Report

Assignment 2 - MySQL

Group: 35
Students: Albert Lesniewski, Thomas Tran, Md Anwarul Hasan

Introduction

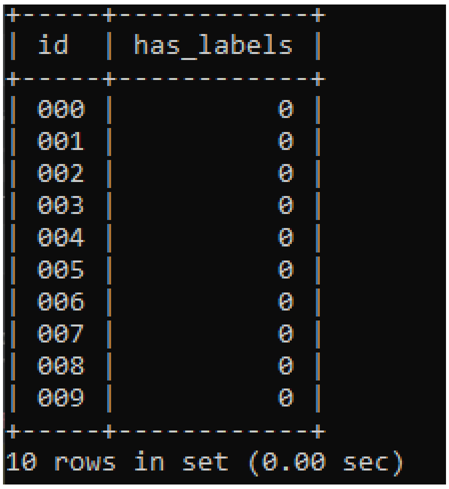
In this assignment we were tasked with designing a database schema for the GeoLife dataset, importing the data into the database, and doing any necessary data processing, and finally, running queries against the database in order to answer some of the questions that were specified as part of the assignment. We used the default proposed schema in the problem text. We focused on making our import code work in batches in order to achieve good import speed. As a group, we have worked mostly together on campus, in a pair-programming style, where we were programming and discussing the tasks at the same time. This way everyone could contribute to the programming tasks and be familiar with all parts of the code. For the questions that were supposed to be answered by querying the database, we split those between group members evenly and everyone did their part.

Results

Part 1

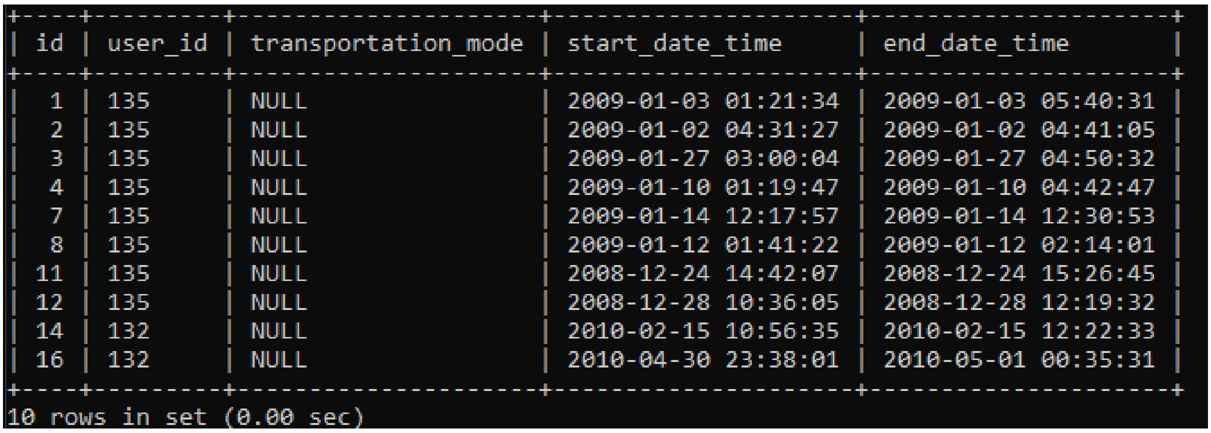
For part 1 we were supposed to import the GeoLife dataset into mysql database. We used the default proposed schema, as it seemed fairly reasonable. Meaning we first defined queries for creating the tables (user, activity and track_point). First we imported data for the user database, then the activities, and lastly the track points.

User table contains only two columns, the user id, and whether the user has specified the labels for their activities. The user id is a simple integer and the has_lables column is determined by the python script by checking if a given user id is specified in the labeled_id.txt file. Below you can see the top 10 entries in the user table after importing the data.



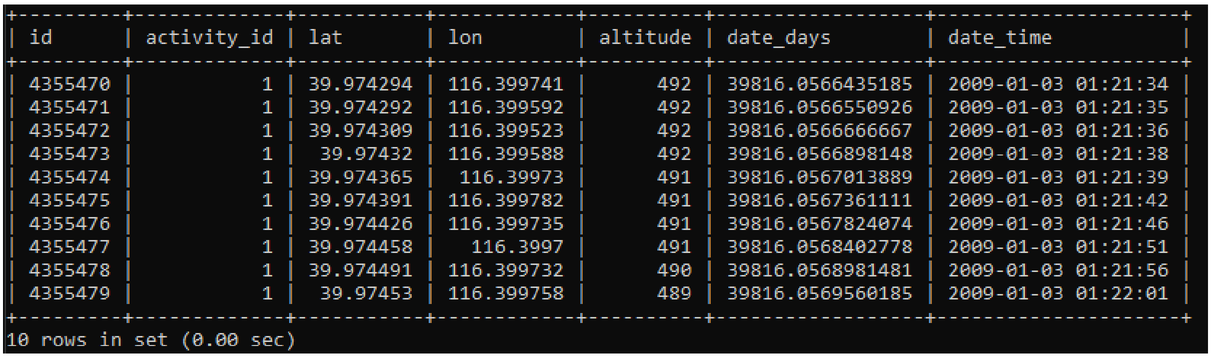
```
+-----+------------+
| id  | has_labels |
+-----+------------+
| 000 |          0 |
| 001 |          0 |
| 002 |          0 |
| 003 |          0 |
| 004 |          0 |
| 005 |          0 |
| 006 |          0 |
| 007 |          0 |
| 008 |          0 |
| 009 |          0 |
+-----+------------+
10 rows in set (0.00 sec)
```

