Report: Public Transport Project

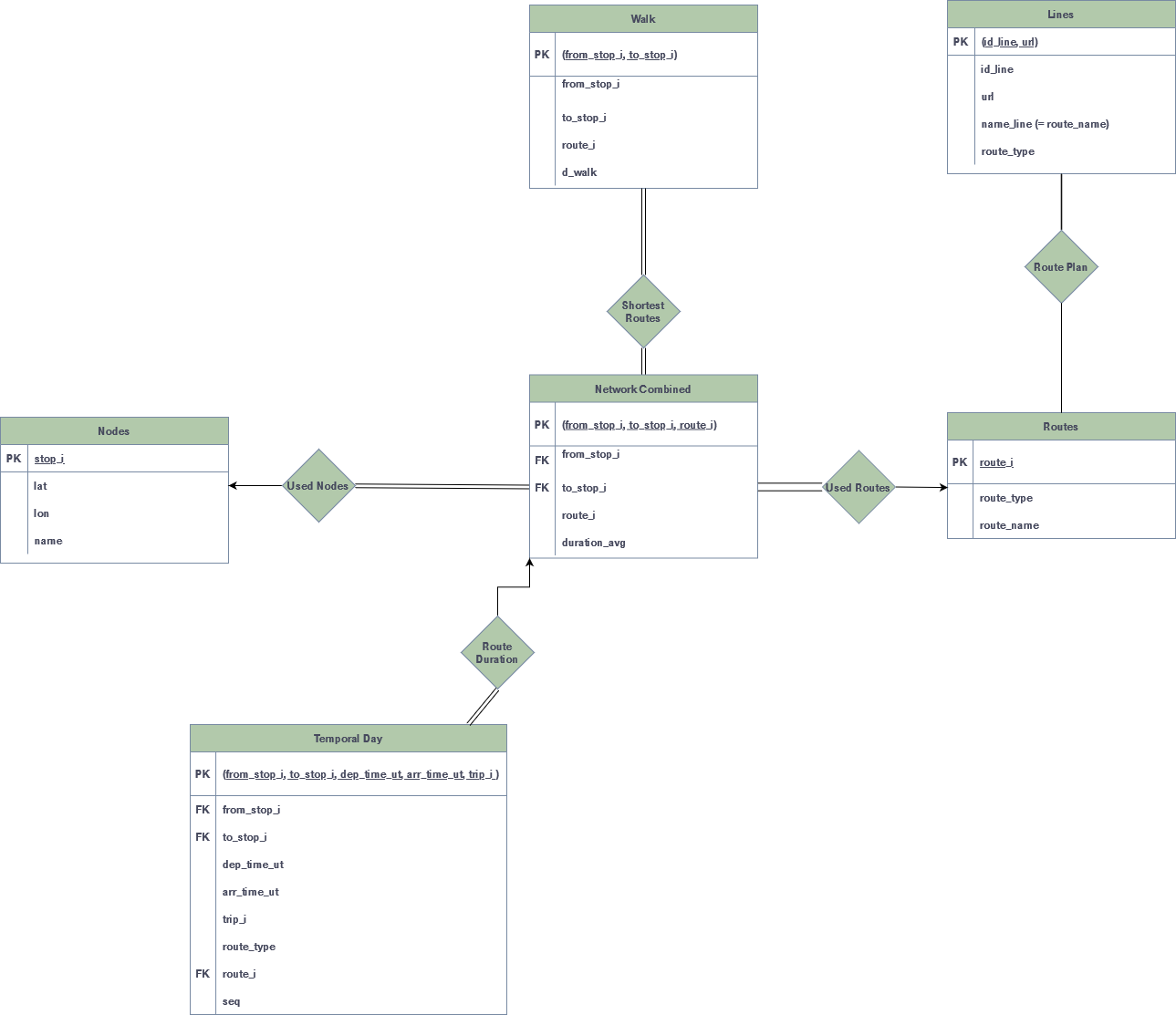
I – Functionalities list

A **requirements.txt** file has been generated to help the end-user install the modules necessary for this program. A **README.MD** file was written to clarify the installation and usage.

