Report

Assignment 2 - MySQL

Group: 19
Students: Sara Cinquini, Lucie Perez, Victoria Stasik

Introduction

In the context of our course entitled “Very large, distributed data volumes”, our assignment presented the following challenge: to design and implement an SQL database, create relevant tables, populate them with given data, and to finish, perform complex queries on this dataset. This assignment required not only technical skills but also collaborative capabilities as we had to work in groups.

In our collaborative journey, the group worked closely together for the task concerning the creation and insertion of the data into the tables, as this task was the most crucial for the success of the entire project. As a team, we had many discussions to find the best approach for handling the huge amount of data. It allowed us to create a well-structured database that was ready for the “query task”. For the last task, we equally distributed the work among the team members. This division allowed each member to focus on specific queries. However, throughout the assignment, we were here to support one another in case of any need of help.

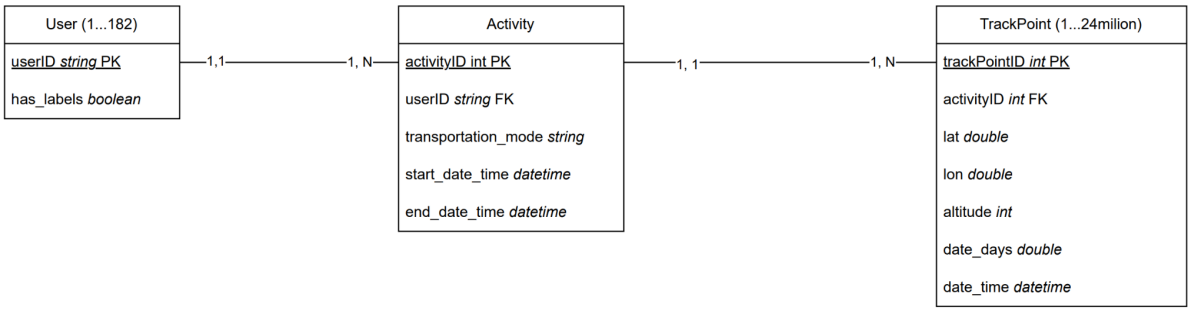
This report outlines our solution with the different strategies we employed.

Link to the Git repository :
https://github.com/stasikvictoria/Very_large_distributed_data_volumes_exercise2.git

Results

Task 1

The first task consisted of creating the tables and inserting the data from the dataset, while respecting the different constraints that were given. Here is a reminder of how the data looks like :



To solve this task, we decided to use Pandas dataframes for each of the three tables, as it is much easier to manipulate data frames to make computations on tables.

User table

The User table is the easiest to create as it has only two columns and all the information we need is in the labeled_ids.txt file which contains the ids of the users that have labeled activities. To create the table we've created the create_table_user function which executes a simple SQL query.

```
def create_table_user(self):
    query = """CREATE TABLE IF NOT EXISTS %s (
                userID VARCHAR(30) NOT NULL PRIMARY KEY,
                has_labels BOOLEAN)
            """
    # this adds table_name to the %s variable and executes the query
    self.cursor.execute(query % self.USER)
    self.db_connection.commit()
```

To insert the data, we've created a second function called insert_data_user. This function saves all user ids into a list using the names of all user directories. Then it uses the labeled_ids.txt file to check which user has labeled activities and which has not. This information is saved in another simple list called has_labels (the indexes in both lists are corresponding to one another). The final step is the conversion of both lists into one single dataframe and we use this dataframe to insert all the information into the User table.

```python
def insert_data_user(self):
    print('*** INSERT IN USER ***')
    # we go to the Data directory
    os.chdir('/Users/victoriastasik/Documents/Very_large_distributed_data_volumes_exercise2/dataset/dataset/Data')

    # we get the list of all the diferent directory names (users ids) and we sort the list
    usersID =  os.listdir()
    usersID.sort()

    # we go back to the dataset directory and we read the labeled_ids.txt file
    os.chdir("..")
    with open('labeled_ids.txt') as f:
        labeled_ids = f.readlines()
    f.close()
    os.chdir('/Users/victoriastasik/Documents/Very_large_distributed_data_volumes_exercise2/dataset/dataset/Data')

    # # we delete the \n in each string
    for i in range (0, len(labeled_ids)):
        labeled_ids[i] = labeled_ids[i].strip()

    # we check if each user has a label or not and we save the info in a list
    # the indexes of has_labels and id lists are correponding
    has_labels = []
    for i in usersID :
        if i in labeled_ids :
            has_labels.append(True)
        else:
            has_labels.append(False)

    user_table = {'userID': usersID, 'has_labels': has_labels}
    user_dataframe = pd.DataFrame(user_table)
    user_data = user_dataframe.to_records(index=False).tolist()
    query = "INSERT INTO {} (userID, has_labels) VALUES (%s, %s)".format(self.USER)
    self.cursor.executemany(query, user_data)
    self.db_connection.commit()
```

