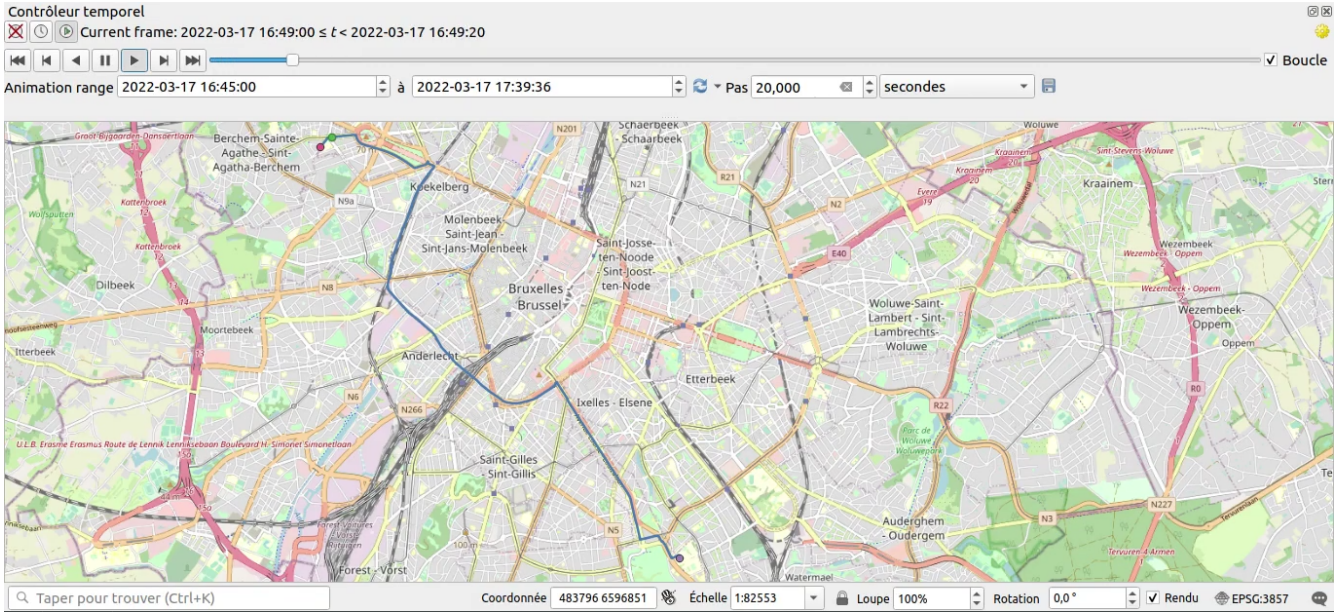
We can display our `mobilitytrips` table on QGis. However, in order to be able to visualize our points (people) moving in time, we will use the Move module.

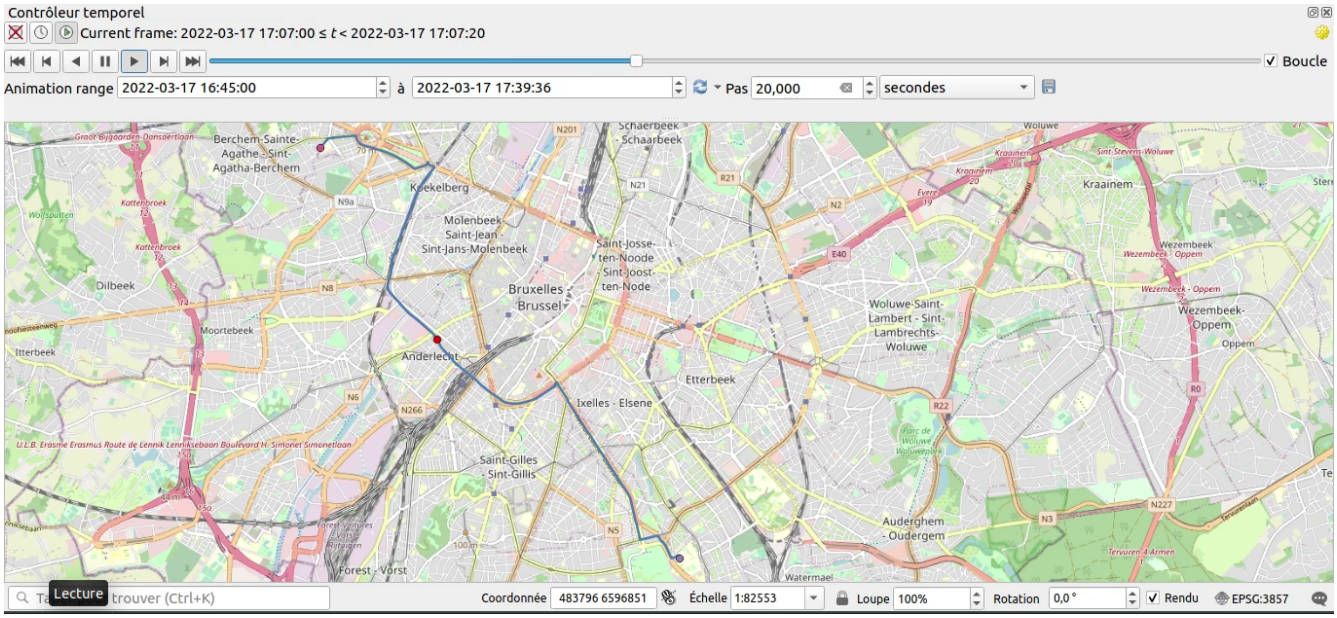
In order to visualize the points in a temporal way, we just need to use the Move plugin (installed previously). Click on the plugin to display a window at the bottom where you insert the following queries and then, click on execute. This will create temporal layers. Open the controller panel (View->Panels->Temporal Controller Panel) and you can then see the moving points move over time.