Given a dataset of public transport network data (.**csv** files containing network nodes and temporal data, .**geojson** files containing routes and stops names, all taken from <https://www.nature.com/articles/sdata201889>, as well as additional **.json** files taken from ) that is kept within a folder on the user’s computer, as well as a *PostgreSQL* server, our *PyQT5* application can:

* Prompt the user the path to the **data** folder (with a *FileDialog* that will be used to select the folder), as well as the login credentials necessary to connect to the postgres database (with a custom *QDialog* that has four entries: **user**, **password** (this one does not show plain text), **database** and **host** (the IP address which the postgres server runs on).
* Save the variables from previous step inside a **params.json** file so they can be reused next time the program is launched. The json file, if already existing, is converted back into a set of variables by using the *json* module.
* Fill the postgres database with data obtained from the files within the **data** folder. The module *pandas* is used to convert the *.csv* files into a **DataFrame** (pandas’ way of storing tabular data). Said DataFrame is then used to create an equivalent *PostgreSQL* table. When necessary, we use the *psycopg2* module to run additional *SQL* requests that set the uniqueness of certain attributes in the table.
* Display a *QWindow* where the user can select a departure and a destination from two different Combo Boxes. Once both are selected, the user can press the “Go” button to launch the query that will give a path from the departure point to the arrival point. The user can also click the “Clear” button to clear the text within the Combo Boxes.
* Said window also features a *QTableWidget* that shows the detailed route the user must take to go from the departure point to the destination point on its first row, and the estimated time it would take on the second row. Clicking on a transport name (*RER E, BUS 165*…) within that table will make appear a message box, asking the user if they want to see said transport’s plan/map. If the user accepts, we will open the transport’s plan page within their default web browser. If there are multiple plans related to a transport name (often happens given how generic bus names are for example).
* Below that table, is a map (displayed with the *folium* module from a web page), which, when a route has been decided, shows all the stops of said route, linking each stop by a line, otherwise it shows all the stops that we know of on the map. The user can click a location on the map to select their departure and destination stops as well. All these actions are done by running *JavaScript* code with the *folium* module to interact with the map websites.
* Once the user has decided on a path to take, we use the *NetworkX* module to define the shortest path between the departure node and the arrival node. Given a set of *nodes* linked by *edges* (here routes), the module runs the *Dijkstra algorithm*, a graph known for its efficiency on finding the shortest path for road networks.

II – Entity-Relationship Diagram & Dependencies



*Fig.X: The* ***Entity-Relation Diagram***

The entities have been found with the following logic:

* The tables **nodes, combined, walk** and **temporal\_day** are based on the .*csv* files in the data folder.
* The table**route** is based on the .*geojson* file in the data folder.
* The table **line** is based on the *.json* files *fichers-horaires-et-plans* & *referentiel-des-lignes.*
* All the tables above have been imported by keeping the structure of their file:
* the keys for the tables based on the *.csv* files take the values of the first row within said files for each column that we need.
* The keys for the **route** table uses the *geojson* module to read the file as a dictionary and base the final table on the keys of said dictionary.
* The keys for the **line** table were generated in the same way, except that we merged both *json* dictionaries to only keep the data of transport lines that have an *URL* attached to them.