Activity and TrackPoint

The Activity and TrackPoint tables are more challenging to create as there is a lot of information in a lot of different files and also, there are a lot of constraints to take into account. First we've created two functions create_table_trackpoint and create_table_activity with simple SQL queries. The function to create the Activity table is a little bit different here because we added a constraint to check if the start_date_time attribute takes place before the end_date_time attribute.

```python
def create_table_trackpoint(self):
    query = """CREATE TABLE IF NOT EXISTS %s (
                    trackPointID INT NOT NULL,
                    activityID int NOT NULL,
                    lat double NOT NULL,
                    lon double NOT NULL,
                    altitude int NOT NULL,
                    date_days double NOT NULL,
                    date_time datetime NOT NULL,
                    PRIMARY KEY (trackPointID))

                """
    # FOREIGN KEY (activityID) REFERENCES Activity(activityID))
    # this adds table_name to the %s variable and executes the query
    self.cursor.execute(query % self.TRACK_POINT)
    self.db_connection.commit()
```

```
def create_table_activity(self):
    query = """CREATE TABLE IF NOT EXISTS %s (
                activityID int NOT NULL,
                userID varchar(30) NOT NULL,
                transportation_mode varchar(30),
                start_date_time datetime NOT NULL,
                end_date_time datetime NOT NULL,
                PRIMARY KEY (activityID, transportation_mode)

            )"""
    # FOREIGN KEY (userID) REFERENCES User(userID)
    # this adds table_name to the %s variable and executes the query
    self.cursor.execute(query % self.ACTIVITY)
    self.db_connection.commit()

    query_constr = """ ALTER TABLE %s
                ADD CONSTRAINT check_start_end_dates
                CHECK (start_date_time < end_date_time)
            """

    self.cursor.execute(query_constr % self.ACTIVITY)
    self.db_connection.commit()
```

We've also added another function to create foreign key constraints for the userID and activityID attributes. We are calling this function after inserting all the data in our tables to check if the userIDs present in the Activity table are also present in the User table and if the activityIDs present in the TrackPoint table are also present in the Activity table.

```
def altering_tables_with_foreign_key(self):

    query1 = """ALTER TABLE Activity
            ADD CONSTRAINT fk_activity
            FOREIGN KEY (userID)
            REFERENCES User(userID);"""
    self.cursor.execute(query1)
    self.db_connection.commit()

    query2 = """ALTER TABLE TrackPoint
            ADD CONSTRAINT fk_trackpoint
            FOREIGN KEY (activityID)
            REFERENCES Activity(activityID);"""
    self.cursor.execute(query2)
    self.db_connection.commit()
```

To get all the information from the labels.txt files we've created a special function called get_info_from_labels_txt_files that goes through all our directories and returns 4 lists:

- user_id_list,
- transportation_mode_list,
- start_date_time_list
- end_date_time_list.

```python
def get_info_from_labels_txt_file() :
    user_id_list = []
    transportation_mode_list = []
    start_date_time_list = []
    end_date_time_list = []

    for dirpath, dirnames, filenames in os.walk(os.getcwd()):
        for filename in filenames:
            # if there is a labels.txt file, we save the info
            if filename.endswith('.txt'):
                try :
                    with open(os.path.join(dirpath, filename)) as f:
                        lines = f.readlines()

                        # skip the header
                        lines = lines[1:]

                        for line in lines:

                            # we get the info of one line
                            data = line.split()

                            # we save each information into the correct list
                            last_directory_name = os.path.basename(dirpath)
                            user_id_list.append(last_directory_name)

                            transportation_mode_list.append(data[4])

                            start_date = data[0]
                            start_time = data[1]
                            end_date = data[2]
                            end_time = data[3]
                            start_datetime_str = start_date + " " + start_time
                            end_datetime_str = end_date + " " + end_time
                            start_datetime_str = start_datetime_str.replace('/', '-')
                            end_datetime_str = end_datetime_str.replace('/', '-')
                            combined_start_datetime = datetime.strptime(start_datetime_str, "%Y-%m-%d %H:%M:%S")
                            combined_end_datetime = datetime.strptime(end_datetime_str, "%Y-%m-%d %H:%M:%S")

                            start_date_time_list.append(combined_start_datetime)
                            end_date_time_list.append(combined_end_datetime)
                    f.close()

                # error handling
                except FileNotFoundError:
                    print(f"the file {filename} doesn't exist.")
                except Exception as e:
                    print(f"Error: {e}")

    return user_id_list, transportation_mode_list, start_date_time_list, end_date_time_list
```