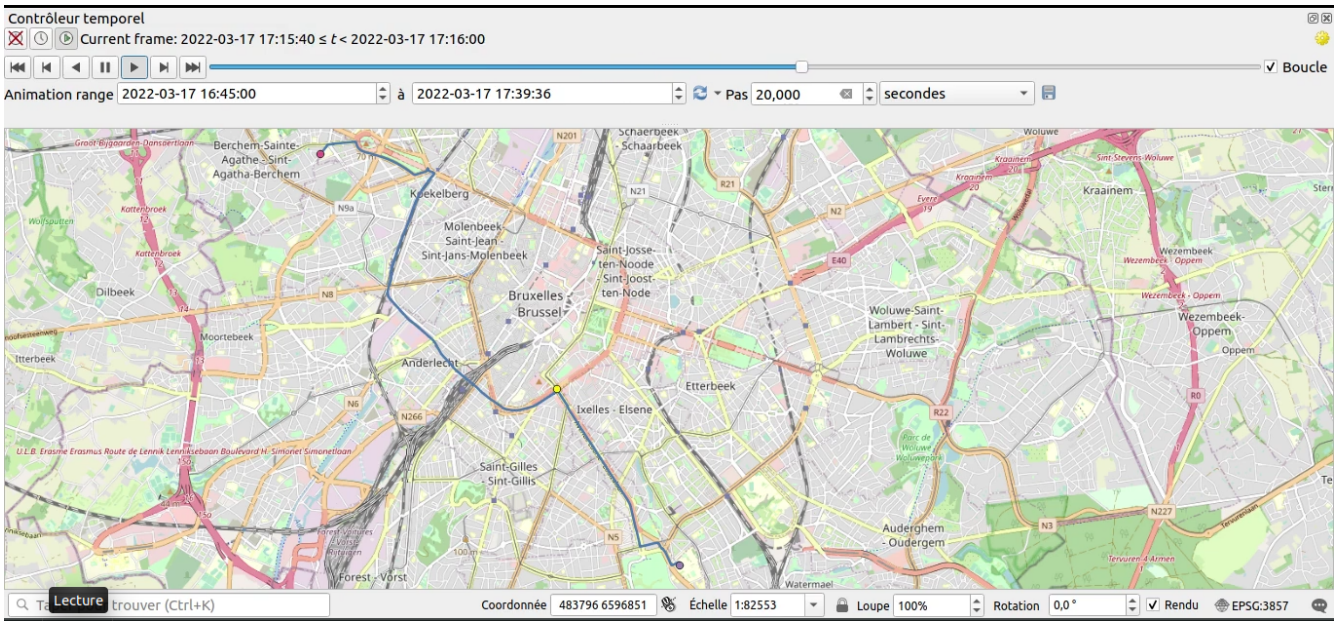
```
-- on the plugin Move in Qgis
select trip as walktrip from walktrip
select trip as waittrip from waittrip
select trip as stibtrip from stibtrip
```

Please note that in Belgium we are in UTC+1. I'm currently having trouble displaying the correct time on Move. Therefore, if you generate trips between 04:00 pm and 04:30 pm for example, the display of the points on Move will be between 03:00 pm and 03h30 pm.

You should have a similar result, the green dot representing the person walking, the red dot representing the person moving in a public transport (tram, bus, metro) and the yellow dot representing the person waiting for a transfer


Contrôleur temporel
Current frame: 2022-03-17 16:49:00 ≤ t < 2022-03-17 16:49:20
Boucle
Animation range 2022-03-17 16:45:00 à 2022-03-17 17:39:36 Pas 20,000 secondes
Berchem-Sainte-Agathe - Sint-Agatha-Berchem
Koekelberg
Molenbeek-Saint-Jean - Sint-Jans-Molenbeek
Schaerbeek - Schaarbeek
Kraainem
Wezembeek-Oppem
Saint-Josse-ten-Noode - Sint-Joost-ten-Node
Bruxelles - Brussel
Woluwe-Saint-Lambert - Sint-Lambrechts-Woluwe
Dilbeek
Moortebeek
Itterbeek
Anderlecht
Etterbeek
Ixelles - Elsene
Saint-Gilles - Sint-Gillis
Auderghem - Oudergem
Forest - Vorst
Watermael
Taper pour trouver (Ctrl+K)
Coordonnée 483796 6596851
Échelle 1:82553
Loupe 100%
Rotation 0,0 °
Rendu
EPSG:3857


Contrôleur temporel
Current frame: 2022-03-17 17:07:00 ≤ t < 2022-03-17 17:07:20
Boucle
Animation range 2022-03-17 16:45:00 à 2022-03-17 17:39:36 Pas 20,000 secondes
Berchem-Sainte-Agathe - Sint-Agatha-Berchem
Koekelberg
Molenbeek-Saint-Jean - Sint-Jans-Molenbeek
Schaerbeek - Schaarbeek
Kraainem
Wezembeek-Oppem
Saint-Josse-ten-Noode - Sint-Joost-ten-Node
Bruxelles - Brussel
Woluwe-Saint-Lambert - Sint-Lambrechts-Woluwe
Dilbeek
Moortebeek
Itterbeek
Anderlecht
Etterbeek
Ixelles - Elsene
Saint-Gilles - Sint-Gillis
Auderghem - Oudergem
Forest - Vorst
Watermael
Lecture
trouver (Ctrl+K)
Coordonnée 483796 6596851
Échelle 1:82553
Loupe 100%
Rotation 0,0 °
Rendu
EPSG:3857