The **nodes** file stores data about all the stops in Paris. It contains the following keys:

* *stop\_i*: the ID of a stop
* lat: the latitude of a stop
* *lon*: the longitude of a stop
* *name*: the name of a stop

Une image contenant texte

Description générée automatiquement

*Fig.X: The* ***nodes*** *table*

**stop\_i** allows us to know the other columns, and, logically speaking, knowing the **latitude** and the **longitude** of a stop should give us both the **ID** and the **name** of it but it doesn’t mean the **ID** of a stop is dependent on its coordinates. In other words, the dependencies in this table are the following:

stop\_i -> lon, lat, name (primary key)

This means the **nodes** table is BCNF since stop\_i is a superkey of the table, which respects the condition ( for all dependencies a -> b in the list of functional dependcies of **routes**, a is superkey

The **routes** file stores the data of all the possible routes that exist in Paris, without worrying about the context (aka which stops are bound to said routes). It contains the following keys:

* *route\_i: the ID given to a route (*i.e a possible path between two stops by using a certain transport)
* *route\_type:* the transport ID that we use to go between two stops (0 is for trams, 1 is for metro, 2 is for trains, 3 is for buses… some other IDs exist but are not used in the Paris data, such as a Funicular ID or even a Boat ID)
* *route\_name:* the name that is given to said route (A, 14, T8…)

Une image contenant texte, équipement électronique, capture d’écran

Description générée automatiquement

*Fig.X: The* ***routes*** *table*

In this table, knowing **route\_i** gives **route\_type** and **route\_name**, making it a primary key. Knowing **route\_type** or **route\_**name alone does not allow to know any other key in the table, knowing (**route\_type,** **route\_name)** does NOT give **route\_i**, since we can have multiple transports of the same type and the same name from different places (Bus 116 for Epinay sur Orge and Bus 116 for Rosny-Bois-Perrier for example, are different bus lines)

This leaves us with the following dependency:

route\_i -> route\_type, route\_name

That means the **routes** table is in **BCNF** form, we only have one dependency and it’s a superkey of the table (we can derive it into route\_i -> route\_type; route\_i -> route\_name but route\_i would still be a superkey).

The **walk** file stores the data of all the possible routes one could have in Paris between two stops by walking. According to the data documentation, it will only show a relation between two stops if the distance between them is equal or shorter than 1km. It contains the following keys:

* *from\_stop\_i*: the departure stop of a route
* *to\_stop\_i*: the arrival stop of a route
* *d:* the straight-line distance between two stops, **we won’t use it**
* *d\_walk:* the distance we need to walk between said stops. We will divide each value by a human’s average walking speed, 1.5m^s-1 to get the average duration of a trip, giving us a value like *duration\_avg* contextually
* (self-made) *route\_i:* given that a walking route is not considered in the **routes** table, we give it the value ‘w’, as in ‘walk’.

Une image contenant texte

Description générée automatiquement

*Fig.X: The* ***walk*** *table*

Since the **route\_i** value doesn’t matter much here (there is only one path to walk form one stop to another), the only primary key we have is the composite key **(from\_stop\_i, to\_stop\_i)**. As a duration can be shared between multiple routes, the only dependency we have in this table is the following:

From\_stop\_i, to\_stop\_i-> duration\_avg, route\_i

If we derive this dependency to (From\_stop\_i, to\_stop\_i-> duration\_avg; From\_stop\_i, to\_stop\_i-> route\_i), (from\_stop\_i, to\_stop\_i) is still a superkey of all the dependencies, so we can say that the table **walk** is in BCNF form.

The **combined** file, as its name indicates, stores the data of all the possible routes one could have in Paris between two stops, no matter the transport used, instead of having to look up the routes for each transport separately (as it’s the case in any other network\_[mode].csv file). It contains the following keys:

* *from\_stop\_i*: the departure stop of a route
* *to\_stop\_i*: the arrival stop of a route
* *d:* the straight-line distance between two stops, **we won’t use it**
* *n\_vehicles*: the number of vehicles that took the rout, **we won’t use it**
* *duration\_avg:* the average duration of a trip between two stops on a certain route
* *route\_i: the ID given to a route (*i.e a possible path between two stops by using a certain transport)
* *route\_type*: the transport ID that we use to go between two stops, **we won’t use it** (we can get that value from the **routes** table)

The keys **from\_stop\_i, to\_stop\_i** and **route\_i** are *foreign:* **from\_stop\_i** and **to\_stop\_i** reference the **stop\_i** key from the **nodes** table, whereas **route\_i** references the eponym key from the **routes** table.