Activity tablecontains 5 columns, the activity id, which is just a auto incremented integer, user id which works as a foreign key constraint and references the id of a user in the user table, transportation mode which specifies what kind of transportation was used for a activity, this column can be NULL, in which case no transportation mode was specified, start date time and end date time are column which specify the start and end date and time for an activity. Filling the activity table with data was probably the most challenging, as it required quite some data processing. Figuring out the transportation mode was especially difficult. In our implementation we built a hash map (python dictionary) for each user, with activity id as key and the transportation mode as value. This way, when building the insert query, we would check the hashmap, and check if the activity id for the current user exists in the hash map. If it exists, we add the specified mode to the query. If not, the value is set to NULL. When building the insert query we iterate over all users. For each user we get that user's activities, and build the insert values for that user. In the end we concatenate all the insert values for all the users into one insert query and we execute it. This way we do only one insert query for all activities instead of executing the insert query hundreds or thousands of times. Below is the top 10 entries in the activity table.



```
+----+---------+---------------------+---------------------+---------------------+
| id | user_id | transportation_mode | start_date_time     | end_date_time       |
+----+---------+---------------------+---------------------+---------------------+
|  1 | 135     | NULL                | 2009-01-03 01:21:34 | 2009-01-03 05:40:31 |
|  2 | 135     | NULL                | 2009-01-02 04:31:27 | 2009-01-02 04:41:05 |
|  3 | 135     | NULL                | 2009-01-27 03:00:04 | 2009-01-27 04:50:32 |
|  4 | 135     | NULL                | 2009-01-10 01:19:47 | 2009-01-10 04:42:47 |
|  7 | 135     | NULL                | 2009-01-14 12:17:57 | 2009-01-14 12:30:53 |
|  8 | 135     | NULL                | 2009-01-12 01:41:22 | 2009-01-12 02:14:01 |
| 11 | 135     | NULL                | 2008-12-24 14:42:07 | 2008-12-24 15:26:45 |
| 12 | 135     | NULL                | 2008-12-28 10:36:05 | 2008-12-28 12:19:32 |
| 14 | 132     | NULL                | 2010-02-15 10:56:35 | 2010-02-15 12:22:33 |
| 16 | 132     | NULL                | 2010-04-30 23:38:01 | 2010-05-01 00:35:31 |
+----+---------+---------------------+---------------------+---------------------+
10 rows in set (0.00 sec)
```

Track point table contains 7 columns. Track point id, which is auto incremented integer, activity id, which binds (references) a track point to a activity, latitude of the track point, longitude of the track point, altitude of the prack point, date days is number of days since the UNIX epoch and lastly date time of the track point. As specified in the task, we have filtered out activities that contain more than 2500 track points. We have tried to insert all the track points as one insert query, like we did with activities, but we got a problem with exceeding the max packet limit. Thus we split the insert queries into batches of 1000 insert values at a time. Below you can see the top 10 entries in the track point table.



```
+---------+-------------+-----------+------------+----------+------------------+---------------------+
| id      | activity_id | lat       | lon        | altitude | date_days        | date_time           |
+---------+-------------+-----------+------------+----------+------------------+---------------------+
| 4355470 |           1 | 39.974294 | 116.399741 |      492 | 39816.0566435185 | 2009-01-03 01:21:34 |
| 4355471 |           1 | 39.974292 | 116.399592 |      492 | 39816.0566550926 | 2009-01-03 01:21:35 |
| 4355472 |           1 | 39.974309 | 116.399523 |      492 | 39816.0566666667 | 2009-01-03 01:21:36 |
| 4355473 |           1 |  39.97432 | 116.399588 |      492 | 39816.0566898148 | 2009-01-03 01:21:38 |
| 4355474 |           1 | 39.974365 |  116.39973 |      491 | 39816.0567013889 | 2009-01-03 01:21:39 |
| 4355475 |           1 | 39.974391 | 116.399782 |      491 | 39816.0567361111 | 2009-01-03 01:21:42 |
| 4355476 |           1 | 39.974426 | 116.399735 |      491 | 39816.0567824074 | 2009-01-03 01:21:46 |
| 4355477 |           1 | 39.974458 |   116.3997 |      491 | 39816.0568402778 | 2009-01-03 01:21:51 |
| 4355478 |           1 | 39.974491 | 116.399732 |      490 | 39816.0568981481 | 2009-01-03 01:21:56 |
| 4355479 |           1 |  39.97453 | 116.399758 |      489 | 39816.0569560185 | 2009-01-03 01:22:01 |
+---------+-------------+-----------+------------+----------+------------------+---------------------+
10 rows in set (0.00 sec)
```

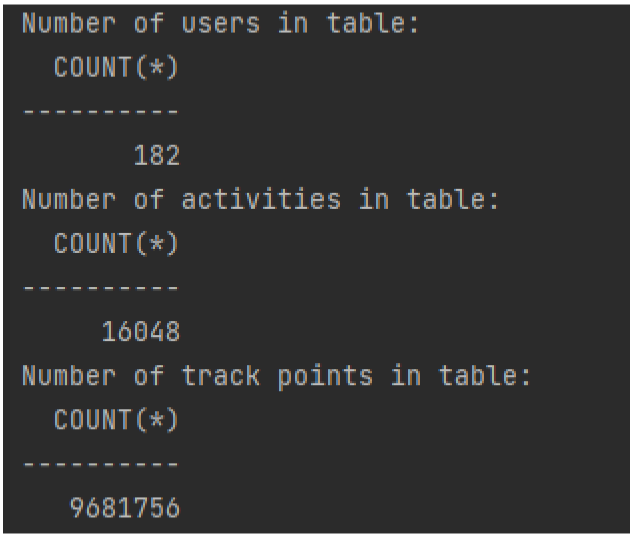
On a MacBook with a m1 chip, we managed to insert all the data in around 5 minutes.