3.2 People moving in Brussels

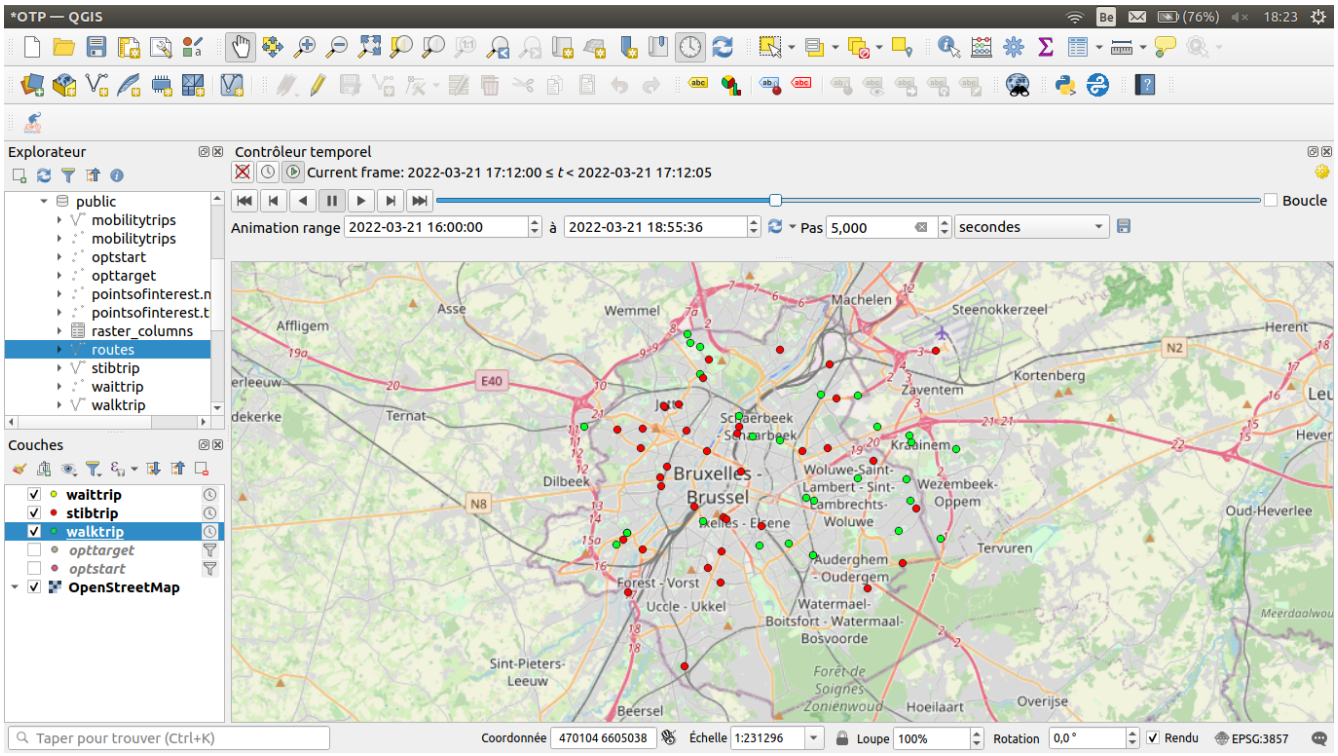
We will now generate multi-modal trips for a multitude of people. The principle is the same as before, except for the number of sources and target nodes generated.

```
$ psql -h localhost -U postgres -d brusselsOTP  -f ./src/combination.sql -v o=40
-- -v o=40 means we want to generate 40 source nodes and 40 target nodes
$ python ./src/manageOTP.py localhost brusselsOTP dbowner password

please enter the desired parameters ( key=value ) : ( Click on Enter )

$ psql -h localhost -U dbowner -d brusselsOTP -f ./src/generateMobility_Trips.sql -v  ↩
    itineraries=false
```

Inside Move module in Qgis

```
-- on the plugin Move in Qgis
select trip as walktrip from walktrip
select trip as waittrip from waittrip
select trip as stibtrip from stibtrip
```

Here is the result



3.3 Comparison with alternatives itineraries

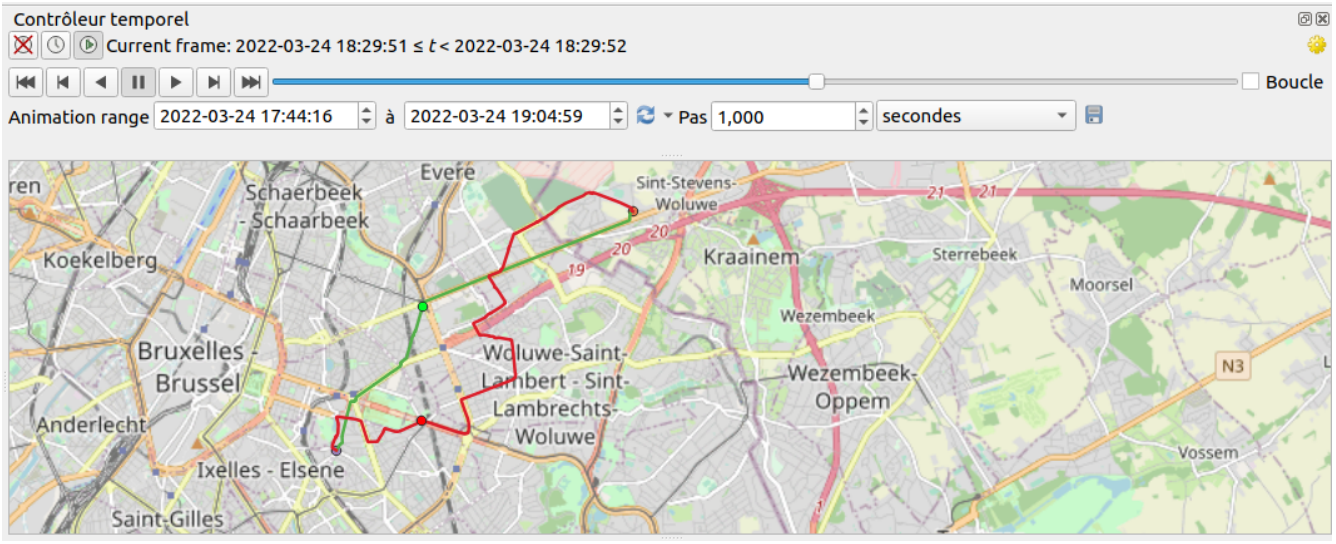
You can also generate multiple itineraries for a single trip. In this workshop, we will take a simple example by comparing only 2 itineraries proposed to one person.

```
$ psql -h localhost -U postgres -d brusselsOTP  -f ./src/combination.sql -v o=1
-- -v o=1 means we want to generate 1 source nodes and 1 target nodes
$ python ./src/manageOTP.py localhost brusselsOTP dbowner password

please enter the desired parameters ( key=value ) : ( Click on Enter )

$ psql -h localhost -U dbowner -d brusselsOTP -f ./src/generateMobility_Trips.sql -v  ↩
    itineraries=true
```