We have the exact same strategy for all the plt files. Indeed, we've created a function called get_info_from_plt_files that goes through all directories and returns 9 lists : activity_id_list, user_id_list, lat_list, lon_list, altitude_list, date_days_list, current_date_time_list, start_date_time_list and end_date_time_list. The current_date_time_list corresponds to all the different dates written at the end of each line of the plt files (date of the specific trackpoint), while start_date_time_list and end_date_time_list correspond to the first date and the last date of each plt file (start and end of one activity). Once again all indexes of all these lists are corresponding to one another. In this function, we are also dealing with the plt file length constraint. Indeed, before getting the information of a plt file, the code checks if the length of the plt file is less or equal to 2500.

```python
def get_info_from_plt_file() :
    activity = 0
    activity_id_list = []
    user_id_list = []
    lat_list = []
    long_list = []
    altitude_list = []
    date_days_list = []
    current_date_time_list = []
    start_date_time_list = []
    end_date_time_list = []

    for dirpath, dirnames, filenames in os.walk(os.getcwd()):
        for filename in filenames:

            # we get the information of each plt file
            if filename.endswith('.plt'):
                try :
                    with open(os.path.join(dirpath, filename)) as f:
                        lines = f.readlines()

                        # skip the first 6 lines
                        lines = lines[6:]

                        # check the length of the plt file
                        if len(lines) <= 2500:

                            # we get the start and end date of each plt file
                            start_line= lines[0].split(',')
                            start_date = start_line[5]
                            start_time = start_line[6]
                            start_datetime = start_date + ' ' + start_time
                            start_datetime = start_datetime.rstrip('\n')
                            start_datetime = datetime.strptime(start_datetime, "%Y-%m-%d %H:%M:%S")

                            end_line = lines[len(lines)-1].split(',')
                            end_date = end_line[5]
                            end_time = end_line[6]
                            end_datetime = end_date + ' ' + end_time
                            end_datetime = end_datetime.rstrip('\n')
                            end_datetime = datetime.strptime(end_datetime, "%Y-%m-%d %H:%M:%S")

                            # we save the information of each line of the plt file
                            for line in lines:

                                data = line.split(',')

                                activity_id_list.append(activity)

                                parent_directory = os.path.dirname(dirpath)
                                directory_name = os.path.basename(parent_directory)
                                user_id_list.append(directory_name)

                                lat_list.append(float(data[0]))
                                long_list.append(float(data[1]))
                                altitude_list.append(int(float(data[3])))
                                date_days_list.append(float(data[4]))

                                date = data[5]
                                time = data[6]
                                datetime_draft = date + ' ' + time
                                datetime_draft = datetime_draft.rstrip('\n')
                                combined_datetime = datetime.strptime(datetime_draft, "%Y-%m-%d %H:%M:%S")
                                current_date_time_list.append(combined_datetime)

                                start_date_time_list.append(start_datetime)
                                end_date_time_list.append(end_datetime)

                        f.close()
                        activity +=1

                    except FileNotFoundError:
                        print(f"the file {filename} doesn't exist.")
                    except Exception as e:
                        print(f"Error: {e}")

    return activity_id_list, user_id_list, lat_list, long_list, altitude_list, date_days_list, current_date_time_list, start_date_time_list, end_date_time_list
```

To finally create our activity dataframe and trackpoint dataframe to be able to insert the data into the SQL tables, we wrote the creating_activity_and_trackpoint_dataframe function. This function uses both previous functions to create the two dataframe. Note that for the activity table, we've decided to put the string "missing" for the activities without transportation mode.

```python
def creating_activity_and_trackpoint_dataframe():
    # getting info from txt files
    user_id_list, transportation_mode_list, start_date_time_list, end_date_time_list = get_info_from_labels_txt_file()
    labels_txt = {'user_id': user_id_list, 'transportation_mode': transportation_mode_list, 'start_datetime': start_date_time_list, 'end_datetime': end_date_time_list}
    labels_txt_df = pd.DataFrame(labels_txt)

    # getting info from plt files
    activity_id_list, user_id_list_2, lat_list, long_list, altitude_list, date_days_list, current_date_time_list, start_date_time_list2, end_date_time_list2 = get_info_from_plt_file()
    plt = {'activity_id': activity_id_list, 'user_id': user_id_list_2, 'lat': lat_list, 'long': long_list, 'altitude': altitude_list, 'date_days': date_days_list,
           'current_date_time': current_date_time_list, 'start_datetime': start_date_time_list2, 'end_datetime': end_date_time_list2}
    plt_df = pd.DataFrame(plt)

    # merging both dataframes
    merged_df = pd.merge(plt_df, labels_txt_df, on=['user_id', 'start_datetime', 'end_datetime'], how='left')

    # creating activity table
    activity_table = merged_df[['activity_id','user_id', 'transportation_mode', 'start_datetime', 'end_datetime']]
    activity_table = activity_table.fillna("missing")
    activity_table['start_datetime'] = activity_table['start_datetime'].astype(str) # converting into string to be able to insert into the sql table
    activity_table['end_datetime'] = activity_table['end_datetime'].astype(str)
    activity_table = activity_table.drop_duplicates()

    # creating trackpoint table
    trackpoint_table = merged_df[['activity_id','lat', 'long', 'altitude', 'date_days', 'current_date_time']]
    trackpoint_table.rename(columns={'current_date_time': 'date_time'}, inplace=True)
    trackpoint_table['id'] = range(1, len(trackpoint_table) + 1)
    trackpoint_table = trackpoint_table[['id'] + [col for col in trackpoint_table.columns if col != 'id']]
    trackpoint_table['date_time'] = trackpoint_table['date_time'].astype(str)

    return activity_table, trackpoint_table
```