Part 2

Q1: In order to answer how many users, activities, and track points there are in the table we executed those queries:

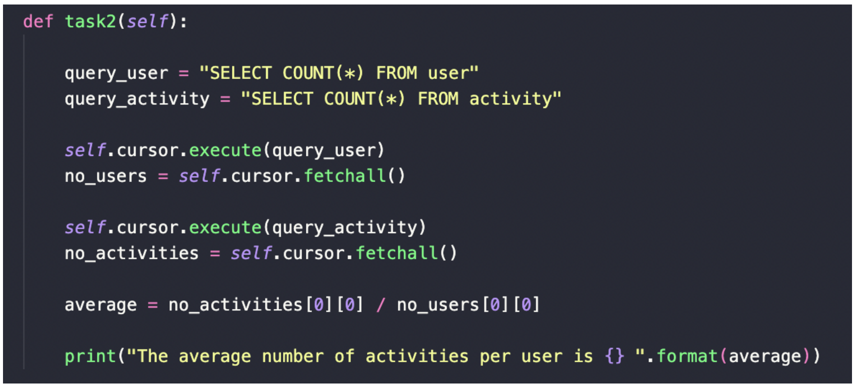
```
SELECT COUNT(*) FROM user;
SELECT COUNT(*) FROM activity;
SELECT COUNT(*) FROM track_point;
```

The results where:

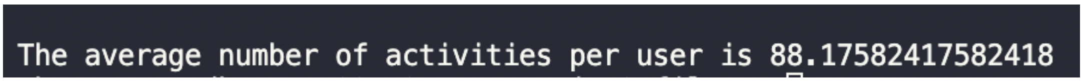


```
Number of users in table:
  COUNT(*)
----------
       182
Number of activities in table:
  COUNT(*)
----------
     16048
Number of track points in table:
  COUNT(*)
----------
   9681756
```

Q2: In order to find the average number of activities per user, the number of activities and users where found. The average was then calculated by dividing the number of activities by the number of users.

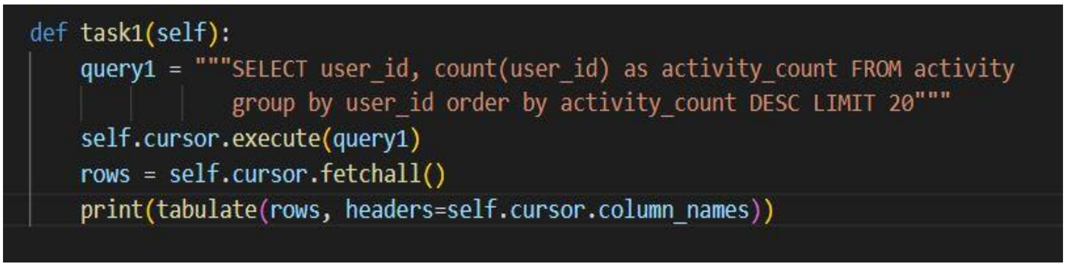


```python
def task2(self):

    query_user = "SELECT COUNT(*) FROM user"
    query_activity = "SELECT COUNT(*) FROM activity"

    self.cursor.execute(query_user)
    no_users = self.cursor.fetchall()

    self.cursor.execute(query_activity)
    no_activities = self.cursor.fetchall()

    average = no_activities[0][0] / no_users[0][0]

    print("The average number of activities per user is {} ".format(average))
```

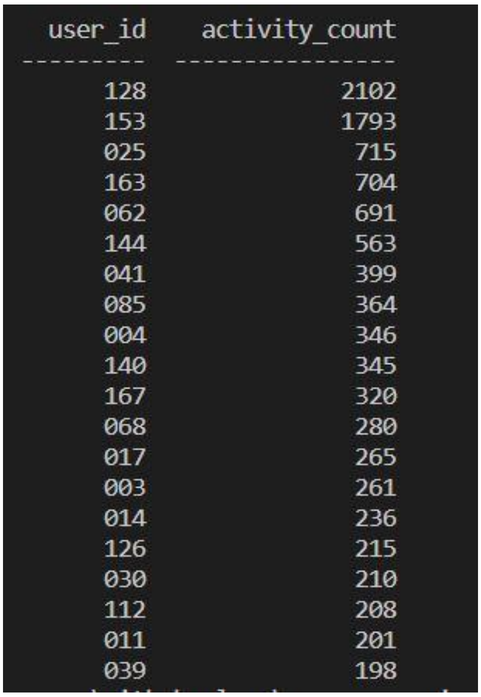


```
The average number of activities per user is 88.17582417582418
```

Q3: To find the top 20 users with the highest number of activities, we run queries in the activity table. We counted the activity of each user and selected the top 20.



```
def task1(self):
    query1 = """SELECT user_id, count(user_id) as activity_count FROM activity
                group by user_id order by activity_count DESC LIMIT 20"""
    self.cursor.execute(query1)
    rows = self.cursor.fetchall()
    print(tabulate(rows, headers=self.cursor.column_names))
```

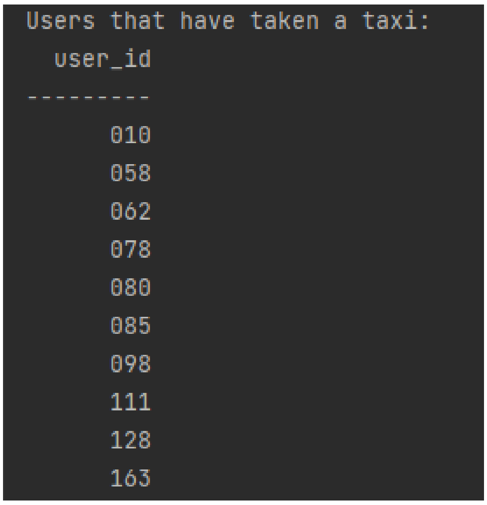


```
  user_id    activity_count
---------  ----------------
      128              2102
      153              1793
      025               715
      163               704
      062               691
      144               563
      041               399
      085               364
      004               346
      140               345
      167               320
      068               280
      017               265
      003               261
      014               236
      126               215
      030               210
      112               208
      011               201
      039               198
```

Q4: In order to find out all users that have taken a taxi we have executed the following query:

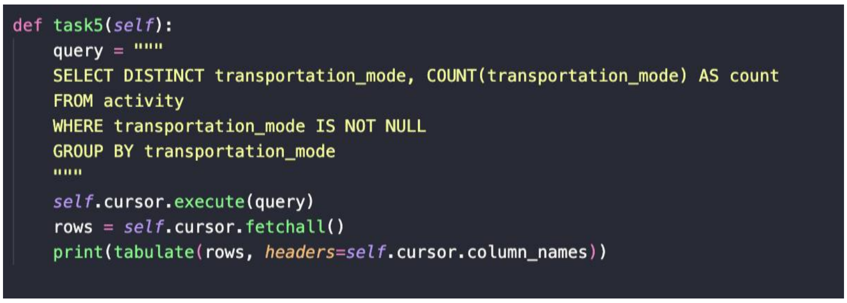
> **SELECT user_id FROM activity WHERE transportation_mode='taxi' GROUP BY user_id;**

The results where:

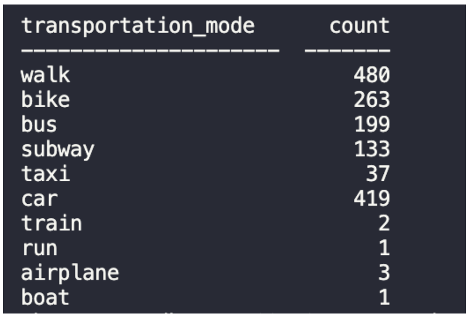


```
Users that have taken a taxi:
  user_id
---------
      010
      058
      062
      078
      080
      085
      098
      111
      128
      163
```

Q5: The number of activities per transportation mode was found by using the keyword DISTINCT and the count of a given transportation mode. In our activity table, those activities that are not labeled have a value of NULL.

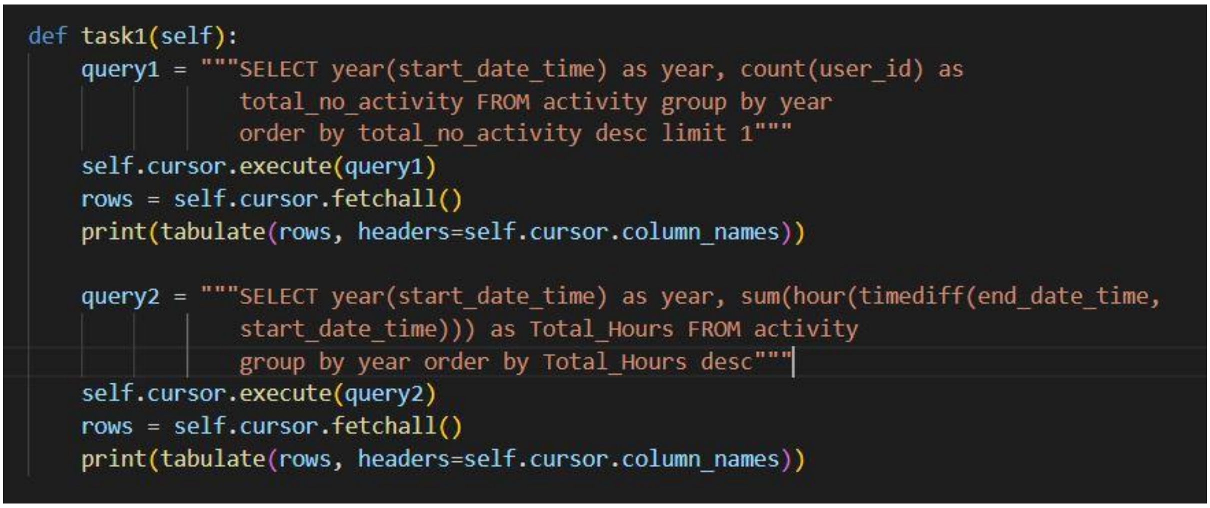


```
def task5(self):
    query = """
    SELECT DISTINCT transportation_mode, COUNT(transportation_mode) AS count
    FROM activity
    WHERE transportation_mode IS NOT NULL
    GROUP BY transportation_mode
    """
    self.cursor.execute(query)
    rows = self.cursor.fetchall()
    print(tabulate(rows, headers=self.cursor.column_names))
```



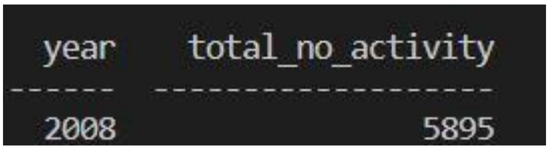
```
transportation_mode      count
---------------------  -------
walk                       480
bike                       263
bus                        199
subway                     133
taxi                        37
car                        419
train                        2
run                          1
airplane                     3
boat                         1
```

Q6:



```
def task1(self):
    query1 = """SELECT year(start_date_time) as year, count(user_id) as
                total_no_activity FROM activity group by year
                order by total_no_activity desc limit 1"""
    self.cursor.execute(query1)
    rows = self.cursor.fetchall()
    print(tabulate(rows, headers=self.cursor.column_names))

    query2 = """SELECT year(start_date_time) as year, sum(hour(timediff(end_date_time,
                start_date_time))) as Total_Hours FROM activity
                group by year order by Total_Hours desc"""
    self.cursor.execute(query2)
    rows = self.cursor.fetchall()
    print(tabulate(rows, headers=self.cursor.column_names))
```

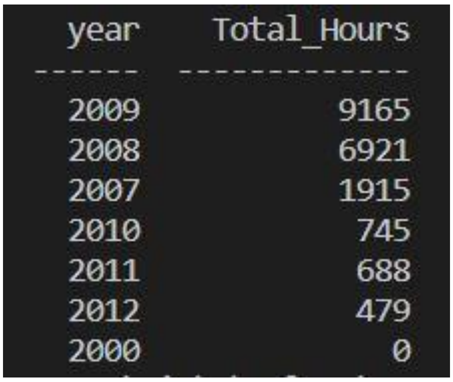
a) To find the year with the most activity we searched the activity table. We selected each year & counted the number of activities each year. Thus we found the year with the most activity.



```
  year    total_no_activity
------  -------------------
  2008                 5895
```

b) To get the answer whether this year has the most recorded hours we calculated total hours of each year and found that this is not the year that has the most recorded hours rather year 2009 has the most recorded hours.