As we said before, the variable `numItineraries` is already set to 2, so you don't have to change this value. However, in order to display those alternatives itineraries, you have to set `itineraries=true` the command parameter `-v` when you run `generateMobility_Trips.sql`

Obviously, you could generate smore than 2 itineraries for one trip by giving the value you want to the parameter `numItineraries`

Below you see the red line representing a multi-modal trip (the red dot representing a person taking a public transport) while the green line represents the trip by foot. Then, you see that the multi-modal trip is faster than the other one.


Contrôleur temporel
Current frame: 2022-03-24 18:29:51 ≤ t < 2022-03-24 18:29:52
Boucle
Animation range 2022-03-24 17:44:16 à 2022-03-24 19:04:59 Pas 1,000 secondes
Schaerbeek - Schaarbeek
Evere
Sint-Stevens-Woluwe
Koekelberg
Kraainem
Sterrebeek
Moorsel
Wezembeek
Bruxelles - Brussel
Woluwe-Saint-Lambert - Sint-Lambrechts-Woluwe
Wezembeek-Oppem
N3
Anderlecht
Ixelles - Elsene
Vossem
Saint-Gilles

Chapter 4

Understanding the generation process

We now describe the different codes used to generate our different scenarios. We start with the file `combination.sql`

```
DROP FUNCTION IF EXISTS generateCombinations;
CREATE OR REPLACE FUNCTION generateCombinations(noCombinations int)
RETURNS void AS $$
DECLARE
  noVertices int;
  startVertice int;
  targetVertice int;
  allstartVertices int[];
  alltargetVertices int[];

BEGIN
  DROP TABLE IF EXISTS optstart;
  DROP TABLE IF EXISTS opttarget;
  CREATE TABLE optstart (id int,cnt int,chk int,ein int,eout int,the_geom geometry);
  CREATE TABLE opttarget (id int,cnt int,chk int,ein int,eout int,the_geom geometry);

  select count(*) from ways_vertices_pgr into noVertices;
  FOR i in 1..noCombinations LOOP
     select ceiling(random()*noVertices) into startVertice;
     INSERT INTO optstart ( select id,cnt,chk,ein,eout,the_geom from ways_vertices_pgr ↩
         where id = startVertice );

     select ceiling(random()*noVertices) into targetVertice;
     INSERT INTO opttarget ( select id,cnt,chk,ein,eout,the_geom from ways_vertices_pgr ↩
         where id = targetVertice );

     UPDATE opttarget set ein = startVertice where id = targetVertice;
  END LOOP;
  RETURN;
END;
$$ LANGUAGE plpgsql STRICT;

Select generateCombinations(:o);
```

In this file we loop over the number of trips we want to generate .

We chose the source nodes and target nodes with a uniform distribution among all nodes in the table `ways_vertices_pgr`. We modify the `ein` column of the `opttarget` table to match each target node with its source node.

