MobilityDB Workshop

Alice Lombard and Raphaël Dubuget

COLLABORATORS			
	TITLE : MobilityDB Workshop		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Alice Lombard and Raphaël Dubuget	August 23, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

List of Figures

List of Tables

Abstract

This document is an example of how you can use MobilityDB in order to analyse and visualize data. It is directly inspired by the MobilityDB Workshop already existing. We will throughout this document present different use cases of MobilityDB, which is an extension on PostGIS and PostgreSQL.

This workshop was made by two students from INSA Rennes during an internship in June, July and August 2024.

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

Chapter 1

FirstSteps

This document is a guide on how you can install MobilityDB and use it in order to analyse data.

1.1 Part 1: Starting a database with MobilityDB

1.1.1 Installing MobilityDB

For this workshop, you will need to have MobilityDB installed. Here is a little guide to do that. If you run Ubuntu 22.04 (Jammy Jellyfish), you can simply execute the following in a terminal:

```
sudo apt update
# We will need the foloowing packages:
sudo apt install curl ca-certificates gnupg

# Install the public key for the repository (if not done previously):
curl https://www.postgresql.org/media/keys/ACCC4CF8.asc | gpg --dearmor | sudo tee /etc/apt ↩
    /trusted.gpg.d/apt.postgresql.org.gpg >/dev/null

# Create the repository configuration file:
sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt/ jammy-pgdg main" >> /etc/apt ↩
    /sources.list.d/postgresql.list'
sudo apt update
sudo apt upgrade

# Install the mobilitydb package with apt. It will automatically install the dependancies  ↩
    needed, PosgreSQL and PostGIS in particular.
sudo apt install postgresql-16-mobilitydb
```

1.1.2 PgAdmin4: a PostgreSQL Editor

We are going to execute SQL commands. To do so, you can do it in a terminal or use any IDE you like. If you do not have one, here is the link to install PgAdmin4, you just need to choose your distribution and follow the instructions: https://www.pgadmin.org/-download/.

You could also try DataGrip from Jetbrains. We didn't use it in this workshop, but we're sure it's a good tool.

1.1.3 Starting the database

In order to start the PostgreSQL database, you need to do as following, replacing username by your own username. Note that in this case "workshop" is the name of the database we are creating, but you can choose your own name if you take care to change it every time it is needed later. Additionnaly, <username> should correspond to your OS user name.

```
sudo -u postgres createuser -s -i -d -r -l -w <username>
sudo -u postgres createdb workshop
sudo service postgresql start
psql -U <username> -d workshop
```

You can now connect to the database in pgAdmin. Here are the steps to do so:

- On the dahboard click 'Add New Server'
- Fill out a name of your choosing for the server
- Fill out the host name, which is 'localhost' if you are hosting the database on your machine
- Fill out the port, which is '5432' if you are hosting the database on your machine
- Fill out the username and password you used to create the database

You can now find your database in the browser on the left of the screen. If you're having troule finding it, here is the explicit path in the Objext Explorer on the left: Servers → Server_name → Databases → workshop You can now right click on 'workshop' to open the Querry Tool. This is where we will write and execute our querries.

1.2 Part 2: Playing with the data types

1.2.1 PostgreSQL

In this chapter, we are going to see how we can represent points or a trajectory in mobilityDB. Before that, we'll see how to do it without MobilityDB so that you can see the difference.

First, in PostgreSQL without any extension, we can represent a point with it's coordinates. Here is a small SQL code you can copy and paste into your editor and run it:

```
CREATE TABLE postgrePoints(
    time int,
    longitude float,
    latitude float
);

INSERT INTO postgrePoints VALUES
(0, 0, 0),
(2, 1, 1),
(3, 0.5, 2),
(4, 0, 1),
(6, 1, 1),
(7, 1, 0);

SELECT * FROM postgrePoints;
```

As you can see, we have a table with 3 columns: the time, the longitude and the latitude of each point. We can then manually compute the distance between each point, or visualize the trajectory, but it is not very convenient.

1.2.2 PostGIS

With PostGIS, we can represent a point with a type geometry. Here is a small SQL code you can copy and paste into your editor and run it:

```
-- First, we need to create the extension
CREATE EXTENSION PostGIS;

DROP TABLE IF EXISTS postgispoints;
CREATE TABLE postgisPoints(
    time int,
    point geometry(point)
);

INSERT INTO postgispoints ("time", point)
SELECT time, ST_MakePoint(longitude, latitude)
FROM postgrePoints;

SELECT * FROM postgispoints;
```

As you can see, we have a table with 2 columns: the time and the point. The point is a geometry of type point, which is a PostGIS type. We can then use PostGIS functions to compute the distance between each point, or visualize the trajectory, which is more convenient. For example, we can create a line from the points and visualize it on a map:

```
CREATE TABLE postgisLine(
    line geometry
);

INSERT INTO postgisLine (line)
SELECT ST_MakeLine(ARRAY(SELECT point FROM postgisPoints ORDER BY time));

SELECT * FROM postgisLine;
```

1.2.3 MobilityDB

With MobilityDB, we can represent a trajectory with a tgeompoint. It's pretty much the same as a PostGIS line, except that each point has a time associated with it. Here is how you can make it:

```
DROP EXTENSION IF EXISTS mobilityDB CASCADE;
CREATE EXTENSION mobilityDB CASCADE;

DROP TABLE IF EXISTS mdbTraj CASCADE;
CREATE TABLE mdbTraj(
    traj tgeogpoint
);

INSERT INTO mdbtraj(traj)
SELECT tgeogpointSeq(array_agg(tgeogpoint(point, time) ORDER BY time))
FROM postgisPoints;
```

Chapter 2

Managing Bus Trajectories

We will, for our examples, use data from STAR which is the Rennes' public transport company. This data is protected by ODbL (Open Database License): Data source: STAR Data Explore/Rennes Métropole.

2.1 Collecting the data

The data we used for this chapter can be found here. You can download the data in CSV format. However, since it is real time data, you would just get the positions of the buses at the time of the download. To counter this, we will use the data we collected over the course of one week and stored in the file position-bus.csv. It is the raw data collected with just one simple change: we added a column with the timestamp at which the data was collected.

Here is the script we used to collect the data:

```
import requests
import os

def get_filepaths():
    # Get the current working directory
    cwd = os.path.dirname(os.path.realpath(__file__))

    # Name of the file to create
    filename = 'position-bus.csv'

    # Full paths of the file
    filepath = os.path.join(cwd, filename)

    return filepath

# Use of the function above
filepath = get_filepaths()

# 7 download links
url = 'https://data.explore.star.fr/api/explore/v2.1/catalog/datasets/tco-bus-vehicules- ↩
    position-tr/exports/csv?lang=fr&timezone=Europe%2FBerlin&use_labels=true&delimiter=%3B'

# Function to download the data
def download_data():
    try:
        response = requests.get(url)
        response.raise_for_status()  # Check for HTTP errors
        # Save the data to a file
        with open(filepath, 'ab') as file:
            file.write(response.content)
```

```
        print(f"Data {filepaths.index(filepath) + 1} downloaded successfully.")
    except requests.exceptions.RequestException as e:
        print(f"Error downloading data {filepaths.index(filepath) + 1}: {e}")

if __name__ == "__main__":
    download_data()
```

You can find this script named request.py in the repository, we will reuse it for GBFS data later.