To insert the data into our SQL tables we've created two more functions called insert_data_trackpoint and insert_data_activity. In these functions, we use batches to insert our data as there are a lot of rows in our dataframes.

```python
def insert_data_trackpoint(self, trackpoint_dataframe):
    print('*** INSERT IN TRACKPOINT ***')
    trackpoint_data = trackpoint_dataframe.to_records(index=False).tolist()
    query = "INSERT INTO {} (trackpointID, activityID, lat, lon, altitude, date_days, date_time) VALUES (%s, %s, %s, %s, %s, %s, %s)".format(self.TRACK_POINT)

    # Batch insert (e.g., insert 100 rows at a time)
    batch_size = 1000
    for i in range(0, len(trackpoint_data), batch_size):
        batch_data = trackpoint_data[i : i+batch_size]
        self.cursor.executemany(query, batch_data)
        self.db_connection.commit()
```

```python
def insert_data_activity(self, activity_dataframe):
    print('*** INSERT IN ACTIVITY ***')
    activity_data = activity_dataframe.to_records(index=False).tolist()
    query = "INSERT INTO {} (activityID, userID, transportation_mode, start_date_time, end_date_time) VALUES (%s, %s, %s, %s, %s)".format(self.ACTIVITY)

    # Batch insert (e.g., insert 100 rows at a time)
    batch_size = 1000
    for i in range(0, len(activity_data), batch_size):
        batch_data = activity_data[i : i+batch_size]
        self.cursor.executemany(query, batch_data)
        self.db_connection.commit()
```

At the end, here is how our tables look like :
[note, the output was truncated]

userID	has_labels
000	0
001	0
002	0
003	0
004	0
005	0
006	0
007	0
008	0
009	0
010	1

```
SELECT ACTIVITY
  activityID    userID  transportation_mode    start_date_time      end_date_time
------------  --------  ---------------------  -------------------  -------------------
           0       135  missing                2009-01-03 01:21:34  2009-01-03 05:40:31
           1       135  missing                2009-01-02 04:31:27  2009-01-02 04:41:05
           2       135  missing                2009-01-27 03:00:04  2009-01-27 04:50:32
           3       135  missing                2009-01-10 01:19:47  2009-01-10 04:42:47
           6       135  missing                2009-01-14 12:17:57  2009-01-14 12:30:53
           7       135  missing                2009-01-12 01:41:22  2009-01-12 02:14:01
          10       135  missing                2008-12-24 14:42:07  2008-12-24 15:26:45
          11       135  missing                2008-12-28 10:36:05  2008-12-28 12:19:32
          13       132  missing                2010-02-15 10:56:35  2010-02-15 12:22:33
          15       132  missing                2010-04-30 23:38:01  2010-05-01 00:35:31
```

```
  trackPointID    activityID      lat    lon    altitude    date_days  date_time
--------------  ------------  -------  -----  ----------  -----------  -------------------
             1             0  39.9743  116.4         492      39816.1  2009-01-03 01:21:34
             2             0  39.9743  116.4         492      39816.1  2009-01-03 01:21:35
             3             0  39.9743  116.4         492      39816.1  2009-01-03 01:21:36
             4             0  39.9743  116.4         492      39816.1  2009-01-03 01:21:38
             5             0  39.9744  116.4         491      39816.1  2009-01-03 01:21:39
             6             0  39.9744  116.4         491      39816.1  2009-01-03 01:21:42
             7             0  39.9744  116.4         491      39816.1  2009-01-03 01:21:46
             8             0  39.9745  116.4         491      39816.1  2009-01-03 01:21:51
             9             0  39.9745  116.4         490      39816.1  2009-01-03 01:21:56
            10             0  39.9745  116.4         489      39816.1  2009-01-03 01:22:01
```

Task 2

[Note, all the tables are limited to 10 lines, but the time execution doesn't consider this limitation.]

QUERY 1 How many users, activities and trackpoints are there in the dataset (after it is inserted into the database).

```
query1 = """
        SELECT
            (SELECT COUNT(*) FROM User) AS TotUser,
            (SELECT COUNT(*) FROM Activity) AS TotActivity,
            (SELECT COUNT(*) FROM TrackPoint) AS TotTrackPoint;
        """
```

Time execution: 2.5s

```
QUERY 1
  TotUser    TotActivity    TotTrackPoint
---------  -------------  ---------------
      182          16049          9682005
```

Explanation *- for this query it was only necessary to use the function COUNT() to identify the number of elements for each table.*

QUERY 2 Find the average, maximum and minimum number of trackpoints per user.

```
query2 =  """
        SELECT
            userID,
            avg(nb_tp_per_activity) as mean_trackpoint,
            max(nb_tp_per_activity) as max_trackpoint,
            min(nb_tp_per_activity) as min_trackpoint
        FROM (select count(*) as nb_tp_per_activity, activityID
              FROM TrackPoint as tp
              GROUP BY activityID) tp
        LEFT JOIN Activity a
        ON tp.activityID = a.activityID
        GROUP BY userID
        ORDER BY userID;
    """
```

Time execution: 9.5s

```
QUERY 2
  userID    mean_trackpoint    max_trackpoint    min_trackpoint
--------  -----------------  ----------------  ----------------
     000           670.871              2359                 5
     001           824.965              2472                33
     002           924.801              2438                 4
     003           807.387              2485                 3
     004           761.858              2482                 4
     005           587.548              2058                 5
     006           801.125              2478                14
     007           758.275              2228                 6
     008          1376                  2499               165
     009          1218.32               2396               134
```

Explanation *- For this query, we need to compute first the number of trackpoint per query so we use a subquery which creates a table tp with activityID and the number of trackpoint. Then we can compute the aggregate by joining tp and the activity table so we have the information about the user. The "order by" is not needed to give the result and it will take more time with this clause but for visualization it is better.*

QUERY 3 Find the top 15 users with the highest number of activities.

```
query3 = """
        SELECT userID, count(activityID) AS activity_nb
        FROM Activity
        GROUP BY userID
        ORDER BY activity_nb DESC LIMIT 15;
        """
```

Time execution: 0.1s

```
QUERY 3
  userID    activity_nb
--------  -------------
     128           2102
     153           1793
     025            715
     163            704
     062            691
     144            563
     041            399
     085            364
     004            346
     140            345
```

Explanation *- for this task, we count the number of activities using COUNT associated with a GROUP BY on userID. The table we need is only the Activity table. At the end we order the result using DESC and we take the top 15 users using LIMIT 15.*

QUERY 4 Find all users who have taken a bus.

```
query4 =  """
        SELECT DISTINCT userID
        FROM Activity
        WHERE transportation_mode = 'bus'
        ORDER BY LOWER(userID);
        """
```

Time execution: 0.1s

```
QUERY 4
  userID
--------
     010
     052
     062
     073
     081
     084
     085
     091
     092
     112
     125
     128
     175
```

Explanation *- knowing that in the table Activity is possible to find the "transportation_mode" and also the "userID", we selected all the (distinct) users that have transportation_mode = 'bus'. For us it was not necessary to execute a JOIN operation in the user table, because the information that we needed was also present in the Activity table.*

QUERY 5 List the top 10 users by their amount of different transportation modes.

```
query5 = """
    SELECT userID,
    COUNT(transportation_mode) as nb_transportation_mode
    FROM(
        SELECT
            userID,
            transportation_mode
        FROM Activity
        GROUP BY UserID,transportation_mode) t
    GROUP BY userID
    ORDER BY nb_transportation_mode DESC
    LIMIT 10;
"""
```

Time execution: 0.1s

```
QUERY 5
  userID    nb_transportation_mode
--------  ------------------------
     128                        10
     062                         8
     085                         5
     058                         4
     078                         4
     081                         4
     084                         4
     112                         4
     163                         4
     010                         3
```

Explanation *- For this query, we only need the activity table. First we need to group by on the user and the transprtation_mode. So we have a table t with only user and different transprtation_mode. Then we group by userID so we have the number of transportation_mode by user. We want the maximum of transportation modes so we order by DESC and limit the result to the top 10.*
In this case, we count the value 'missing' as one transportation mode but we can change that by adding a where clause after the group by transportation_mode, userID HAVING transportation_mode<>"missing" in the subquery t.

QUERY 6 Find activities that are registered multiple times. You should find the query even if it gives zero result.

```
query6 = """
    SELECT
        A.userID,
        A.transportation_mode,
        A.start_date_time,
        A.end_date_time,
        COUNT(*) AS registration_nb
    FROM Activity A
    GROUP BY
        A.userID,
        A.transportation_mode,
        A.start_date_time,
        A.end_date_time HAVING COUNT(*) > 1;
"""
```

Time execution: 0.2s

```
QUERY 6
userID    transportation_mode    start_date_time    end_date_time    registration_nb
--------  ---------------------  -----------------  ---------------  -----------------
```

Explanation *- we consider that two activities are the same when the userID, the transportation_mode, the start_date_time and end_date_time are the same. We count the same activities using COUNT*

associated with a GROUP BY on all previous attributes. The COUNT must be > 1 (there are at least two identical activities).

QUERY 7

a) Find the number of users that have started an activity in one day and ended the activity the next day.

```
#info: DATE extracts the "date" part
query7_a = """
        SELECT COUNT(DISTINCT userID) as tot_users_with_activity
        FROM Activity
        WHERE
                DATE(start_date_time) != DATE(end_date_time)
                and
                DATEDIFF(end_date_time, start_date_time) = 1;
        """
```

Time execution: 0.1s

```
QUERY 7 A
  tot_users_with_activity
-------------------------
                      98
```

Explanation *- For this query was required to count all the users that have started an activity in one day and ended in the next one. So, it was necessary to use the function DATEDIFF() that returns the difference between two dates (in our case the difference was 1). Using COUNT(DISTINCT …) was possible to count the total of the users.*

b) List the transportation mode, user id and duration for these activities.

```
query7_b = """
    SELECT
        userID,
        transportation_mode,
        TIMESTAMPDIFF(MINUTE, start_date_time, end_date_time) as duration
    FROM Activity
    WHERE
        DATE(start_date_time) != DATE(end_date_time)
        and
        DATEDIFF(end_date_time, start_date_time) = 1
    ORDER BY userID ASC, duration DESC;
"""
```

Time execution: 0.3

```
QUERY 7 B
  userID  transportation_mode      difference
--------  --------------------  ------------
     132  missing                         57
     104  missing                        155
     104  missing                        531
     104  missing                        608
     104  missing                        148
     104  missing                        723
     103  missing                         39
     103  missing                         51
     103  missing                         59
     168  missing                        445
     168  missing                        187
     168  missing                         41
     168  missing                         62
     168  missing                         73
     168  missing                         16
     168  missing                         27
     168  missing                        227
```

Explanation *- this query was very similar to the previous one, but the request was related also to the selection of the userID, transportation mode and the duration. To calculate the duration was necessary to use another function, called TIMESTAMPDIFF(unit,expr1,expr2) that takes in input*

- ***unit*** *It denotes the unit for the result. It can be one: MICROSECOND, SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER, YEAR. As a unit we decided to use MINUTE.*
- ***expr1****, first date or DateTime expressions.*
- ***expr2****, second date or DateTime expressions.*

QUERY 8 Find the number of users which have been close to each other in time and space. Close is defined as the same space (50 meters) and for the same half minute (30 seconds)

```
SELECT
    user1,
    user2,
    COUNT (
        CASE WHEN (ST_Distance(point(long1, lat1),
        point(long2, lat2)) <= 50)
        and (abs(TIMESTAMPDIFF(Second,date1,date2))<30)
        THEN 1 ELSE null end) as nb
FROM (
    SELECT
        tp1.userID as user1,
        tp1.lat as lat1,
        tp1.lon as long1,
        tp1.date_time as date1,
```

```
        tp2.userID as user2,
        tp2.lat as lat2,
        tp2.lon as long2,
        tp2.date_time as date2
    FROM (
        SELECT a.userID,tp.lat, tp.lon, tp.date_time
        FROM TrackPoint tp
        JOIN Activity a
        ON a.activityID=tp.activityID)tp1
        JOIN (
            SELECT a.userID, tp.lat, tp.lon,tp.date_time
            FROM TrackPoint tp
            JOIN Activity a
            ON a.activityID=tp.activityID) tp2
            ON date(tp1.date_time) = date(tp1.date_time)
            WHERE tp1.userID<tp2.userID) tp_compare
GROUP BY user1,user2;
```

Explanation *- For finding users who have been close to each other in time and space, we need to compare each trackpoint with another. We have more than 9 million trackpoints so we need to make* 8.10^{10} *comparisons which is a lot. The query is very simple but not very efficient and our computer can't give us the result in a decent amount of time.*
We join the trackpoint on itself on the date for keeping only trackpoint with the same date. Then we compute the distance with the function ST_Distance() and the time difference with the function timestampdiff(). Then we group by users to keep only the pair of users who have been close to each other.

We can reduce the trackpoint by comparing only the trackpoint which are the same day for different users.

QUERY 9 Find the top 15 users who have gained the most altitude meters.

```
SET @prev_userID = NULL;
SET @prev_altitude = NULL;

SELECT userID,  SUM(gained_altitude) AS total_altitude
FROM (
      SELECT userID, altitude,
      CASE
          WHEN @prev_userID IS NULL OR userID <> @prev_userID THEN 0
          WHEN altitude > @prev_altitude THEN altitude - @prev_altitude
          ELSE 0
          END AS gained_altitude,
          @prev_userID := userID,
          @prev_altitude := altitude
          FROM TrackPoint
          JOIN Activity ON TrackPoint.activityID = Activity.activityID
          ORDER BY userID, trackPointID) AS Tab
GROUP BY userID ORDER BY total_altitude DESC LIMIT 15;
```

userID	total_altitude
128	2591138
153	2229827
004	1171993
041	946361
062	894650
144	893494
003	819353
163	815877
085	790323
030	604144
039	529142
025	517369
084	478627
140	453499
167	433619

Time execution: 12 min 40,45 s

Explanation *- for this query we use SQL variables prev_userID and prev_altitude to keep in memory those information to be able to compute the gained altitude per user. First, we are creating a table called Tab containing the userID, the altitude and the gained_altitude. We create this table using the TrackPoint table and the Activity table (JOIN) ordered by userID and trackPointID. If the id of the previous user is NULL or the id of the previous user is different from the id of the current one, then the gained_altitude equals 0. If the current altitude is bigger than the previous altitude then the gained_altitude is equal current altitude - previous altitude. Then, the previous userID and the previous altitude become the current userID and altitude. Once we have the Tab table, we compute the SUM of all gained altitudes for all users and we order the result using DESC.*

QUERY 10 Find the users that have traveled the longest total distance in one day for each transportation mode.

```
SELECT t.userID, t.transportation_mode, t.dis
    FROM (
        SELECT
            userID,
            transportation_mode,
            dis,
            ROW_NUMBER() OVER (PARTITION BY transportation_mode ORDER
BY dis DESC) AS RowNum
        FROM (
            SELECT
                userID,
                transportation_mode,
                max(dis) as dis
            FROM (
                SELECT
                    userID,
                    date,
                    transportation_mode,
                    sum(distance) as dis
                FROM (
```

```
                    SELECT
                        tp_max.*,
                        tp2.lon as lon_min,
                        tp2.lat as lat_min,
                        ST_Distance(point(lon_max,
lat_max),point(tp2.lon, tp2.lat)) as distance
                    FROM (
                        SELECT
                            ac_min_max.*,
                            tp1.lon as lon_max,
                            tp1.lat as lat_max
                        FROM (
                            SELECT
                                max(trackpointID) as max,
                                min(trackpointID) as min,
                                activityID
                            FROM (
                                SELECT *
                                FROM TrackPoint
                                ORDER BY trackpointID)tp
                            GROUP BY activityID)ac_min_max
                        LEFT JOIN TrackPoint tp1
                        ON ac_min_max.max=tp1.trackpointID)tp_max
                    LEFT JOIN TrackPoint tp2
                    ON tp_max.min=tp2.trackpointID)tab
                RIGHT JOIN (
                    SELECT
                        userID,
                        activityID,
                        transportation_mode,
                        date(start_date_time) as date
                    FROM Activity
                    WHERE date(start_date_time)=date(end_date_time) ) a
                ON a.activityID= tab.activityID
                GROUP BY userID,date,transportation_mode)tt
            GROUP BY transportation_mode,userID
            ORDER BY max(dis) DESC
            )tot
    )t
WHERE t.RowNum = 1;
```

Time execution: 5.5s

```
QUERY 10
  userID  transportation_mode             dis
--------  ---------------------  ------------
     128  airplane                10.2009
     128  bike                     0.313325
     128  boat                     0.586744
     128  bus                      1.45441
     128  car                      3.45451
     128  missing                248.996
     062  run                      0.00020134
     128  subway                   0.245041
     128  taxi                     0.234522
     062  train                    2.99243
     108  walk                     0.25725
```

Explanation *- The overall query is quite complex and involves multiple nested subqueries and joins to calculate and retrieve data related to transportation modes, distances, and user IDs. The result is a list of user IDs, transportation modes, and distances for the top row within each partition of transportation modes, ordered by maximum distance in descending order.*

The ST_Distance function in SQL is used to calculate the distance between two geometric objects or points in a two-dimensional (2D) or three-dimensional (3D) Cartesian space.

`ST_Distance(a , b)` ***a*** *and* ***b*** *are two geometric objects for which we can calculate the distance.*

The code then identifies all the activities that started and ended on the same day, in order to then identify the distance for each of them (calculated with the ST_Distance function). The distance is calculated on the furthest trackpoints (identified using the MAX() and MIN() functions). Once the distance has been calculated, the maximum distance is calculated in order to identify all the users who have made the longest journey, for each transportation_mode.