The `manageOTP.py`file will contact the OpenTripPlanner API and parse the result into an sql table that we can then process

```
conn = psycopg2.connect("host="+str(sys.argv[1])+" dbname="+str(sys.argv[2])+" user="+str( ↩
    sys.argv[3])+" password="+sys.argv[4])
cur = conn.cursor()

match_item = matchNodes()
sql = "DROP TABLE IF EXISTS ROUTES;"
cur.execute(sql)
sql = "CREATE TABLE ROUTES (route_legid int,route_routeid int,route_from_starttime  ↩
    timestamp, route_leg_starttime timestamp, route_leg_distance double precision,  ↩
    route_leg_endtime timestamp, route_leg_from_lat double precision, route_leg_from_lon  ↩
    double precision, source_id bigint, target_id bigint,route_leg_mode text, geom geometry) ↩
    ;"
cur.execute(sql)

route_routeid = 0
route_legid = 0
parameters = input("please enter the desired parameters ( key=value ) : ")
d_parameters = setParameters(parameters)


for source, target in match_item :
  source_id = source
  cur.execute('SELECT st_astext(the_geom) from optstart where id='+str(source)+';')
  response = cur.fetchall()
  coordinate_start = getCoordinate(response)

  target_id = target
  cur.execute('SELECT st_astext(the_geom) from opttarget where id='+str(target)+';')
  response = cur.fetchall()
  coordinate_target = getCoordinate(response)

  coordinate_start=coordinate_start[0].split(' ')
  coordinate_target=coordinate_target[0].split(' ')
  route_url = 'http://localhost:8080/otp/routers/default/plan?fromPlace='+coordinate_start ↩
      [1]+','+coordinate_start[0]+'&toPlace='+coordinate_target[1]+','+coordinate_target[0]
  for key in d_parameters :
    route_url = route_url +'&'+key+'='+d_parameters[key]

  route_headers = {"accept":"application/json"}
  route_request = urllib.request.Request(route_url, headers=route_headers)
  route_response = urllib.request.urlopen(route_request)
  response_data = route_response.read()
  encoding = route_response.info().get_content_charset('utf-8')
  route_data = json.loads(response_data.decode(encoding))

  route_from_lat = route_data['plan']['from']['lat']
  route_from_lon = route_data['plan']['from']['lon']

  try :
    route_from_name = route_data['plan']['from']['name']
  except :
    route_from_name = None # not available

  try :
    route_errormessage = route_data['error']['message']
    if route_errormessage == 'LOCATION_NOT_ACCESSIBLE' :
      cur.execute('DELETE from optstart where id='+str(source)+';')
      cur.execute('DELETE from opttarget where id='+str(target)+';')
      continue
  except :
    pass
  route_to_lat = route_data['plan']['to']['lat']
```

```
  route_to_lon = route_data['plan']['to']['lon']
  for iter in route_data['plan']['itineraries']:
    route_routeid += 1
    route_from_starttime = iter['startTime']
    route_to_endtime = iter['endTime']
    route_total_duration = iter['duration']
    route_total_transittime = iter['transitTime']
    route_total_waitingtime = iter['waitingTime']
    route_total_walktime = iter['walkTime']
    route_total_walkdistance = iter['walkDistance']
    route_total_transfers = iter['transfers']
    route_leg_totaldistcounter = 0


    for leg in iter['legs']:
      route_legid += 1
      route_leg_starttime = leg['startTime']
      route_leg_departuredelay = leg['departureDelay']
      route_leg_endtime = leg['endTime']
      route_leg_arrivaldelay = leg['arrivalDelay']
      route_leg_duration = leg['duration']
      route_leg_distance = leg['distance']
      route_leg_mode = leg['mode']
      route_leg_from_lat = leg['from']['lat']
      route_leg_from_lon = leg['from']['lon']
      route_leg_from_name = leg['from']['name']
      route_leg_from_departure = leg['from']['departure']
      route_leg_to_lat = leg['to']['lat']
      route_leg_to_lon = leg['to']['lon']
      route_leg_to_name = leg['to']['name']
      route_leg_to_arrival = leg['to']['arrival']
      route_leg_encodedpolylinestring = leg['legGeometry']['points']
      route_leg_decodedpolylinestring_aspointlist = decode_polyline( ↩
          route_leg_encodedpolylinestring)

      sql1 = "ST_Setsrid(ST_Makeline(ARRAY["
      for coordinate in route_leg_decodedpolylinestring_aspointlist :
        sql1 = sql1 + "ST_Point("+str(coordinate[0])+","+str(coordinate[1])+"),"
      sql1 = sql1[:-1]
      sql1 = sql1 +']),4326)'


      sql = "INSERT INTO ROUTES VALUES("+str(route_legid)+","+str(route_routeid)+", ↩
          TIMESTAMP '"+str(datetime.datetime.fromtimestamp(route_from_starttime/1000))+"', ↩
          TIMESTAMP '"+str(datetime.datetime.fromtimestamp(route_leg_starttime/1000))+"',"+ ↩
          str(route_leg_distance)+",TIMESTAMP '"+str(datetime.datetime.fromtimestamp( ↩
          route_leg_endtime/1000))+"',"+str(route_leg_from_lat)+","+str(route_leg_from_lon) ↩
          +","+str(source_id)+","+str(target_id)+", '"+route_leg_mode+"',"+str(sql1)+");"
      cur.execute(sql)

conn.commit()
conn.close()
```

We start by creating a `routes` table which will contain all our trips. Then, for each route, we will call the OpenTripPlanner API which will answer us in json format. Then we parse the result. Before creating the sql query that we will insert in our `routes` table, we call a function `decode_polyline`

```
# Source: https://stackoverflow.com/a/33557535/8947209 (slightly modified)
def decode_polyline(polyline_str):
  index, lat, lng = 0, 0, 0
  #coordinates = []
  pointlist = []
```

```
  changes = {'latitude': 0, 'longitude': 0}

  # Coordinates have variable length when encoded, so just keep
  # track of whether we've hit the end of the string. In each
  # while loop iteration, a single coordinate is decoded.
  while index < len(polyline_str):
    # Gather lat/lon changes, store them in a dictionary to apply them later
    for unit in ['latitude', 'longitude']:
      shift, result = 0, 0
      while True:
        byte = ord(polyline_str[index]) - 63
        index+=1
        result |= (byte & 0x1f) <  < shift
        shift += 5
        if not byte >= 0x20:
          break

      if (result & 1):
        changes[unit] = ~(result >> 1)
      else:
        changes[unit] = (result >> 1)

    lat += changes['latitude']
    lng += changes['longitude']

    qgspointgeom = (float(lng / 100000.0),float(lat / 100000.0))
    pointlist.append(qgspointgeom)
  return pointlist
```

The routes given by OpenTripPlanner in the form of a polyline. Indeed, OpenTripPlanner uses a loosy compression algorithm that encodes a set of coordinates into a lighter string. The encoding process will convert bits into a series of ASCII characters. To do this, each bit is translated into a numeric value, then summed with 63 to obtain an ASCII character that will be displayed cleanly. More specifically, here are the different steps required to encode a value into a polyline with an exemple.