Une image contenant texte

Description générée automatiquement

*Fig.X: The* ***combined*** *table*

Knowing any key by itself isn’t enough to find the other values. For example, knowing **route\_i** doesn’t allow us to **from\_stop\_i** nor **to\_stop\_i**. There might me multiple routes that allow us to go from one stop to another, so the composite key (**from\_stop\_i, to\_stop\_**i) wouldn’t be enough to find **route\_**i. We can then then write the following dependencies:

From\_stop\_i, to\_stop\_i, route\_i -> duration\_avg

As we cannot derive this dependency, and it’s a superkey of the table, we can safely say that the table **combined** is in BCNF form.

The **temporal\_day** file contains the following columns:

* *from\_stop\_i*: the departure stop of a route
* *to\_stop\_i*: the arrival stop of a route
* *dep\_time\_ut*: the Unix time at which we take the route from the departure stop
* *arr\_time\_ut*: the Unix at which we are supposed to arrive to the arrival stop
* *route\_type*: the transport ID that we use to go between two stops, **we won’t use it**
* *trip\_I*: gives the ID given to a certain sequence of routes taken, **we won’t use it**
* *seq*: gives the sequence number within a trip, **we won’t use it**
* *route\_i: the ID given to a route (*i.e a possible path between two stops by using a certain transport)

Une image contenant texte, capture d’écran, moniteur

Description générée automatiquement

*Fig.X: The* ***temporal\_day*** *table*

This table is used to get the estimated duration of a trip between two stops. We can subtract the departure time from the arrival time to get a trip’s duration (in seconds).

This table has a composite *foreign key*, **(from\_stop\_i, to\_stop\_i, route\_i)**, which references **combined**’s *primary key*: both tables feature all the possible routes going from one stop to another within Paris, with the difference being that **combined** gives the average duration of a trip on said route, while **network\_day** gives the duration of a trip on said route depending on the current time (so it has way more values). Plus, the paper our data is based on describes the **network\_day** dataset as a *“temporal network extract listing the elementary PT connections, or events, that describe the progression of a PT vehicle from a stop to its next stop along the route”*, and, to make it even clearer, “*a listing of temporal network events for the specified Monday matching the other Monday-related data extracts*”. This means that a route between two stops would appear in **network\_day** only if it was featured in the other datasets (rail, tram, bus…), which are all contained within the **combined** table.

As such, we could also set **from\_stop\_i, to\_stop\_i** and **route\_i** as *foreign keys* from **nodes** and **routes**, but this relation is already defined in the **combined** table.

The *primary key* of this table is the composite key **(from\_stop\_i, to\_stop\_i, dep\_time\_ut, arr\_time\_ut, route\_i)**:

* we can go from a stop to multiple other stops, which means having **from\_stop\_i** or **to\_stop\_i** alone isn’t enough to know any other key in the table.
* the same route can be used to have a trip between multiple stops, so having **route\_i** does not allow us to know the departure or the destination stops.
* We can take multiple routes to go from a stop to another, so the composite key (**from\_stop\_i, to\_stop\_i)** isn’t sufficient to know other keys in the table.
* knowing the departure time alone does not allow us to know which route was taken, nor where it was from or to, neither when the arrival time would be. Thus, knowing **dep\_time\_ut** or **arr\_time\_ut** alone wouldn’t be of much use.
* The composite key (**dep\_time\_ut, arr\_time\_ut)** doesn’t tell much either
* we can take a route from one stop to another at different times so having **(from\_stop\_i, to\_stop\_i, route\_i)** alone wouldn’t be enough.
* We can also have multiple routes assigned to a same time so using **(from\_stop\_i, to\_stop\_i, dep\_time\_u, arr\_time\_ut)** wouldn’t suffice.
* We might be able to go from a stop to multiple others, during the same timelapse and using the same route. Same goes from a set of departure stops that would have the same arrival stop in the same timelapse using the same route, meaning that neither the **(from\_stop\_i, dep\_time\_ut, arr\_time\_ut, route\_i)** nor the (**to\_stop\_i, dep\_time\_ut, arr\_time\_u, route\_i)** are enough.