Similarly, the python script add_timestamp.py is the one that adds a column with the timestamps.

2.2 Cleaning the data

The CSV file obtained, named tposition-bus.csv, is 576 MB, and it contains more than 5.2 million rows. It's columns are listed below:

Timestamp	Timestamp of the row, format: yyyy-mm-dd hh:mm:ss.xxxxx (the x's are tenth of thousandth of a second)
Bus (ID)	Unique ID of the bus for that row
Bus (numéro)	Another ID for the buses
Etat	State of the bus (in service, out of service, unknown, deadrunning)
Ligne (ID)	ID of the bus line
Ligne (nom court)	Short name of the bus line
Code du sens	0 or 1 depending on the direction of the bus (if in service)
Destination	Destination of the bus (if in service)
Coordonnées	Exact coordinates of the bus at that moment
Avance / Retard	Earliness / Delay

Table 2.1: Bus columns

While checking if the file was correctly downloaded and there hadn't been any error, we found some rows that were missing. Here is an exemple around row 92740:

```
2024-07-22 14:05:02.815085;--
2024-07-22 14:05:02.815085;Streaming interrupted due to the following error: NotFoundError ↩
    (404, 'search_phase_execution_exception', 'No search context found for id [219908759]') ↩
    2024-07-22 14:06:02.218453
```

To fix this, we simply used yet another python script: cleanup.py. This script will simply remove any line that does not contain data in the expected format. After running the script, we finally have a CSV file that is exploitable with MobilityDB: tposition-bus-clean.csv.

2.3 Preparing the Database

Create a new database RennesBusTrajectories, then use your SQL editor to create the extension MobilityDB:

```
CREATE EXTENSION MobilityDB CASCADE;
```

The cascade command will automatically create the dependencies of MobilityDB, namely PostGIS.

Now, we can create the table that will store the bus trajectories:

```
CREATE TABLE BusInput(
    T timestamp,
    BusID int,
    BusNumber int,
```

```
    State varchar(20),
    LineID int,
    LineName varchar(5),
    Direction int,
    Destination varchar(50),
    Position varchar(30),
    lat float,
    lon float,
    Points GEOMETRY(Point, 4326),
    Delay int
);
```

2.4 Loading the data

Now we want to import CSV data into a PostrgreSQL table. We will use the COPY command as follows:

```
COPY BusInput (T, BusID, BusNumber, State, LineID, LineName,Direction, Destination,  ↩
    Position, Delay)
FROM 'path/to/tposition-bus-clean.csv' DELIMITER ';' CSV HEADER;
```

It is possible that the above command fails with a permission error. The reason for this is that COPY is a server capability, while the CSV file is on the client side. To overcome this issue, one can use the \copy command of psql as follows:

```
psql -d RennesBusTrajectories -c "\copy BusInput (T, BusID, BusNumber, State, LineID,  ↩
    LineName,Direction, Destination, Position, Delay) FROM 'path/to/tposition-bus-clean.csv' ↩
     DELIMITER ';' CSV HEADER;"
```

You can find a bash script in the repository to do that, it is named copyBus.sh.

Whatever method you're using, don't forget to change the path to the CSV file on your computer.

We then create the geometry column Points from the Position column:

```
UPDATE BusInput
SET Points = ST_SetSRID(ST_MakePoint(SPLIT_PART(Position, ',', 2)::FLOAT, SPLIT_PART( ↩
    Position, ',', 1)::FLOAT), 4326);
```

Here, we use the function SPLIT_PART to separate the latitude and longitude from the Position column based on the comma.

While testing, we found some issues with some rows being duplicated, or others having the same timestamp but different positions. Once again, we decided to remove these rows. Here is how to do that:

```
DELETE FROM BusInput WHERE T IN (
  SELECT DISTINCT b1.T FROM BusInput b1 JOIN BusInput b2
  ON b2.BusID = b1.BusID AND b2.T = b1.T AND NOT ST_Equals(b1.Points, b2.Points)
) OR (BusID, Points, T) IN (
  SELECT BusID, Points, T
    FROM BusInput
    GROUP BY BusID, Points, T
    HAVING COUNT(*) > 1
);
```

This effectively deletes any row that falls into one of the two aforementioned categories.

2.5 Constructing the trajectories

Now that our data is clean, we can start constructing the trajectories. They will be created in another table: Busses. Here is the SQL command to create the table:

```
CREATE TABLE Busses(ID, Trip) AS
SELECT BusID, tgeompointSeq(array_agg(tgeompoint(ST_Transform(Points, 4326), T) ORDER BY T) ↩
    )
FROM (
    SELECT BusID, Points, T
    FROM BusInput
    ORDER BY BusID, T) AS SortedBusInput
GROUP BY BusID;
```

This query constructs, per bus, its spatiotemporal trajectory Trip, which is a temporal geometry point.

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

We can visualize this data on QGIS. As you can see on the image below, the data is a bit messy and it is hard to really see the paths taken by the busses. This is probably due to the fact that the positions are updated only once a minute, which is to broad if we want to see the precise turns taken by the buses.

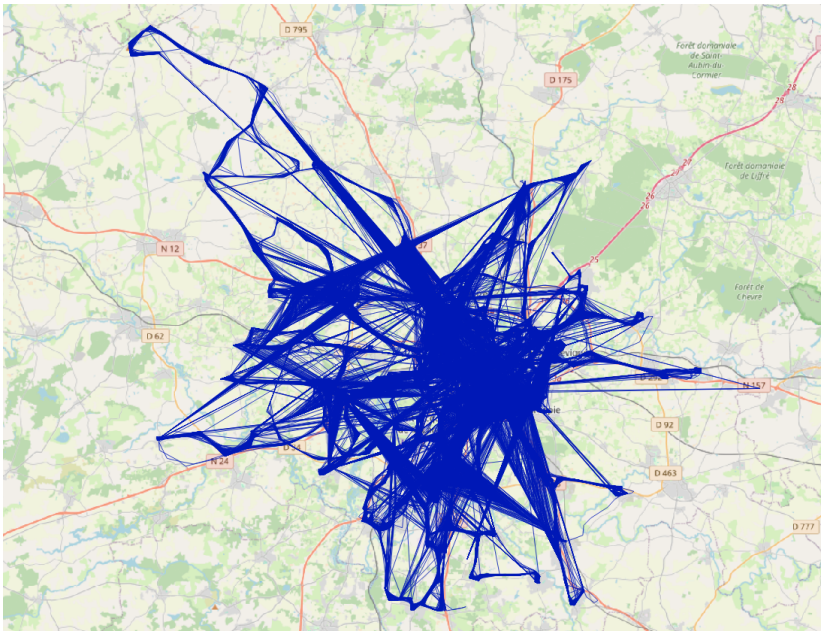


Figure 2.1: Visualisation of the trajectories

Chapter 3

Animating GPX data

We will use data from OpenStreetMap. The traces we are going to use in this example are rando_morbi1.gpx and rando_morbi2.gpx. GPX is a common file format for storing GPS data. It is used by many GPS devices and applications to store track logs and waypoints. In our case, a person did two hikes on the 26th of June 2024, in the Morbihan, and saved the data in two GPX files.
Then, this person exported the data to OpenStreetMap and we can use his trace as we want. We will use these GPX files in this workshop.
Please download the files rando_morbi1.gpx and rando_morbi2.gpx from the OpenStreetMap website and put in a folder you will remember.