- We have got a initial signed value
  - **-179.9832104**
- Take this value, multiply it by 1e5 and round the result
  - **-17998321**
- Convert the value to binary (note that we use two's complement technique)
  - **.... .... 11111110 11101101 01011110 00001111**
- We take each 32-bit and make a left-shift of one bit
  - **11111101 11011010 10111100 00011110**
- If the original decimal value is negative, invert this encoding
  - **00000010 00100101 01000011 11100001**
- We break the value out into 5-bit chunks (starting from the right hand side)
  - **00001 00010 01010 10000 11111 00001**
- Place the 5-bit chunks into reverse order
  - **00001 11111 10000 01010 00010 00001**
- OR each value with 0x20 if another bit chunk follows

  - **100001 111111 110000 101010 100010 000001**
- Convert each value to decimal
  - **33 63 48 42 34 1**
- Add 63 to each value
  - **96 126 111 105 97 64**
- Finally convert each value to its ASCII equivalent:
  - **`~oia@**

This procedure describes the encoding. Please note that the example and a more complete definition can be found on this page.

Our procedure `decode_polyline` has to be an inverse function of `encode_polyline` in order to get coordinates value. That's the reason why we use `ord` to get the integer representing the unicode character, then we reduce it by removing 63

We decode each character, whenever we get a latitude or longitude, we break. Indeed, the step `OR each value with 0x20 if another bit chunk follow` at the encode process does not apply on the last byte. As a consequence, this byte is lesser than 0x20. Then if necessary we reverse the byte, corresponding to the step `If the original decimal value is negative, invert this encoding` and we make a right-shift of one bit, corresponding to the step `We take each 32-bit and make a left-shift of one bit`. Finally we divide by a constant number our latitude and longitude and as a result we obtain our coordinate.

Note that the step `Place the 5-bit chunks into reverse order` is done in the nested loop by the line `result |= (byte & 0x1f) << shift`

The `setParameters` function encodes the various parameters that will be used when calling the API. Some parameters have a default value if not specified by the user

```
def setParameters(parameters) :
  ''' create paramaters which be inserted into the API call
    May introduce default paramters if not specified by the user '''
  if len(parameters) == 0 :
    tempo_parameters = []
  else :
    tempo_parameters = parameters.split(' ')
  res = {}
  for i in range (0,len(tempo_parameters)) :
    res[tempo_parameters[i].split('=')[0]]=tempo_parameters[i].split('=')[1]
  if 'date' not in res :
    res['date']= str(datetime.date.today())
  if 'time' not in res :
    now = datetime.datetime.now()
    res['time'] = now.strftime("%H:%M:%S")
  if 'mode' not in res:
    res['mode']='WALK,TRANSIT'
  if 'numItineraries' not in res :
    res['numItineraries'] = '2'
  return res
```

In order to generate our Mobility trips, we run the `generateMobility_trips.sql`

```
 DROP TYPE IF EXISTS step CASCADE;
CREATE TYPE step as (geom geometry,route_leg_from_lat double precision, route_leg_from_lon ↩
    double precision,
route_leg_starttime timestamp,route_leg_endtime timestamp,route_leg_distance float, ↩
    route_leg_mode text);
```

We create a type step which corresponds to a trip's edge.

Specifically it contains the geometry, the coordinate, the time of when the user uses this edge and the time of when he leaves the edge, the distance of this segment and the mode that corresponds to the modality.

```
DROP FUNCTION IF EXISTS createTrips;
CREATE OR REPLACE FUNCTION createTrips(itineraries bool)
RETURNS void AS $$
DECLARE
  trip tgeompoint;
  notrips int;
  id int;
  d date;
  tmode text;
  actual_source int;
  next_source int;
  maxleg int;
  nolegs int;
  baseleg int;
  nodeSource bigint;
  nodeTarget bigint;
  leg_starttime timestamptz;
  leg_endtime timestamptz;
  leg_geom geometry;
  changed_itinerary bool;
  path step[];
BEGIN
  id = 1;
  DROP TABLE IF EXISTS MobilityTrips CASCADE;
  CREATE TABLE MobilityTrips(id int, routeid int, day date, source bigint,
    target bigint, transport_mode text, trip tgeompoint, trajectory geometry,
    PRIMARY KEY (id));

  select max(route_routeid) from routes into notrips;
  select max(route_legid) from routes into maxleg;

  For actualtrip in 1..notrips LOOP


    IF itineraries != true THEN
      -- Verification in order to delete first itinary composed only of walk
      select source_id from routes into actual_source where actualtrip=route_routeid;
      IF actualtrip != notrips THEN
        select source_id from routes into next_source where actualtrip+1=route_routeid;
        continue when actual_source=next_source;
      END IF;
    END IF;

    select min(route_legid) from routes into baseleg where route_routeid=actualtrip;
    changed_itinerary = true;
    select count(route_routeid) from routes where route_routeid=actualtrip into nolegs;

    For j in 0..nolegs-1 LOOP
      IF baseleg+j != 1 and baseleg+j != maxleg and changed_itinerary != true THEN
        select route_leg_starttime from routes into leg_starttime where baseleg+j =  ↩
            route_legid;
        select route_leg_endtime from routes into leg_endtime where baseleg+j-1 =  ↩
            route_legid;
        select geom from routes into leg_geom where baseleg+j-1 = route_legid;

        select source_id from routes into nodeSource where baseleg+j = route_legid;
        select target_id from routes into nodeTarget where baseleg+j = route_legid;

        -- We check if the user has to wait for a transfer
        If leg_starttime != leg_endtime THEN
          trip = wait(leg_endtime,leg_starttime,leg_geom);
```

```
          INSERT INTO MobilityTrips VALUES (id, actualtrip, d, nodeSource, nodeTarget, ' ↩
              WAIT', trip, trajectory(trip));
          id = id+1;
        END IF;
      END IF;

      changed_itinerary = false;
      SELECT array_agg((geom,route_leg_from_lat,
        route_leg_from_lon,route_leg_starttime,route_leg_endtime,route_leg_distance, ↩
            route_leg_mode)::step)
        INTO path FROM routes where baseleg+j = route_legid;

      select route_from_starttime from routes into d where baseleg+j = route_legid;
      select source_id from routes into nodeSource where baseleg+j = route_legid;
      select target_id from routes into nodeTarget where baseleg+j = route_legid;
      select route_leg_mode from routes into tmode where baseleg+j = route_legid;

      trip = generatetrip(path);
      INSERT INTO MobilityTrips VALUES (id, actualtrip,d, nodeSource, nodeTarget, tmode,  ↩
          trip, trajectory(trip));
      id = id+1;
    END LOOP;
  END LOOP;
  RETURN;
END;
$$ LANGUAGE plpgsql STRICT;
```