And so, it seems that the only dependency the **network\_day** table possesses is the combination of all the keys.

The functional dependencies are the following:

from\_stop\_i, to\_stop\_i, route\_i, dep\_time\_ut, arr\_time\_ut ->

from\_stop\_i, to\_stop\_i, route\_i, dep\_time\_ut, arr\_time\_ut (primary key)

Obviously, the **network\_day** table is in **BCNF** form since its functional dependencies are trivial. Since a -> b is trivial, this implies that the derived dependencies from a -> b will be trivial as well.

The **fiches-horaires-et-plans** and **referential-des-lignes** files store data about all the transport lines in Paris. It contains the following keys:

* *transportmode*: the type of transport related to a line; we can use it like **route\_type**
* *name\_line*: the name of the line
* *id\_line*: the internal ID of a line in the *IDF Mobilités* dataset, has NOTHING to do with any other ID of our tables, such as route\_i or stop\_i.
* *url*: the URL on which a plan related to the line is located.
* *type*: the type of document the url redirects to (can be a plan of the line or its schedule)**, we won’t use it** (as some schedules contain both the plan and the schedule, filtering the data would not be useful)
* *externalcode\_line****:*** external ID of a line in the IDF Mobilités dataset, **we won’t use it**
* *operatorref:* the ID of the line’s operator, **we won’t use it**
* *colourprint\_cmjn*: the colors of the line, in CMYK form, **we won’t use it**
* *textcolourprint\_hexa*: same, but in hexadecimal, **we won’t use it**
* *accessibility:* how accessible the line is, **we won’t use it**
* *audiblesigne\_available:* if the line has audible signs, **we won’t use it**

***etc, etc…***

Une image contenant texte

Description générée automatiquement

*Fig.X: the* ***lines*** *table*

It isn’t possible to know any other value in the table by knowing **route\_type** alone, same goes for the **url** (some plans show multiple lines at the same time) and **id\_line** ( the same **id\_line** can have multiple **url** values: one for the plan, the other for the schedule).

If we know the **url** and **id\_line**, we can find **route\_type** and **name\_line** (a certain transport line can only have one url associated to it)**.**

Knowing **(url, name\_line)** or even **(url, name\_line, route\_type)** isn’t enough to know **id\_line**, we can imagine the case of two lines “XX” lines of the same transport mode hosted on a main website where the url would redirect to the main page (example: the table contains two lines named *TER Hauts-de-France* which have the same *route\_type* and redirect to https://ter.sncf.com/hauts-de-france).

As such, the only dependency we have is the following:

id\_line, url -> route\_type, name\_line (primary key)

Once again, we can safely conclude this table is in **BCNF** form.

Now that we know the characteristics of our entity tables, we can derive the relations from our **E-R Diagram**:

* The relation *Used Nodes* between the tables **nodes** and **combined** don’t require the creation of a new table, since the values **from\_stop\_i** and **to\_stop\_i** from the **combined** table are already shown to be related to the **nodes** table with the *foreign key* we’ve found.
* The relation *Used Routes* between the tables **routes** and **combined** don’t require the creation of a new table, since the key **route\_i** from the **combined** table is already related to the **routes** table with the *foreign key* explained above.
* The relation *Route Duration* between the tables **temporal\_day** and **combined** don’t require the creation of a new table, since the combinated value (**from\_stop\_i** ,**to\_stop\_i, route\_i)** from the **temporal\_day** table are already shown to be related to the **combined** table with the *foreign key*.
* The relation *Route Plan* between the tables **routes** and **lines** cannot be done in a practical way, as the IDs used in both tables don’t match. We can do guess work between both by relying on the composite key **(route\_type, route\_name)** but, as we’ve discussed before, the obtained tuples are not unique.
* The relation *Shortest Routes* between the tables **walk** and **combined** can be done by concatenating the values of both tables in one: the resulting table would give us the list of all the routes one can take, even by walking or with any transport, as well as the duration of a trip on said route. This was done by **INSERT**ing the values of both table into a new table called **shortest\_route**.