c)



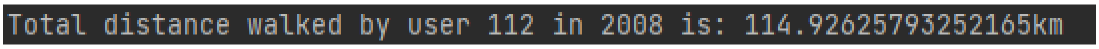
```
  year    Total_Hours
------  -------------
  2009           9165
  2008           6921
  2007           1915
  2010            745
  2011            688
  2012            479
  2000              0
```

Q7: In order to find the total distance walked by user 112 in 2008, we first had to execute a query that would give us all the track points for user 112 that were registered in 2008 and for which the activity was registered with walk as the transportation mode, the query was:

SELECT activity.id, lat, lon

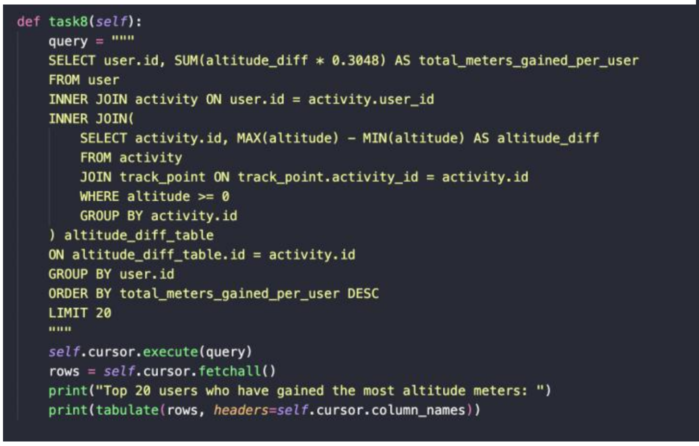
```
 FROM activity INNER JOIN track_point ON
 activity.id=track_point.activity_id
 WHERE user_id=112 AND YEAR(start_date_time)=2008 AND
 transportation_mode='walk'
ORDER BY activity.id
```

Then, we wrote a simple python script that processed the resulting track point data from the query, and used the haversine distance to compute the distance between track points that are registered in latitude and longitude coordinates.
The result was:

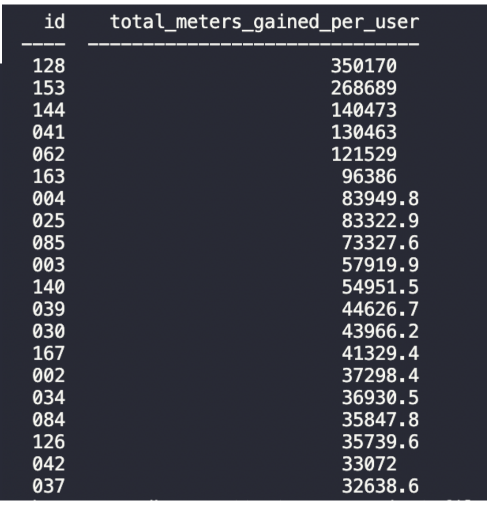


```
Total distance walked by user 112 in 2008 is: 114.92625793252165km
```

Q8: To find the top 20 users who have gained the most altitude, the user table, activity table and a sub table of activity and track_point containing the altitude difference for each activity were inner joined so that the program could take the sum of all the altitudes in every activity connected to each user. Because the altitude is given in feet and the solution requires meters, the altitude was multiplied by 0.3048.

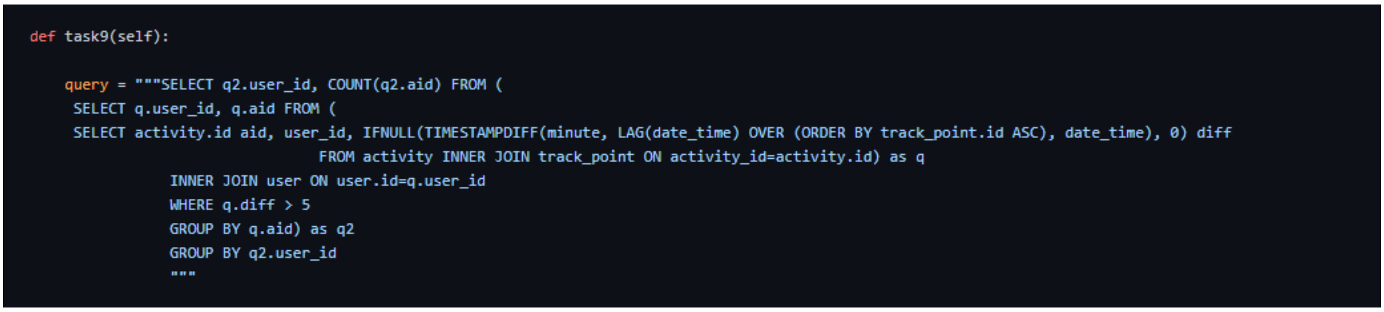


```
def task8(self):
    query = """
    SELECT user.id, SUM(altitude_diff * 0.3048) AS total_meters_gained_per_user
    FROM user
    INNER JOIN activity ON user.id = activity.user_id
    INNER JOIN(
        SELECT activity.id, MAX(altitude) - MIN(altitude) AS altitude_diff
        FROM activity
        JOIN track_point ON track_point.activity_id = activity.id
        WHERE altitude >= 0
        GROUP BY activity.id
    ) altitude_diff_table
    ON altitude_diff_table.id = activity.id
    GROUP BY user.id
    ORDER BY total_meters_gained_per_user DESC
    LIMIT 20
    """
    self.cursor.execute(query)
    rows = self.cursor.fetchall()
    print("Top 20 users who have gained the most altitude meters: ")
    print(tabulate(rows, headers=self.cursor.column_names))
```

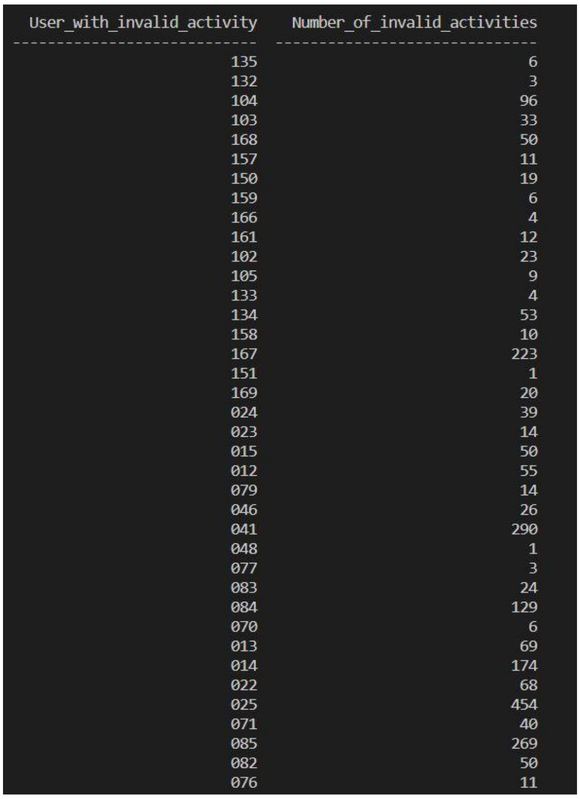


```
  id    total_meters_gained_per_user
----  ------------------------------
 128                      350170
 153                      268689
 144                      140473
 041                      130463
 062                      121529
 163                       96386
 004                       83949.8
 025                       83322.9
 085                       73327.6
 003                       57919.9
 140                       54951.5
 039                       44626.7
 030                       43966.2
 167                       41329.4
 002                       37298.4
 034                       36930.5
 084                       35847.8
 126                       35739.6
 042                       33072
 037                       32638.6
```

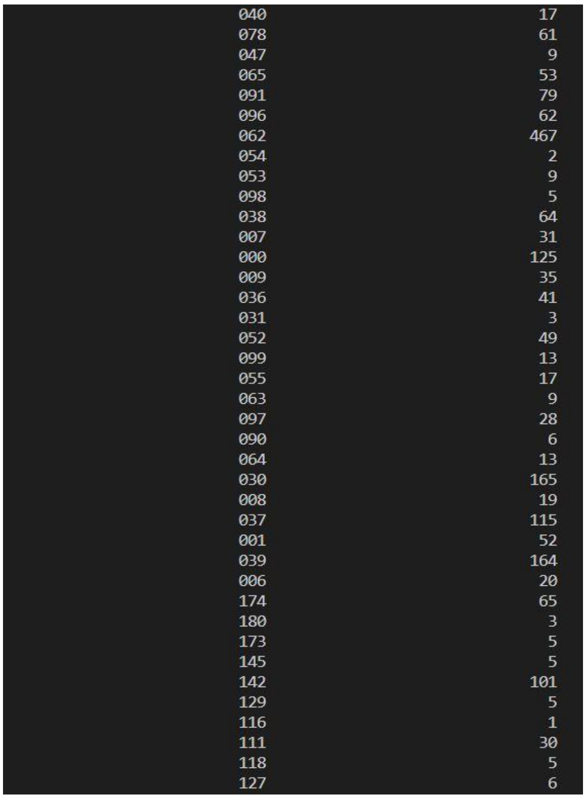
Q9: To find the users with invalid activity and also total number of invalid activity, we first searched for track points which have a gap of 5 minutes between them. We used LAG and TIMESTAMPDIFF in this regard. Then we combined these results with the activity table to find the users with invalid activity by comparing the id field of the activity table with activity_id field of trackpoint table. After that we summed the invalid activity and grouped those by user_id. The query and the outputs are provided below.



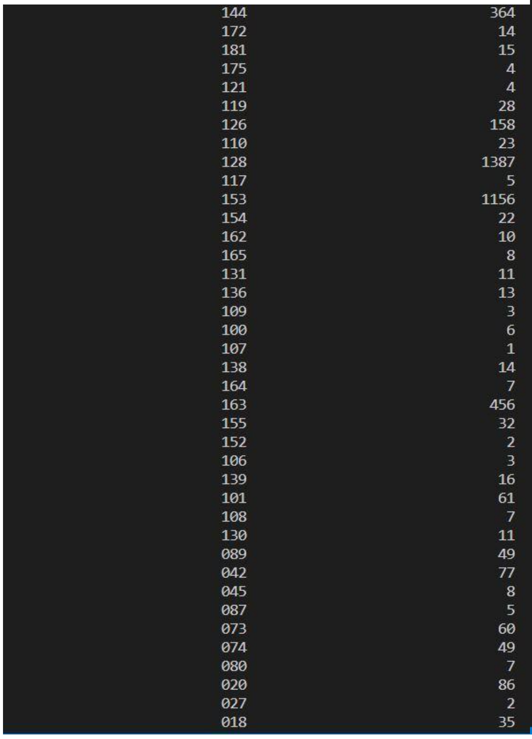
```
def task9(self):

    query = """SELECT q2.user_id, COUNT(q2.aid) FROM (
     SELECT q.user_id, q.aid FROM (
     SELECT activity.id aid, user_id, IFNULL(TIMESTAMPDIFF(minute, LAG(date_time) OVER (ORDER BY track_point.id ASC), date_time), 0) diff
                            FROM activity INNER JOIN track_point ON activity_id=activity.id) as q
                INNER JOIN user ON user.id=q.user_id
                WHERE q.diff > 5
                GROUP BY q.aid) as q2
                GROUP BY q2.user_id
                """
```



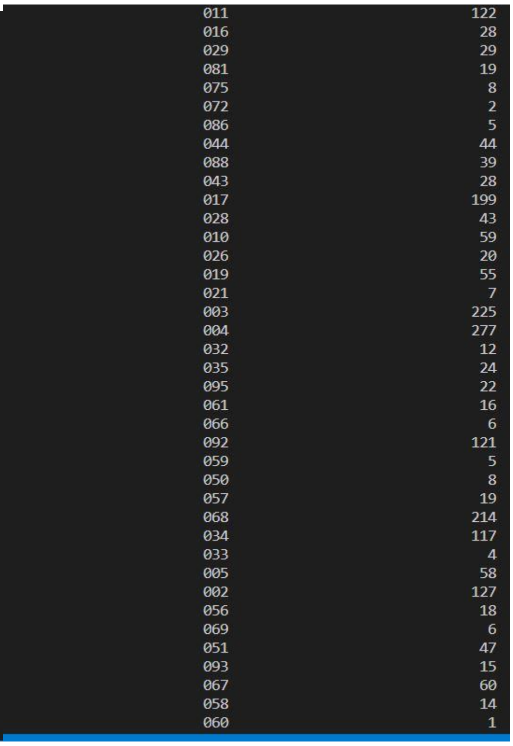
```
 User_with_invalid_activity   Number_of_invalid_activities
----------------------------  ------------------------------
                         135                               6
                         132                               3
                         104                              96
                         103                              33
                         168                              50
                         157                              11
                         150                              19
                         159                               6
                         166                               4
                         161                              12
                         102                              23
                         105                               9
                         133                               4
                         134                              53
                         158                              10
                         167                             223
                         151                               1
                         169                              20
                         024                              39
                         023                              14
                         015                              50
                         012                              55
                         079                              14
                         046                              26
                         041                             290
                         048                               1
                         077                               3
                         083                              24
                         084                             129
                         070                               6
                         013                              69
                         014                             174
                         022                              68
                         025                             454
                         071                              40
                         085                             269
                         082                              50
                         076                              11
```



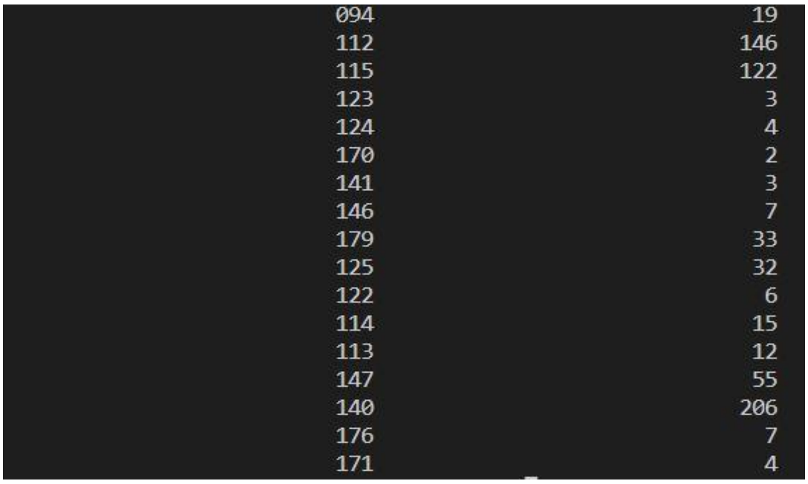
```
                         040                              17
                         078                              61
                         047                               9
                         065                              53
                         091                              79
                         096                              62
                         062                             467
                         054                               2
                         053                               9
                         098                               5
                         038                              64
                         007                              31
                         000                             125
                         009                              35
                         036                              41
                         031                               3
                         052                              49
                         099                              13
                         055                              17
                         063                               9
                         097                              28
                         090                               6
                         064                              13
                         030                             165
                         008                              19
                         037                             115
                         001                              52
                         039                             164
                         006                              20
                         174                              65
                         180                               3
                         173                               5
                         145                               5
                         142                             101
                         129                               5
                         116                               1
                         111                              30
                         118                               5
                         127                               6
```



```
144         364
172          14
181          15
175           4
121           4
119          28
126         158
110          23
128        1387
117           5
153        1156
154          22
162          10
165           8
131          11
136          13
109           3
100           6
107           1
138          14
164           7
163         456
155          32
152           2
106           3
139          16
101          61
108           7
130          11
089          49
042          77
045           8
087           5
073          60
074          49
080           7
020          86
027           2
018          35
```



```
011         122
016          28
029          29
081          19
075           8
072           2
086           5
044          44
088          39
043          28
017         199
028          43
010          59
026          20
019          55
021           7
003         225
004         277
032          12
035          24
095          22
061          16
066           6
092         121
059           5
050           8
057          19
068         214
034         117
033           4
005          58
002         127
056          18
069           6
051          47
093          15
067          60
058          14
060           1
```

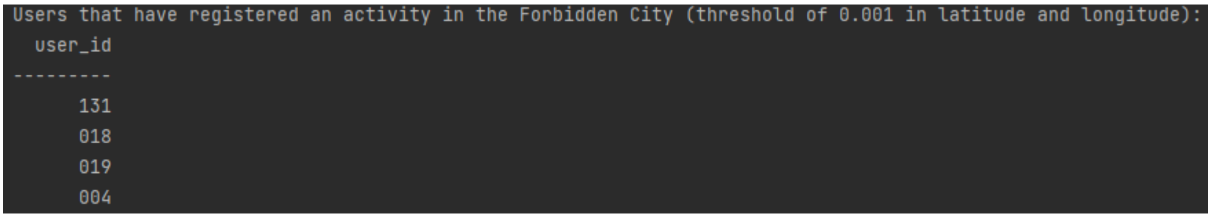


```
094          19
112         146
115         122
123           3
124           4
170           2
141           3
146           7
179          33
125          32
122           6
114          15
113          12
147          55
140         206
176           7
171           4
```

Q10: In order to find out what users have registered an activity in the Forbidden City in Beijing we had to execute the following query:

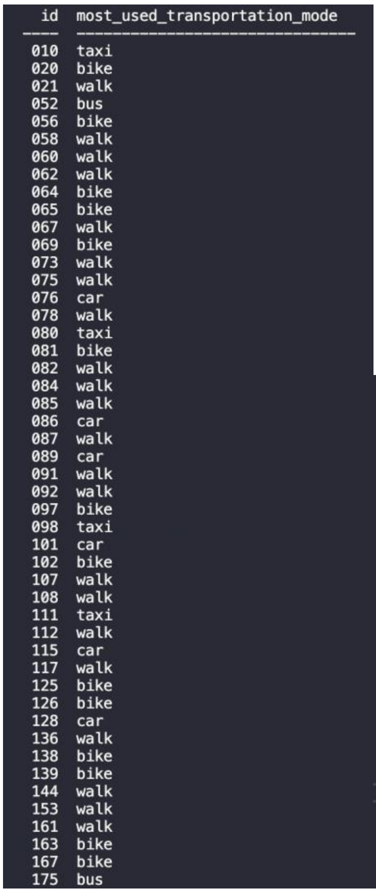
```
SELECT user_id
    FROM activity INNER JOIN track_point ON
    activity.id=track_point.activity_id
    WHERE lat BETWEEN 39.915 AND 39.917 AND lon BETWEEN 116.396
    AND 116.398
   GROUP BY user_id
```

The result was:

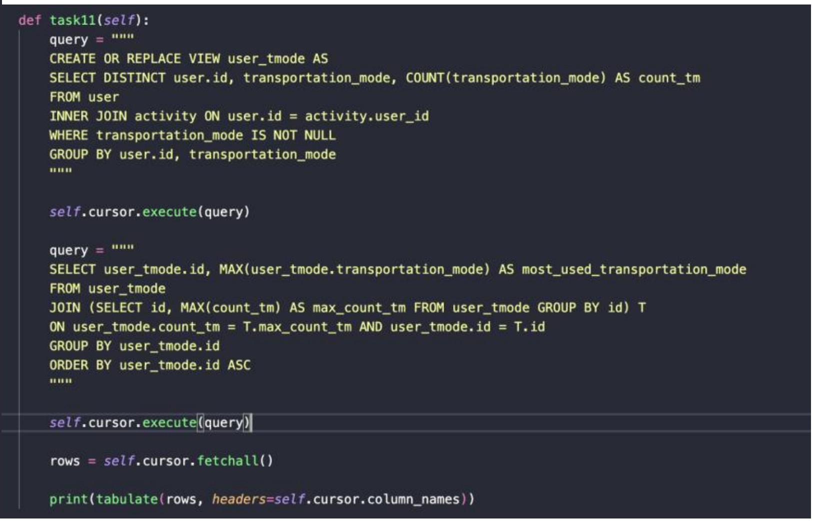


```
Users that have registered an activity in the Forbidden City (threshold of 0.001 in latitude and longitude):
  user_id
---------
      131
      018
      019
      004
```

Q11: To find the most used transportation mode for each user (which has their activities labeled), the program first constructs a view of a table with the user and count for each transportation mode the user has. This view is then later used in another query. In this query, the user id and the max transportation mode is chosen so that only one transportation mode is chosen if a user has the same number of activities tagged with the labels. user_tmode (which is the view) is then joined with another subquery (called T) which contains the number of counts of the transportation mode that occurs the most for a user. By doing this, the most used transportation mode for each user is found.



```
  id  most_used_transportation_mode
----  -------------------------------
 010  taxi
 020  bike
 021  walk
 052  bus
 056  bike
 058  walk
 060  walk
 062  walk
 064  bike
 065  bike
 067  walk
 069  bike
 073  walk
 075  walk
 076  car
 078  walk
 080  taxi
 081  bike
 082  walk
 084  walk
 085  walk
 086  car
 087  walk
 089  car
 091  walk
 092  walk
 097  bike
 098  taxi
 101  car
 102  bike
 107  walk
 108  walk
 111  taxi
 112  walk
 115  car
 117  walk
 125  bike
 126  bike
 128  car
 136  walk
 138  bike
 139  bike
 144  walk
 153  walk
 161  walk
 163  bike
 167  bike
 175  bus
```



```
def task11(self):
    query = """
    CREATE OR REPLACE VIEW user_tmode AS
    SELECT DISTINCT user.id, transportation_mode, COUNT(transportation_mode) AS count_tm
    FROM user
    INNER JOIN activity ON user.id = activity.user_id
    WHERE transportation_mode IS NOT NULL
    GROUP BY user.id, transportation_mode
    """

    self.cursor.execute(query)

    query = """
    SELECT user_tmode.id, MAX(user_tmode.transportation_mode) AS most_used_transportation_mode
    FROM user_tmode
    JOIN (SELECT id, MAX(count_tm) AS max_count_tm FROM user_tmode GROUP BY id) T
    ON user_tmode.count_tm = T.max_count_tm AND user_tmode.id = T.id
    GROUP BY user_tmode.id
    ORDER BY user_tmode.id ASC
    """

    self.cursor.execute(query)

    rows = self.cursor.fetchall()

    print(tabulate(rows, headers=self.cursor.column_names))
```

Discussion

In part 1 of this assignment, what surprised us the most was how few of the labels actually matched with an activity. But the reason as to why this happened might be due to the method we used to match them. We looked at the start time and end time for a label and tried to match it up with the start time and end time of an activity by looking at its first and last track point. However, there might have been a few seconds or even minutes differences which we didn't take into account. Removing activities with more than 2500 track points might have affected this result as well, because some of these activities might've had labels.

In part 2, something that we didn't expect was the amount of invalid activities from query 9. By comparing the number of invalid activities of an user with an user's total number of activities, we saw that in general, almost half of an user's activities were invalid.

From this assignment, by writing the data in batches to the database, we saw the practical results of sequential writes vs random writes. By writing in batches, the data is written in clumps of blocks, which makes it faster than if it were to write each data by its own across a storage medium. This was clearly shown, as we got a huge performance boost when writing the data in batches.

Summary

In this assignment, we managed to clean up the data from the dataset and insert it into a database. By using a combination of python and SQL queries, we also retrieved the desirable results as described in the query tasks.