3.1 Analyzing the data

3.1.1 Loading GPX data

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

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

```
      <time>2014-12-14T14:32:16.00Z</time>
    </trkpt>
    <trkpt lat="30.16408" lon="31.467218">
      <ele>74</ele>
      <time>2014-12-14T14:32:19.00Z</time>
    </trkpt>
    [...]
  </trkseg>
  <trkseg>
    [...]
  </trkseg>
  [...]
</trk>
<trk>
  [...]
</trk>
[...]
<gpx>
```

To load the GPX data into MobilityDB, we fare going to use ogr2ogr, a command line tool that converts data between different formats.
First, you need to create a database. Let's name it "hiking". Here is how you can do it:

```
sudo -u user psql -d postgres -c "CREATE DATABASE hiking;"
```

Of course you need to replace user by your username.
Then, you need to load the MobilityDB extension into the database. Here is how you can do it:

```
CREATE EXTENSION mobilitydb CASCADE;
```

We are now ready to import the data. Go to the folder where you have put the GPX files and run the following command:

```
ogr2ogr -append -f PostgreSQL PG:dbname=hiking 11390305.gpx
```

You need to replace 11390305 by the name of the file you want to import.
Please run this command with both files.
Now, you should have the data in the database.

3.1.2 Transforming the data

Let's have a look at the table "tracks" and "track_points" which were created by ogr2ogr.

```
SELECT * FROM track_points;
SELECT * FROM tracks;
```

You can see that the table "track_points" contains the points of the track, and the table "tracks" contains the entire tracks. We are going to work with the table "track_points".
First, we see that the two tracks are in the same table. We are going to put an ID for each track so that we can differentiate them. Here is how you can do it:

```
ALTER TABLE track_points ADD COLUMN track_number int;

WITH NumberedTracks AS (
SELECT *, LAG(track_seg_point_id) OVER (ORDER BY ogc_fid) AS prev_value
FROM track_points
),
TrackIdentifiers AS (
SELECT ogc_fid, SUM(CASE WHEN track_seg_point_id < prev_value THEN 1 ELSE 0 END) OVER ( ↩
    ORDER BY ogc_fid) + 1 AS track_id
FROM NumberedTracks
```

```
)
UPDATE track_points tp
SET track_number = ti.track_id
FROM TrackIdentifiers ti
WHERE tp.ogc_fid = ti.ogc_fid;
```

What we are doing here is that we are adding a column "track_number" to the table "track_points". Then, we are creating "NumberedTracks" which contains the previous value of the "track_seg_point_id" column.
Then, we are creating "TrackIdentifiers" which contains the track_id, depending on the previous track_seg_point_id: if it is smaller then we sum 0 but if it is higher, saying we start a new sequence in track_seq_point_id, we sum 1.
Finally, we are updating the table "track_points" with the track_id.

We have to transform these geopgraphic points into a MobilityDB trajectory. Here is how you can do it:

```
DROP TABLE IF EXISTS trips_mdb;
CREATE TABLE trips_mdb (
id int,
date date,
trip tgeompoint,
trajectory geometry,
PRIMARY KEY (id)
);

INSERT INTO trips_mdb(id, date, trip)
SELECT track_number, date(time), tgeompointSeq(array_agg(tgeompoint(
wkb_geometry, time) ORDER BY time))
FROM track_points
GROUP BY track_number, date;

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

What we are doing here is that we are creating a table "trips_mdb" with an id, a date, a trip and a trajectory. The trip is a tgeompoint, which is a MobilityDB type.
We are inserting into this table the track_number, the date and the tgeompointSeq of the array of the tgeompoint.
Finally, we are updating the table with the trajectory of the trip.

We are done with the data loading. In conclusion, we have loaded the GPX data into MobilityDB, and we have created a table "trips_mdb" which contains the trajectories of the trips. We are now ready to visualize the data.

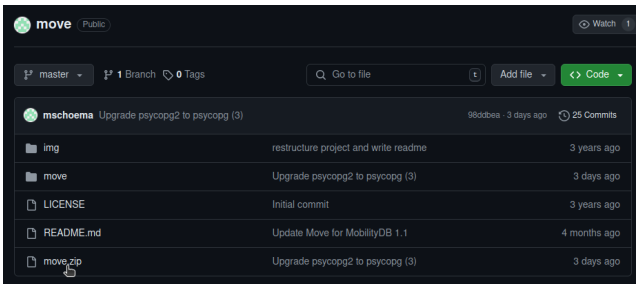
3.2 Visualizing the data

3.2.1 Introduction

Now that we have cleaned the data and made it ready to be used, we will visualize the data in order to have a better understanding of it.
To do so, we are going to use QGIS, a free and open-source geographic information system. You can download it here.
If you're interested, know that we are going to follow this tutorial: MobilityDB: Hands on Tutorial on Managing and Visualizing Geospatial Trajectories in SQL.
We will use the plugin Move to visualize the data. Here is the repo where you can find it.
Please click on the move.zip file, and download it. Take care to do it like in this following image, otherwise it could not be downloaded correctly. Also, put it in a folder you will remember. Now, open QGIS and click on Plugins → Manage and Install Plugins. Click on the install from zip button and select the move.zip file you just downloaded.
Then, you can use the plugin to visualize the data. A MobilityDB logo will appear on the up-left corner of the screen.

You may have an error saying:

```
Couldn't load plugin 'move' due to an error when calling its classFactory() method
    ModuleNotFoundError: No module named 'psycopg'
```



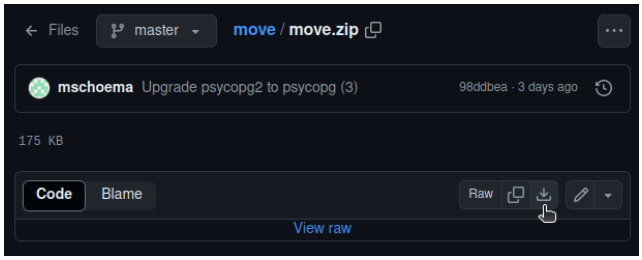


Table 3.1: Download the move.zip file

To fix this, you need to install the psycopg module. You can do it by running the following command in a terminal:

```
pip install psycopg
```

If you've had to install the psycopg module, you will need to uninstall the incorrect move plugin that you just installed (Plugins → Manage and Install Plugins → Installed → Move → Uninstall Plugin) and then reinstall it. You may also need to restart QGIS to make the plugin work.

3.2.2 Connecting to the database

You can now create a new blank project in QGIS. Then in the browser on the left of the screen, you should find a 'PostGIS' line. Right click on it and then 'New Connection...'. You will see this window:

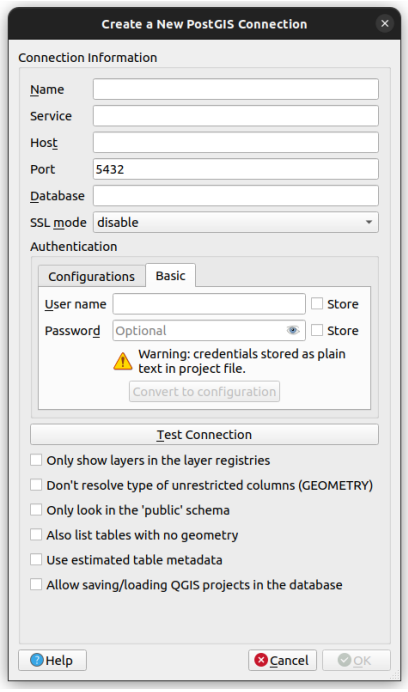


Figure 3.1: New PostGIS Connection

Here is what you need to fill out:

- Name: choose a name for this connection
- Host: if the database is hosted on your machine, then 'localhost'. If not, you should be able to fgure it out by yourself.

- Port: again, if self hosted then keep ’5432’, if not you should know.
- Database: here write the name of the database in which you have loaded the data. In our case, it was "hiking".
- Authentication: go to the ’Basic’ tab and fill out the user name and password for the account that has access to the database. Make sure to check the ’Store’ boxes next to User name and Password.

You can now click OK and find your connection, database and the trips_mdb table in the tree below the PostGIS line in the QGIS Browser.

3.2.3 Visualizing the data

Now that you have connected to the database, you can visualize the data.
First, let’s put a map layer. In the browser window (on the left of the screen), click on ’XYZ Tiles’ and then right click on ’OpenStreetMap’ and select ’Add Layer to Project’.
You will see an OpenStreetMap layer in the inferior left part of the screen.
You can zoom in the part you are studyin. In our case, we are focusing on a part of France, in Britanny, called Morbihan.

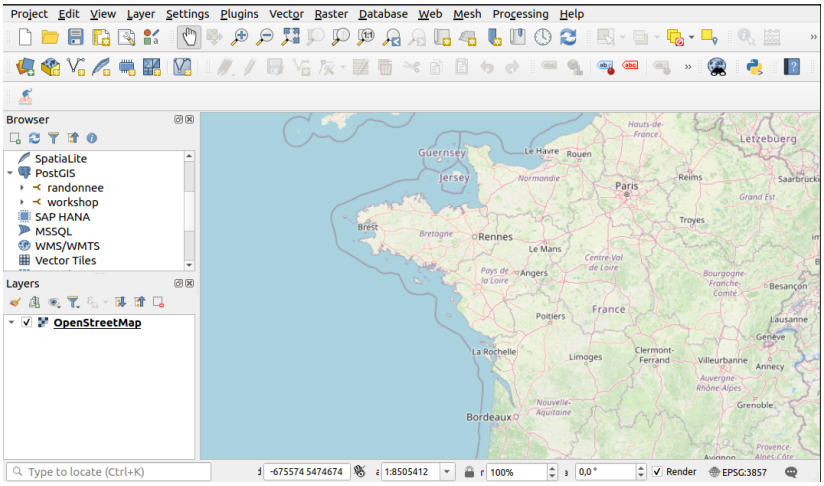


Figure 3.2: OpenStreetMap layer

Then, let’s add the data we have loaded in the database.
First, let’s draw the trajectory. Click on PostGIS → connection_name → public → trips_mdb → Fields → trajectory. You will see the trajectory in the map.
Now, we want to visualize the moving point of the trip.
To do so, you need to click on the ’Move’ logo on the up-left side of the screen.
Select your database and the table you want to visualize. Then, you can write a query and execute it. We are going to put in this query:

```
SELECT date, trip FROM trips_mdb;
```

Now click on ’Execute Query’.
Let’s see the trip moving while we are visualizing it. To do so, click on View → Panels → Temporal Controller. You will see a window like this: Now this is your time to play with the data and visualize it as you want. You can put the time in the temporal controller and see the trip moving. You can also change the color of the trip, the size of the points, etc...

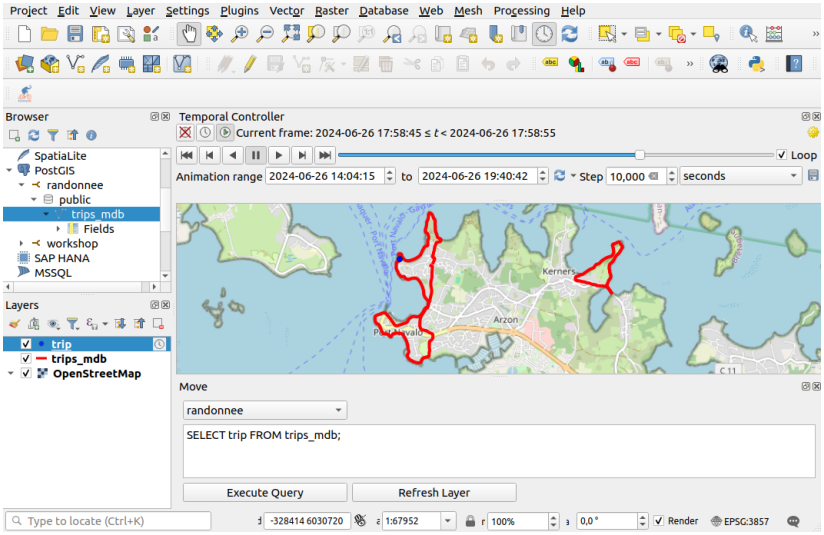


Figure 3.3: Full temporal visualization

Figure 3.4: Visualizing the trip

Chapter 4

GTFS data

4.1 Introduction

In this chapter, we are going to use GTFS data. GTFS (General Transit Feed Specification) is a standard format for public transportation schedules and associated geographic information. It is used by many public transportation companies to share their data.

We are going to use the GTFS data from STAR, the public transportation company of Rennes. You can find the data here, in the static data part, but it will give you the updated data. The exact data we are using in this workshop can be found in the repo, in the GTFS folder.

The GTFS data is composed of several files, such as:

- agency.txt: the agency that operates the transportation service;
- calendar_dates.txt: exceptions for the service;
- calendar.txt: the dates for the service;
- fare_attributes.txt: the fare information;
- feed_info.txt: the feed information;
- routes.txt: the routes of the transportation service;
- shapes.txt: the shapes of the routes;
- stop_times.txt: the times of the stops;
- stops.txt: the stops of the transportation service;
- trips.txt: the trips of the transportation service.

You can find more information about the GTFS data here.

We are not going to use the fare_attributes.txt and feed_info.txt files because it is not relevant for our workshop.

4.2 Importing the data

First thing you need to do is create a MobilityDB extension, just like in the Chapter 1. To do so, follow this command:

```
CREATE EXTENSION mobilitydb CASCADE;
```

Now, we need to import the GTFS data into the database. There is a tool that can help you do that, called gtfs-to-sql. Please install it by following these instructions:

```
sudo apt install npm
sudo apt install moreutils
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"
[ -s "$NVM_DIR/bash_completion" ] && \. "$NVM_DIR/bash_completion"
nvm install 16
nvm use 16
node -v
npm install gtfs-via-postgres
```

Note that you need to have nvm installed on your computer to use this tool, and a recent enough version. gtfs-to-sql will create the tables for you, and fill them with the data from the GTFS files. Here is how to use it:

```
npm exec -- gtfs-to-sql --require-dependencies -- GTFS/*.txt | sponge | psql -d < ↩
    databasename> -b
```

You need to replace <databasename> by the name of your database. You also need to replace GTFS/*.txt by the path to the GTFS files on your computer. You should now have the tables in your database.

Before moving on to the next part, we still have some changes to make.
We need to add some columns to the stops table, fill them with the right SRID, and create a table to store the shape geometries. We also added an index to the shape_geoms table, to make the queries faster.
SRID stands for Spatial Reference System Identifier. It is a unique value that identifies the projection of the data. 4326 is the most common SRID, it is the WGS 84 projection.
We set the SRID to 2154 because it is the projection reference we are using in Rennes, where the data comes from.

```
ALTER TABLE stops ADD COLUMN stop_geom geometry('POINT', 2154);

UPDATE stops
SET stop_geom = ST_SetSRID(stop_loc::geometry,2154);

CREATE TABLE shape_geoms (
  shape_id text NOT NULL,
  shape_geom geometry('LINESTRING', 2154),
  CONSTRAINT shape_geom_pkey PRIMARY KEY (shape_id)
);

CREATE INDEX shape_geoms_key ON shapes (shape_id);
```

4.3 Creating PostGIS geometries

In this part, we are going to create the PostGIS geometries for the stops, the routes and the shapes of the routes. To do so, we are going to use the stops, routes and shapes tables we created earlier.

4.3.1 I

n this part, we are going to make the shapes of the routes. To do so, we are going to use the shape_geoms table we created earlier. This table is used to store the shape geometries of the routes. We are going to use the shapes.txt file to create the shapes of the routes.

```
INSERT INTO shape_geoms
SELECT
    shape_id,
```

```
    ST_MakeLine(array_agg(
        ST_SetSRID(
            ST_MakePoint(shape_pt_lon, shape_pt_lat),
        2154)
    ORDER BY shape_pt_sequence))
FROM shapes
GROUP BY shape_id;
```

We are filling the shape_geoms table with two attributes. The first one is shape_id: the identifier of the shape, which is taken from the shapes table. The second one is the geometry of the shape, which is created in three steps:

- We create a point for each shape_pt_lon and shape_pt_lat in the shapes table, using the ST_MakePoint function. This is a PostGIS function.
- We set the SRID of the point to 2154 because it is the norm we are using (see more here). We use the ST_SetSRID function to do it. This is a PostGIS function.
- We use the array_agg function to aggregate all the points in an array. This is a PostgreSQL function. We create a line with all the points we created, using the ST_MakeLine function. This is a PostGIS function.

We also ordered the points by shape_pt_sequence to make sure the line is in the right order. Finally, we group the lines by shape_id to make sure we have only one line per shape.

Now that we have the shapes of the routes, we can move on to the next part.

4.3.2 I

n this part, we are going to generate the service dates. To do so, we are going to use the calendar and calendar_dates tables we created earlier.
To put it in a nutshell, we compute the information about the service dates from the calendar and calendar_dates tables, so that we have the actual service offered, taking into account exceptions from the calendar_ates table.

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

You can notice that we are using the date_trunc function. This function truncates the date to the specified precision. In this case, we are truncating the date to the day. This is a PostgreSQL function.
We also use the extract function. This function extracts a field from a date or time value. In this case, we are extracting the day of the week. This is a PostgreSQL function. We write "extract(isodow FROM d)" to get the day of the week of the date d. "isodow" stands for ISO day of the week (here is the postgreSQL documentation about isodow)
We also use the generate_series function. This function generates a series of values, in this case, dates. We are using it to generate all the dates between the start_date and the end_date of the service. This is a PostgreSQL function.

4.4 Creating trips

Now that we are fully prepared, we can create the trips. To do so, we are going to create two tables: trip_stops and trip_segs. Let's start with the first one.

4.4.1 I

n this part, we are going to create the trip_stops table.

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

Now, let's fill this table.

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

Here, we are just filling the table with the data from the trips and stop_times tables. We are also adding the no_stops attribute, which is the number of stops of the trip. We are using the MAX function to get the maximum stop_sequence for each trip. MAX is a basic PostgreSQL function, so is OVER (PARTITION BY ...).

Now, let's update the table with the percentage of the trip that has been done.

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

You can see that there are 3 cases:

- The first one is when the stop_sequence is 1. In this case, the percentage is 0.0 because the trip has just started (1 is the first stop).
- The second one is when the stop_sequence is equal to no_stops. In this case, the percentage is 1.0 because the trip is over (no_stops is the last stop).
- The last one is when the stop_sequence is between 1 and no_stops. In this case, we use the ST_LineLocatePoint function. This function returns a float between 0 and 1 representing the location of the closest point on the line to the given point. We use it to get the percentage of the trip that has been done. It is much better than just dividing the stop_sequence by no_stops because it takes into account the shape of the route.

The perc attribute is now filled with the percentage of the trip that has been done. Let's delete NULL values, as they are not relevant.

```
DELETE FROM trip_stops WHERE perc IS NULL;
```

The trip_stops table is now filled with the data we need. Let's move on to the next part.

4.4.2 l

n this part, we are going to create the trip_segs table, that will be used to store the segments of the trips.

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

Now, let's fill this table.

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

There is a new set of functions here.
We are using the LEAD function. This function provides access to a row at a given physical offset beyond that position. In our case, we do not specify the offset, so it is 1 (the default value). This is a PostgreSQL function.
Another notion is the window: this clause defines a window specification that says how to partition the rows and how to order them. In our case, we partition the rows by trip_id and order them by stop_sequence, that is to say the order of the stops in the trip. This is a PostgreSQL function.
In the end, the attributes that need the LEAD function are filled with the next stop's data, so the next row.

We just need to delete the incoherent data, that is to say the data where perc1 is greater or equal than perc2. Indeed, a trip cannot go backwards.

```
DELETE FROM trip_segs WHERE perc1 >= perc2;
```

We still need to fill the seg_geom attribute with the geometry of the segment. To do so, we are going to use the shape_geoms table we created earlier.

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

We are using the ST_LineSubstring function. We use it to get the segment of the line between the two points.

We also need to fill the seg_length and no_points attributes.

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

You can see that we are using the ST_Length function to get the length of the segment and the ST_NumPoints function to get the number of points of the segment.

The trip_segs table is now filled with the data we need. Let's move on to the next part.

4.4.3 I

n this part, we are going to create the trips points table.

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

Now, let's fill this table.

```
INSERT INTO trip_points (trip_id, route_id, service_id, stop1_sequence, point_sequence,  ↩
    point_geom, point_arrival_time)

WITH
temp1 AS (
    SELECT
        trip_id,
        route_id,
        service_id,
        stop1_sequence,
        stop2_sequence,
        no_stops,
        stop1_arrival_time,
        stop2_arrival_time,
        seg_length,
        (dp).path[1] AS point_sequence,
        no_points,
        (dp).geom as point_geom
    FROM trip_segs, ST_DumpPoints(seg_geom) AS dp
),
temp2 AS (
    SELECT
        trip_id,
        route_id,
```

```
        service_id,
        stop1_sequence,
        stop1_arrival_time,
        stop2_arrival_time,
        seg_length,
        point_sequence,
        no_points,
        point_geom
    FROM temp1
    WHERE point_sequence <> no_points OR stop2_sequence = no_stops
),
temp3 AS (
    SELECT
        trip_id,
        route_id,
        service_id,
        stop1_sequence,
        stop1_arrival_time,
        stop2_arrival_time,
        point_sequence,
        no_points,
        point_geom,
        ST_Length(ST_MakeLine(array_agg(point_geom) OVER w)) / seg_length AS perc
    FROM temp2 WINDOW w AS (PARTITION BY trip_id, service_id, stop1_sequence ORDER BY  ↩
        point_sequence)
)

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

FROM temp3;
```

4.4.4 I

n this part, we are going to create the trip_input table. It will be used to store the trips with the right dates.

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

Now, let's fill this table.

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

We also have to delete the data that is not relevant.

```
DELETE FROM trips_input
    WHERE trip_id in (
        SELECT distinct t1.trip_id FROM trips_input t1
        JOIN trips_input t2 ON t2.trip_id = t1.trip_id and t2.service_id = t1.service_id ↩
            and t2.route_id = t1.route_id and t2.t = t1.t and NOT ST_Equals(t2.point_geom,t1 ↩
            .point_geom)
    )and t in (
        SELECT distinct t1.t FROM trips_input t1
        JOIN trips_input t2 ON t2.trip_id = t1.trip_id and t2.service_id = t1.service_id ↩
            and t2.route_id = t1.route_id and t2.t = t1.t and NOT ST_Equals(t2.point_geom,t1 ↩
            .point_geom)
);
```

The trips_input table is now filled with the data we need. Let's move on to the next part.

4.4.5 I

n this part, we are going to create the trips table. It will be used to store the trips with the right dates.

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

Now, let's fill this table.

```
INSERT INTO trips_mdb(trip_id, service_id, route_id, date, trip)
    SELECT trip_id, service_id, route_id, date, tgeompointSeq(array_agg(tgeompoint( ↩
        point_geom, t) ORDER BY T))
    FROM trips_input
    GROUP BY trip_id, service_id, route_id, date
;
```

We also need to insert the trips that are not in the service_dates table.

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

That's it! You now have the trips table filled with the data you need.

Chapter 5

GBFS data (applicable for every JSON data)

We will, for our examples, use data from STAR which is the Rennes' public transport company (link). This data is protected by ODbL (Open Database License): Data source: STAR Data Explore/Rennes Métropole.

5.1 Presentation

The General Bikeshare Feed Specification (GBFS) is a data format that is used to provide real-time information about the status of a bike-sharing system. It is various JSON files that contain information about the stations, the bikes, the status of the stations, the status of the bikes, etc...
Among them, we can find the following files:

- gbfs.json: Root file describing all the files used in the STAR bike share system's use of the GBFS standard. We didn't collect data for 1 week, because data doesn't significally change.
- station_information.json: List of stations in STAR's bike sharing network, their capacities, locations and geolocations.
- system_regions.json: STAR bike share application region.
- system_alerts.json: Alerts and unavailabilities for stations in the STAR bike share network. It is updated every minute. We collected data for 1 week, using a script in the repository.
- system_information.json: General information about STAR bike share (organization, contact, e-mail, website).
- system_hours.json: STAR bike share opening hours.
- system_calendar.json: STAR bike share opening days.
- station_status.json: Information on available services, bike availability and locations for each station in the STAR bike share network. We collected data for 1 week.
- free_bike_status.json: List of bikes available for hire throughout the STAR bike sharing network. We collected data for 1 week.
- system_pricing_plans.json: STAR bike sharing tariffs and subscriptions (Summary version).

We are going to use the station_information.json, station_status.json and free_bike_status.json files in order to illustrate the use of MobilityDB with JSON data.

In order to easily understand what we are going to do in this workshop, here are some schemas.
You can see that we have 3 JSON files: station_information.json, station_status.json and free_bike_status.json. You can also understand their structure.
We are going to put them into 3 separate tables in order to be able to use them with MobilityDB.

Figure 5.1: Schema for JSON Data

station_information

id int	name varchar(100)	position geometry	capacity int
5501	République	01010000206A080000B1A547533DD9FABFD7DAFB54150E4840	45
5502	Mairie - Opéra	01010000206A080000ACE5CE4C30DCFABF6FB9FAB1490E4840	24

docks

id int	tdocks temporal int
5501	[24@2024-07-22 11:06:02.238961+02, 29@2024-07-22 11:14:01.624938+02, ... 37@2024-07-29 11:06:01.994103+02]
5502	[13@2024-07-22 11:06:02.238961+02, 12@2024-07-22 13:21:02.159108+02, ... 14@2024-07-29 11:06:01.994103+02]

bikes

lat float	lon float	tbikes temporal int	tdisabled temporal int	treserved temporal int
48.087309	-1.643683	[4@2024-07-22 11:06:02.12504+02, 7@2024-07-22 14:28:01.896671+02, ... 4@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]
48.089978	-1.690244	[4@2024-07-22 11:06:02.12504+02, 3@2024-07-22 11:33:02.145559+02, ... 7@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]

Figure 5.2: Schema of tables

5.2 Bikes table

In this part, we are focusing on the free_bike_status.json file. This file contains information about the bikes available for hire throughout the STAR bike sharing network.
Do not forget to use the MobilityDB extension in PostgreSQL. You can do it with the following command:

```
CREATE EXTENSION IF NOT EXISTS mobilityDB CASCADE;
```

First, we need to import the data into a table. We are going to use this script in order to create the table and import the data:

```
#!/bin/bash

input_file="../../data/free_bike.json"
temp_file="/tmp/processed_data.sql"

# Drop the raw_json_data table if it exists
psql -d bikes -c "DROP TABLE IF EXISTS raw_json_bike;"

# Create raw_json_data table if it doesn't exist
psql -d bikes -c "CREATE TABLE raw_json_bike (timestamp TIMESTAMP, json_data JSON);"

# Initialize the temporary file
> $temp_file

# Read the input file line by line
while IFS= read -r line
do
    # Check if the line is a timestamp
    if [[ "$line" =~ ^[0-9]{4}-[0-9]{2}-[0-9]{2} ]]
    then
        timestamp="$line"
    else
        # Write an INSERT statement to the temporary file
        echo "INSERT INTO raw_json_bike (timestamp, json_data) VALUES ('$timestamp', '$line ↩
            ');" >> $temp_file
    fi
done < $input_file

# Execute the SQL commands to insert data
psql -d bikes -f $temp_file

# Clean up
rm $temp_file
```

This script is going to create a table called raw_json_bike and import the data from the free_bike_status.json file. We also have to manage the timestamp in the JSON file, that's why we do the for loop in the script.

Now that we have the data in the table (you can check it with a SELECT * FROM raw_json_bike;), we can create a table that will contain the data we are interested in. We are going to create a table called bikes with the following script:

```
DROP TABLE IF EXISTS bikes CASCADE;
CREATE TABLE IF NOT EXISTS bikes (
  time timestamp,
  bike_id VARCHAR(100),
  lat float,
  lon float,
  is_reserved int,
  is_disabled int,
  PRIMARY key(bike_id, time)
);
```

We are going to insert the data from the raw_json_bike table into the bikes table with the following script:

```
INSERT INTO bikes (time, bike_id, lat, lon, is_reserved, is_disabled)
SELECT
    timestamp,
    bike->>'bike_id' AS bike_id,
    (bike->>'lat')::float AS lat,
    (bike->>'lon')::float AS lon,
    (bike->>'is_reserved')::int AS is_reserved,
    (bike->>'is_disabled')::int AS is_disabled
FROM
    raw_json_bike,
    json_array_elements(json_data->'data'->'bikes') AS bike;
```

Now we have the data in the bikes table. We can check it with a SELECT * FROM bikes;.
We can also create a table that will count the number of bikes, the number of disabled bikes and the number of reserved bikes for each timestamp, latitude and longitude. We are going to create a table called bike2 with the following script:

```
DROP TABLE IF EXISTS bike2;
CREATE TABLE IF NOT EXISTS bike2 (
    time timestamp,
    lat float,
    lon float,
    bikes int,
    disabled int,
    reserved int
);

INSERT INTO bike2(time,lat,lon,bikes,disabled,reserved)
SELECT time, lat, lon, COUNT(*) AS bikes, COUNT(*) FILTER(WHERE is_disabled=1) AS disabled, ↩
    COUNT(*)FILTER(WHERE is_reserved=1) AS reserved
FROM bikes GROUP BY time,lat,lon;

SELECT * FROM bike2 order BY time, lat, lon;
```

It should look like this: Now, the final step is to create a table that will contain the data we are interested in: the number of bikes,

	time timestamp without time zone	lat double precision	lon double precision	bikes integer	disabled integer	reserved integer
1	2024-07-22 11:06:02.12504	48.087309	-1.643683	4	0	0
2	2024-07-22 11:06:02.12504	48.089978	-1.690244	4	0	0
3	2024-07-22 11:06:02.12504	48.091114	-1.682284	4	0	0
4	2024-07-22 11:06:02.12504	48.091594	-1.667321	6	0	0
5	2024-07-22 11:06:02.12504	48.093292	-1.674116	7	0	0
6	2024-07-22 11:06:02.12504	48.094986	-1.663731	4	0	0
7	2024-07-22 11:06:02.12504	48.097505	-1.674834	4	0	0
8	2024-07-22 11:06:02.12504	48.099043	-1.694512	6	0	0
9	2024-07-22 11:06:02.12504	48.102015	-1.684015	1	0	0
10	2024-07-22 11:06:02.12504	48.102055	-1.673959	3	0	0

Figure 5.3: Table bikes2

the number of disabled bikes and the number of reserved bikes for each latitude and longitude, using the temporal integer type.

```
DROP TABLE IF EXISTS temporal;
CREATE TABLE IF NOT EXISTS temporal(
```

```
    lat float,
    lon float,
    tbikes tint(SEQUENCE),
    tdisabled tint(SEQUENCE),
    treserved tint(SEQUENCE)
);


WITH bbike AS (
    SELECT lat, lon,
        tintSeq(array_agg(tint(bikes, time) ORDER BY time)) AS sorted_tbikes,
        tintSeq(array_agg(tint(disabled, time) ORDER BY time)) AS sorted_tdisabled,
        tintSeq(array_agg(tint(reserved, time) ORDER BY time)) AS sorted_treserved
    FROM bike2
    GROUP BY lat, lon
)
INSERT INTO temporal(lat, lon, tbikes, tdisabled, treserved)
SELECT lat, lon,
        sorted_tbikes AS tbikes,
        sorted_tdisabled AS tdisabled,
        sorted_treserved AS treserved
FROM bbike;

SELECT * FROM temporal order BY lat, lon;
```

It should look like the bikes table in the following figure:

station_information

id int	name varchar(100)	position geometry	capacity int
5501	République	01010000206A080000B1A547533DD9FABFD7DAFB54150E4840	45
5502	Mairie - Opéra	01010000206A080000ACE5CE4C30DCFABF6FB9FAB1490E4840	24

docks

id int	tdocks temporal int
5501	[24@2024-07-22 11:06:02.238961+02, 29@2024-07-22 11:14:01.624938+02, ... 37@2024-07-29 11:06:01.994103+02]
5502	[13@2024-07-22 11:06:02.238961+02, 12@2024-07-22 13:21:02.159108+02, ... 14@2024-07-29 11:06:01.994103+02]

bikes

lat float	lon float	tbikes temporal int	tdisabled temporal int	treserved temporal int
48.087309	-1.643683	[4@2024-07-22 11:06:02.12504+02, 7@2024-07-22 14:28:01.896671+02, ... 4@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]
48.089978	-1.690244	[4@2024-07-22 11:06:02.12504+02, 3@2024-07-22 11:33:02.145559+02, ... 7@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]

Figure 5.4: Table temporal (bikes in the picture)

5.3 Stations table

In this part, we are focusing on the station_status.json file. This file contains information on available services such as bike availability for each station in the STAR bike share network.
We are going to use the same process as for the bikes table. We are going to create a table called raw_json_status and import the data from the station_status.json file. We are going to use the following script:

```
#!/bin/bash

input_file="../../data/station.json"
temp_file="/tmp/processed_data.sql"

# Drop the raw_json_data table if it exists
psql -d bikes -c "DROP TABLE IF EXISTS raw_json_status;"

# Create the raw_json_data table
psql -d bikes -c "CREATE TABLE IF NOT EXISTS raw_json_status (timestamp TIMESTAMP,  ↩
    json_data JSON);"

# Initialize the temporary file
> $temp_file

# Read the input file line by line
while IFS= read -r line
do
    # Check if the line is a timestamp
    if [[ "$line" =~ ^[0-9]{4}-[0-9]{2}-[0-9]{2} ]]
    then
        timestamp="$line"
    else
        # Write an INSERT statement to the temporary file
        echo "INSERT INTO raw_json_status (timestamp, json_data) VALUES ('$timestamp', ' ↩
            $line');" >> $temp_file
    fi
done < $input_file

# Execute the SQL commands to insert data
psql -d bikes -f $temp_file

# Clean up
rm $temp_file
```

Now that we have the data in the table (you can check it with a SELECT * FROM raw_json_status;), we can create a table that will contain the data we are interested in. We are going to create a table called docks with the following script:

```
drop table if exists docks;
CREATE table docks(
  id int,
  time timestamp,
  docks int
);


INSERT INTO docks (id, time, docks)
SELECT
    (dock->>'station_id')::int,
    timestamp,
    (dock->>'num_docks_available')::int
FROM
    raw_json_status,
    json_array_elements(json_data->'data'->'stations') AS dock;
```

```
select * from docks;
```

We can also create a table that will count the number of docks for each station for each timestamp. We are going to create a table called tempdocks with the following script:

```
DROP TABLE IF EXISTS tempdocks;
CREATE TABLE IF NOT EXISTS tempdocks(
  id int,
  tdocks tint(SEQUENCE)
);


WITH ddocks AS (
    SELECT
    id,
        tintSeq(array_agg(tint(docks, time) ORDER BY time)) AS sorted_tdocks
    FROM docks
    GROUP BY id
)
INSERT INTO tempdocks(id, tdocks)
SELECT id,
       sorted_tdocks AS tdocks
FROM ddocks;

SELECT * FROM tempdocks order BY id;
```

It should look like this:

station_information

id int	name varchar(100)	position geometry	capacity int
5501	République	01010000206A080000B1A547533DD9FABFD7DAFB54150E4840	45
5502	Mairie - Opéra	01010000206A080000ACE5CE4C30DCFABF6FB9FAB1490E4840	24

docks

id int	tdocks temporal int
5501	[24@2024-07-22 11:06:02.238961+02, 29@2024-07-22 11:14:01.624938+02, ... 37@2024-07-29 11:06:01.994103+02]
5502	[13@2024-07-22 11:06:02.238961+02, 12@2024-07-22 13:21:02.159108+02, ... 14@2024-07-29 11:06:01.994103+02]

bikes

lat float	lon float	tbikes temporal int	tdisabled temporal int	treserved temporal int
48.087309	-1.643683	[4@2024-07-22 11:06:02.12504+02, 7@2024-07-22 14:28:01.896671+02, ... 4@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]
48.089978	-1.690244	[4@2024-07-22 11:06:02.12504+02, 3@2024-07-22 11:33:02.145559+02, ... 7@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]

Figure 5.5: Table tempdocks (docks in the picture)

5.4 Stations table

In this part, we are focusing on the station_information.json file. This file contains information about the stations in the STAR bike sharing network, their capacities, locations and geolocations.
This part should be easier than the bikes and docks tables because the information doesn't change along the week. We are going to create a table called raw_json_station and import the data from the station_information.json file. We are going to use the following script:

```
#!/bin/bash

input_file="../station_information.json"

# Drop the raw_json_data table if it exists
psql -d bikes -c "DROP TABLE IF EXISTS raw_json_station;"

# Create the raw_json_data table
psql -d bikes -c "CREATE TABLE IF NOT EXISTS raw_json_station (json_data JSON);"

# Read the JSON data and escape single quotes
json_data=$(cat "$input_file" | sed "s/'/''/g")

# Insert the JSON data into the table
psql -d bikes -c "INSERT INTO raw_json_station (json_data) VALUES ('$json_data');"
```

Now that we have the data in the table (you can check it with a SELECT * FROM raw_json_station;), we can create a table that will contain the data we are interested in. We are going to create a table called stations with the following script:

```
drop table if exists prestation;
CREATE table prestation(
  id int primary key,
  name varchar(100),
  position geometry(point),
  capacity int
);

insert into prestation(id, name, position, capacity)
select
  (stations->>'station_id')::int,
    stations->>'name',
  ST_SetSRID(ST_MakePoint((stations->>'lon')::float,(stations->>'lat')::float),2154),
  (stations->>'capacity')::int
FROM
    raw_json_station,
    json_array_elements(json_data->'data'->'stations') AS stations;

select * from prestation;
```

It should look like this:

5.5 Final view

Now that we have the bikes, docks and stations tables, we can create a view that will contain all the information we are interested in. We are going to create a view called final_view with the following script:

```
CREATE OR REPLACE VIEW station_view AS
SELECT
    prestation.id,
    prestation.name,
    prestation.position,
```

station_information

id int	name varchar(100)	position geometry	capacity int
5501	République	01010000206A080000B1A547533DD9FABFD7DAFB54150E4840	45
5502	Mairie - Opéra	01010000206A080000ACE5CE4C30DCFABF6FB9FAB1490E4840	24

docks

id int	tdocks temporal int
5501	[24@2024-07-22 11:06:02.238961+02, 29@2024-07-22 11:14:01.624938+02, ... 37@2024-07-29 11:06:01.994103+02]
5502	[13@2024-07-22 11:06:02.238961+02, 12@2024-07-22 13:21:02.159108+02, ... 14@2024-07-29 11:06:01.994103+02]

bikes

lat float	lon float	tbikes temporal int	tdisabled temporal int	treserved temporal int
48.087309	-1.643683	[4@2024-07-22 11:06:02.12504+02, 7@2024-07-22 14:28:01.896671+02, ... 4@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]
48.089978	-1.690244	[4@2024-07-22 11:06:02.12504+02, 3@2024-07-22 11:33:02.145559+02, ... 7@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]

Figure 5.6: Table prestation (station_informations in the picture)

```
    prestation.capacity,
    tbikes,
    tdisabled,
    treserved,
    tdocks
FROM
    prestation
  JOIN temporal ON ST_SetSRID(ST_MakePoint(temporal.lon,temporal.lat),2154)=prestation. ↩
      position
  JOIN tempdocks ON tempdocks.id=prestation.id;

SELECT * FROM station_view ORDER BY id;
```

It should look like this:

5.6 Conclusion

We have seen how to import JSON data into tables and how to use MobilityDB with JSON data. We have created 3 tables: bikes, docks and stations. We have also created a view that contains all the information we are interested in. We have used the temporal integer type in order to store the data in the tables. We have also used the MobilityDB extension in PostgreSQL in order to be able to use the temporal integer type.
We could analyze the data in order to see the evolution of the number of bikes, the number of disabled bikes, the number of reserved bikes and the number of docks for each station. We could also analyze the data in order to see the evolution of the number of bikes, the number of disabled bikes, the number of reserved bikes and the number of docks for each latitude and longitude.

Station

id int	name varchar(100)	position geometry	capacity int	tbikes temporal int	tdisabled temporal int	treserved temporal int	tdocks temporal int
5501	République	01010000206A080000B1A547533DD9FABFD7DAFB54150E4840	45	[4@2024-07-22 11:06:02.12504+02, 7@2024-07-22 14:28:01.896671+02, ... 4@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]	[24@2024-07-22 11:06:02.238961+02, 29@2024-07-22 11:14:01.624938+02, ... 37@2024-07-29 11:06:01.994103+02]
5502	Mairie - Opéra	01010000206A080000ACE5CE4C30DCFABF6FB9FAB1490E4840	24	[4@2024-07-22 11:06:02.12504+02, 3@2024-07-22 11:33:02.145559+02, ... 7@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]	[0@2024-07-22 11:06:02.12504+02, 0@2024-07-29 11:06:01.878478+02]	[13@2024-07-22 11:06:02.238961+02, 12@2024-07-22 13:21:02.159108+02, ... 14@2024-07-29 11:06:01.994103+02]

Figure 5.7: Table station_view (final view in the picture)