The **shortest\_route** table’s primary key is ( **from\_stop\_i, to\_stop\_i, route\_i)** since it takes values from the **combined** table, and given that the primary key of the table **walk** is (**from\_stop\_i, to\_stop\_i)**.

Because this table is a combination of the values of two other tables that share the same keys with a different dataset, it’s not possible to have a foreign key about any value (particularly because the data from the **walk** table isn’t compatible with one from **nodes** or **route**).

It’s the only dependency of the table, so this new table is in **BCNF** form.

III – SQL Queries related to each functionality

* Creating the tables:

create table nodes (

stop\_i numeric ,

lat numeric,

lon numeric,

name text,

PRIMARY KEY (stop\_i)

);

create table routes(

route\_type NUMERIC,

route\_name TEXT,

route\_i NUMERIC,

PRIMARY KEY (route\_i)

);

create table walk (

from\_stop\_i numeric,

to\_stop\_i numeric,

duration\_avg numeric,

route\_i text,

PRIMARY KEY (from\_stop\_i, to\_stop\_i)

);

create table combined(

from\_stop\_i numeric,

to\_stop\_i numeric,

duration\_avg numeric,

route\_i numeric,

PRIMARY KEY (from\_stop\_i, to\_stop\_i, route\_i),

FOREIGN KEY (route\_i) references routes(route\_i),

FOREIGN KEY (from\_stop\_i) references nodes(stop\_i),

FOREIGN KEY (to\_stop\_i) references nodes(stop\_i)

);

create table temporal\_day(

from\_stop\_i numeric,

to\_stop\_i numeric,

dep\_time\_ut numeric,

arr\_time\_ut numeric,

route\_type numeric ,

seq numeric ,

route\_i numeric,

PRIMARY KEY (from\_stop\_i, to\_stop\_i, dep\_time\_ut, arr\_time\_ut, route\_i),

FOREIGN KEY (from\_stop\_i, to\_stop\_i, route\_i)

references combined(from\_stop\_i, to\_stop\_i, route\_i)

);

* Inserting the values from pandas into sql:

df.to\_sql(table, con=engine, if\_exists='append', index=False)

try:

cursor.execute(f"""select \* from {table}""")

print(f"successfully copied the dataframe into {table}")

except (Exception, psycopg2.DatabaseError) as error:

print("Error: %s" % error)

conn.rollback()

cursor.close()

return 1

* Creating the shortest\_route table on the go:

cursor.execute("""SELECT \* into shortest\_route

from walk;

insert into shortest\_route

select \* from combined; """)

conn.commit()

cursor.execute("""alter table shortest\_route

add primary key(from\_stop\_i, to\_stop\_i, route\_i)""")

conn.commit()