Then the procedure `createTrips` is called for creating Mobility trips. We loop over the number of trips we have generated. Firstly, if the parameter `itineraries` equals false we do not take into account each route composed exclusively of walking if and only if there is a multi-modal alternative.

Then we loop over the total number of itineraries we have. We check if the user has to wait for a transfer and call the procedure `wait` if so. Otherwise we call the procedure `generatetrip` for creating our Mobility trips.

```
 DROP FUNCTION IF EXISTS wait;
CREATE OR REPLACE FUNCTION wait(starttime timestamptz, endtime timestamptz, trip geometry)
RETURNS tgeompoint AS $$
DECLARE
  p1 geometry;
  curtime timestamptz;
  instants tgeompoint[];
  l int;
BEGIN

  p1 = ST_PointN(trip, -1);
  curtime = starttime;
  endtime = endtime - 1000 * interval '1 ms';
  l=1;
  WHILE (curtime < endtime) LOOP
    curtime = curtime + 100 * interval '1 ms';
    instants[l] = tgeompoint_inst(p1, curtime);
    l = l + 1;
  END LOOP;

  RETURN tgeompoint_seq(instants, true, true, true);
END;
$$ LANGUAGE plpgsql STRICT;
```

The procedure recieves as argument the timestamp at which the user has to start waiting, the timestamp at which the user ends waiting and the trip. The output is a sequence of temporal geometry points staying at the same coordinate, waiting for a transfer

```
CREATE OR REPLACE FUNCTION generateTrip(trip step[])
```

```
RETURNS tgeompoint AS $$
DECLARE
  srid int;
  instants tgeompoint[];
  curtime timestamptz;
  departureTime timestamptz;
  linestring geometry;
  latitude_from double precision;
  longitude_from double precision;
  p1 geometry;
  p2 geometry;
  points geometry [];
  noEdges int;
  noSegs int;
  speed float; x1 float; x2 float;y1 float; y2 float;
  curDist double precision;traveltime double precision;
  l int;
  notrips int;
BEGIN

  p1 = ST_PointN((trip[1]).geom, 1);
  curtime = (trip[1]).route_leg_starttime;
  instants[1] = tgeompoint_inst(p1, curtime);
  l=2;
  noEdges = array_length(trip, 1);
  FOR i IN 1..noEdges LOOP
    linestring = (trip[i]).geom;
    SELECT array_agg(geom ORDER BY path) INTO points FROM ST_DumpPoints(linestring);

    noSegs = array_length(points, 1) - 1;
    speed = (trip[i]).route_leg_distance / EXTRACT(EPOCH from (trip[i]).route_leg_endtime-( ↩
        trip[i]).route_leg_starttime);

    FOR j IN 1..noSegs LOOP
      p2 = ST_setSRID(points[j + 1],4326);
      x2 = ST_X(p2);
      y2 = ST_Y(p2);

      curDist = ST_Distance(p1::geography,p2::geography);
      IF curDist = 0 THEN
        curDist = 0.1;
      END IF;

      travelTime = (curDist / speed);
      curtime = curtime + travelTime * interval '1 second';

      p1 = p2;
      x1 = x2;
      y1 = y2;

      instants[l] = tgeompoint_inst(p1, curtime);
      l = l + 1;
    END LOOP;
  END LOOP;
  RETURN tgeompoint_seq(instants, true, true, true);
END;
$$ LANGUAGE plpgsql STRICT;
```

The procedure recieves as arugment an array of step. The procedure loops for each edge and determines the number of segments of the edge. A segment is composed of a line and two points at its extremities. Then for each segment we compute the speed, the distance and the time needed to reach the extremitie of the segment. The output is a sequence of temporal points following the

trip.

Chapter 5

Benchmark

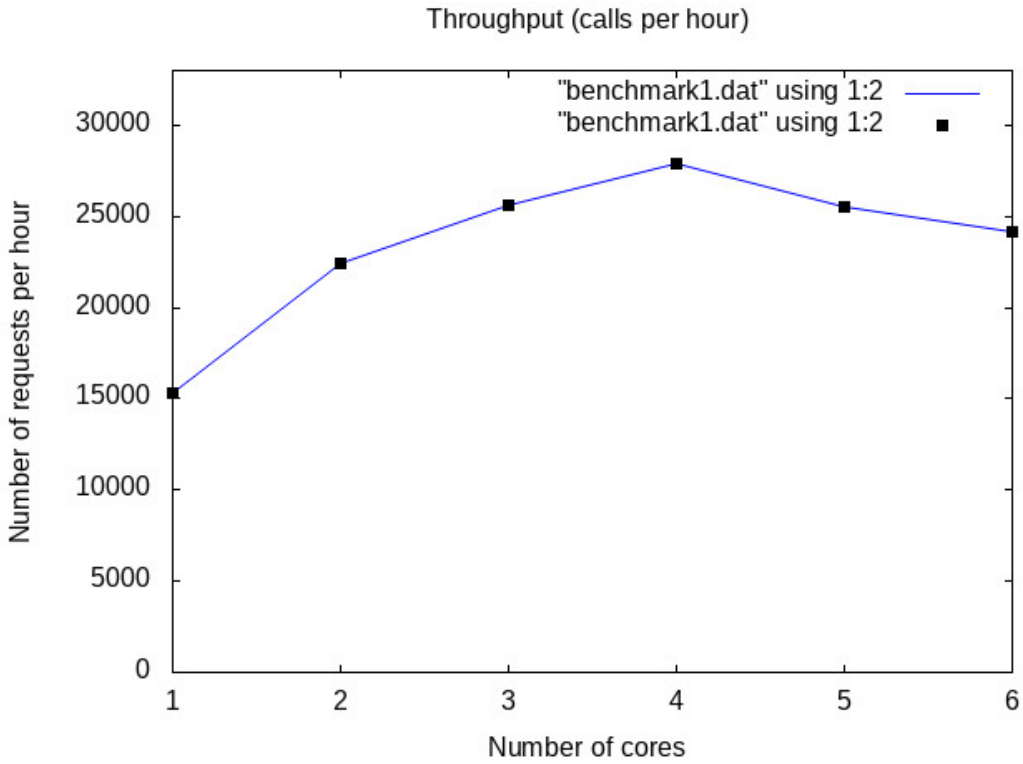
This section is intended to show the maximum number of requests that can be processed per hour. This gives us an idea of the scalability of our application.

The various tests were carried out on a local OpenTripPlanner server. The machine used runs on Ubuntu 18.04 LTS, has 8GB RAM, an 512 Go SSD and an Intel Core i5-5300U CPU running at 2.30GHz (2 physical cores and 2 logical cores).

5.1 Test 1

For this test, we used OSM data from Brussels and GTFS data from the Stib. Here are the results :

- graph building took 1.3 minutes
- the graph build contains : |V| = 139519
- the graph build contains |E| = 367902
- 1 Go was necessery to create the graph

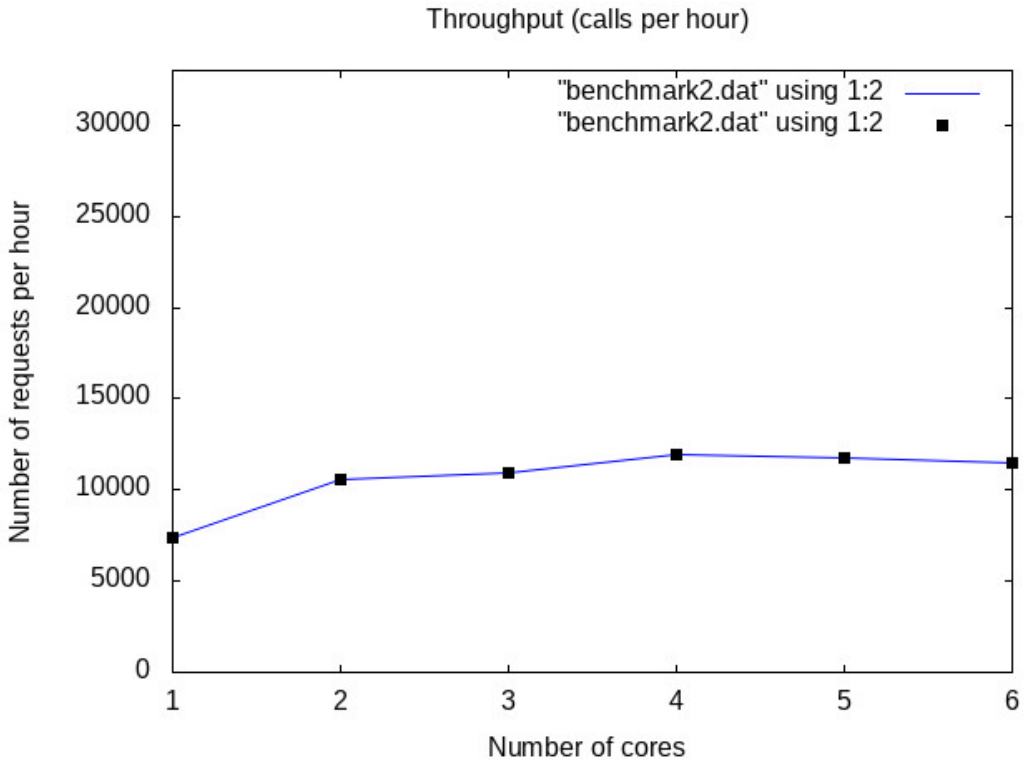


As you can see, the multi-threading allow us to manage more and more queries until we got the limit of our processor. At full capacity, we are able to handle over 25,000 requests per hour (27 720 more exactly) which corresponds to 462 queries per minute. However an user using this application would be limited to Brussels and a trip composed of foot/public transport.

5.2 Test 2

In this second test, we created a bounding box allowing us to enlarge our OSM data by taking Brussels and its periphery. In addition, we have 2 GTFS datasets used by the OpenTripPlanner server. Here are the results :

- graph building took 4.8 minutes
- the graph build contains : |V| = 232752
- the graph build contains |E| = 514152
- 4 Go was necessery to create the graph



Since the graph is larger (|V| increased by 66 % and |E| incresead by 40 %) and more modalities are available, the average number of queries executed per hour decreases significantly compared to our first test. Our application reach 11965 queries per hour which corresponds to 200 queries per minute. We observe a 46% reduction in the number of average requests executed by the application in a given time