QUERY 11 Find all users who have invalid activities, and the number of invalid activities per user.

```
SET @prev_activity = NULL;
SET @prev_time = NULL;

SELECT
      userID,
      activityID,
      trackpointID,
      date_time,
      CASE
            WHEN @prev_activity is null or @prev_activity<>activityID THEN 0
            WHEN TIMESTAMPDIFF(MINUTE,date_time,@prev_time)>5 THEN 1
            ELSE 0 end AS flag_incorect,
            @prev_activity = activityID,
            @prev_time = date_time
FROM TrackPoint tp
JOIN Activity a ON tp.activityID = a.activityID
ORDER BY activityID,trackpointID;
```

Time execution: 14min

Explanation *- For these query we need to use variables. At the beginning, we initiate the variable at NULL. We are working on the table trackpoint join to activity to have the user. We create a column flag_incorrect which contains 0 if we change of activity and we are in the same activity, it will put 1 if the difference of time between the previous trackpoint and the current trackpoint is higher than 5 minutes. Then we group by activity and if the number of invalid trackpoint is different than 0, we flag the activity as incorrect. The last step is to group by user and sum the number of incorrect activity per user.*

QUERY 12 Find all users who have registered transportation_mode and their most used transportation_mode.

```
SELECT Tab3.userID, Tab3.transportation_mode FROM(
     SELECT Tab2.userID,Tab2.transportation_mode,
     ROW_NUMBER() OVER (PARTITION BY Tab2.userID ORDER BY
        Tab2.transportation_mode) AS row_num FROM (
        SELECT userID, transportation_mode FROM (
             SELECT userID, transportation_mode, COUNT(*) AS
count,
             RANK() OVER (PARTITION BY userID ORDER BY COUNT(*)
             DESC) AS usage_rank FROM Activity WHERE
             transportation_mode <> 'missing' GROUP BY userID,
             transportation_mode) AS Tab1 WHERE usage_rank = 1)
        AS Tab2)
AS Tab3 WHERE row_num = 1;
```

Explanation *- This query uses 3 different tables. The first one called Tab1 counts how many times each user has taken each transportation mode using the Activity table (we don't take into consideration activities with a "missing" transportation mode). In this same table we add a column usage_rank that ranks for each user the transportation modes by taking into account the usage count. For instance if one user has taken 3 times the bus and 1 time the car, the bus would be ranked 1 and the car 2. From this table, we keep rows where the rank is equal to 1 and we create a new table called Tab2 with userID and transportation_mode columns. We add a row_num column which corresponds to the number of the rows for each user. For instance, if a user has taken 3 times the bus and 3 times the car and both of these transportation modes have been ranked 1 in the previous table, the user is gonna then have two rows in the Tab2 table (but we only want one). By adding the row number for each user we can filter once again the data by creating one last table called Tab3 which keeps userID and tranporation_mode only where row_num is equal to 1 (which means we keep only the first row for each user).*

Discussion

CLEANING PART

For the cleaning part in task 1 we decided to keep an activity if the number of trackpoint is less than 2500 trackpoint but we did define a minimum number of trackpoint. It can be interesting to do so because an activity with less than 50 trackpoints or with a duration or less than 5 minutes will not give us much information.
It is a possibility for having a cleaner table but it is not really an issue for the computation of the table because we will remove a few lines with this operation.
In addition we decide to keep all the trackpoint because the data is very clean and we don't have any missing parameter :

```
+----------+----------+--------------+-----------+
| lat_null | lon_null | altitude_null | date_null |
+----------+----------+--------------+-----------+
|        0 |        0 |             0 |         0 |
+----------+----------+--------------+-----------+
```

Another possible cleaning operation could concern the "transportation_mode" variable present in the Activity table. Specifically, many of the users did not present any information regarding the transport method, but we felt it was necessary to keep this information as the cleaning would have eliminated important elements within the dataset. To distinguish users without the "transportation_mode" label we decided to insert the string "missing" instead of the NULL value (often controversial and difficult to manage).

FOREIGN KEY

To construct the tables, it was necessary to introduce Foreign Key constraints. Since an error relating to the foreign keys was observed when entering the data, we decided to insert the constraint later using the following code:

```
def altering_tables_with_foreign_key(self):

        query1 = """ALTER TABLE Activity
                ADD CONSTRAINT fk_activity
                FOREIGN KEY (userID)
                REFERENCES User(userID);"""
        self.cursor.execute(query1)
        self.db_connection.commit()

        query2 = """ALTER TABLE TrackPoint
                ADD CONSTRAINT fk_trackpoint
                FOREIGN KEY (activityID)
```

```
                REFERENCES Activity(activityID);"""
        self.cursor.execute(query2)
        self.db_connection.commit()
```

The error was linked to the userID attribute in the User and Activity table. Indeed we did not notice that in the User table, the ids were written as follows : 1, 2, 3, .., 99... In the Activity table the ids were written like this : 001, 002, 003, ..., 099... This mistake shows the importance of having a coherent notation in all the tables in order to be able to use foreign keys.

Globally, this project teaches us how rigorous we have to be with SQL in order to create a good quality database. Also, using Python to manage a database was new for all of us, so it was really interesting to learn how to link both technologies. Sometimes, it was even easier to use Python for certain computations than SQL. This shows how powerful both of these tools can be when they're used together.