* Creating the lines table on the go:

with open(os.path.join(data\_path, 'referentiel-des-lignes.json'), 'r') as f:

json\_file = json.load(f)

data = [[line['fields']['transportmode'], line['fields']['id\_line'], line['fields']['name\_line']] for line

in

json\_file]

lines = pd.DataFrame(data, columns=['transportmode', 'id\_line', 'name\_line'])

with open(os.path.join(data\_path, 'fiches-horaires-et-plans.json'), 'r') as f:

json\_file = json.load(f)

data = [[line['fields']['id\_line'], line['fields']['url']] for line in json\_file]

urls = pd.DataFrame(data, columns=['id\_line', 'url'])

merge = pd.merge(lines, urls, how='inner', on=["id\_line"])

merge = merge.drop\_duplicates()

route\_merge = merge['transportmode'].values

for route\_x in range(len(route\_merge)):

route\_merge[route\_x] = int(route\_type.str\_route\_num(route\_merge[route\_x]))

merge['transportmode'] = route\_merge

merge = merge.rename(columns={'transportmode': 'route\_type'})

merge.to\_sql('lines', con=engine, if\_exists='replace', index=False)

cursor.execute("""ALTER TABLE lines

ADD PRIMARY KEY (url, id\_line)""")

conn.commit()

* Deleting the previous tables if we want to parse the data again:

try:

try:

cursor.execute('DROP SCHEMA public CASCADE')

conn.commit()

except psycopg2.ProgrammingError:

conn.rollback()

cursor.execute('CREATE SCHEMA public')

conn.commit()

cursor.execute('GRANT ALL ON SCHEMA public TO postgres')

conn.commit()

cursor.execute('GRANT ALL ON SCHEMA public TO public')

conn.commit()

except psycopg2.ProgrammingError:

conn.rollback()

cursor.execute('DROP owned by l3info\_32')

conn.commit()

* Checking if the database is filled on the program’s launch, filling the Combo Boxes with the nodes list:

def connect\_DB(self):

print("database project connected to server")

self.cursor = self.conn.cursor()

try:

self.cursor.execute("""SELECT distinct name FROM nodes ORDER BY name""")

self.conn.commit()

rows = self.cursor.fetchall()

for row in rows:

self.from\_box.addItem(str(row[0]))

self.to\_box.addItem(str(row[0]))

except psycopg2.ProgrammingError as e:

self.conn.rollback()

err = QtWidgets.QMessageBox()

err.setIcon(QtWidgets.QMessageBox.Warning)

err.setText("La table n'existe pas! récupération des données en cours.")

err.exec\_()

parse.main()

self.connect\_DB()

return

* Getting the ID of a node depending on the value in the Combo Box:

self.to\_stop\_i = str(self.to\_box.currentText()).replace("'", "''")

self.cursor.execute(f""" SELECT stop\_i FROM nodes WHERE name = '{self.to\_stop\_i}'""")

self.conn.commit()

myrows = self.cursor.fetchall()

self.to\_stop\_i = int(myrows[0][0])

* Getting the nodes table, the routes table, the shortest routes list and storing it inside python dictionaries:

self.nodes = pd.read\_sql("SELECT \* FROM \"{}\";".format("nodes"), self.engine)

self.routes = pd.read\_sql("SELECT \* FROM \"{}\";".format("routes"), self.engine)

super\_short\_comb\_walk = pd.read\_sql("SELECT \* FROM \"{}\";".format("shortest\_route"), self.engine)

IV – Encountered Difficulties & Contributions per member

Mohamed:

Jean-Philipp:

*My main duty was deriving the tables from the ER diagram and optimizing our general code. Before I started working on the code, we had a functional application, but the database had unoptimized tables with no foreign keys declared, leading to having tables that occupied space for no reason (such as a table that had values inserted from both nodes and routes to know the names of each stop in a route). After finding the foreign keys of each table, and eliminating the tables that turned out to be of no use, I reduced the total amount of tables from 12 to 6.*

*I also improved the UX/UI side of our project: I am the one who implemented the credentials prompt to connect to the database, as well as a notification that tells the user when the database is yet to be filled. I also fixed the behavior of the table widget, which did not reset its values properly before, and kept on the screen the stops from the first trip a user requested.*

*Lastly, I am the one who found out about IDF Mobilités having a dataset of transport plans related to Paris. I did not implement it within our project, but I helped Lao Cam understand the structure of each .json file.*

Cao Lam: