Report

Assignment 2 - MySQL

Group: 38

Delivery: 06/Oct/2023, 23:59

Students: Jesper Barfod, Fabian Kongelf, Bawan Nuri

Introduction

This assignment involved a series of database tasks which were solved using MySQL and Python programming. An Ubuntu Virtual Machine equipped with a pre-installed MySQL environment was utilized for database setup. The tasks involved cleaning data. creating tables, inserting data, and formulating queries.

The dataset for this assignment was an extensive collection of users' outdoor movements and routines, recorded as coordinates, timestamps and transportation modes. The data is primarily sourced from Beijing, as well as some users from Europe and the US.

We used github for code collaboration. The repository can be found at https://github.com/FabianKongelf/tdt4225-assignment2

We used the provided python-file `DBConnector.py` to connect to the VM and `example.py` as a baseline for structuring the code. Our code is divided into two parts, one part for task1 and one part for task2, the code is located in the respective files `task1.py` and `task2.py`.

Part 1 - Data cleaning and insertion

The outline provided in the assignment description was used as a baseline to create our database. Our database consists of the three tables: **activity**, **trackpoint**, **user**. All tables have the same fields as stated in the proposed database design from the assignment description. Each table except “user” has an ID field with auto_increment for automatic generation of unique IDs, the IDs for users are provided by the dataset and unique, thus do not need to be automatically generated.

```
mysql> show tables;
+---------------------+
| Tables_in_db_Task2 |
+---------------------+
| activity            |
| trackpoint          |
| user                |
+---------------------+
```

all tables

```
mysql> desc activity;
+---------------------+--------------+------+-----+---------+----------------+
| Field               | Type         | Null | Key | Default | Extra          |
+---------------------+--------------+------+-----+---------+----------------+
| id                  | int          | NO   | PRI | NULL    | auto_increment |
| user_id             | varchar(3)   | NO   | MUL | NULL    |                |
| transportation_mode | varchar(100) | YES  |     | NULL    |                |
| start_date_time     | datetime     | NO   |     | NULL    |                |
| end_date_time       | datetime     | NO   |     | NULL    |                |
+---------------------+--------------+------+-----+---------+----------------+
```

activity table

```
mysql> desc trackpoint;
+-------------+----------+------+-----+---------+----------------+
| Field       | Type     | Null | Key | Default | Extra          |
+-------------+----------+------+-----+---------+----------------+
| id          | int      | NO   | PRI | NULL    | auto_increment |
| activity_id | int      | NO   | MUL | NULL    |                |
| lat         | double   | NO   |     | NULL    |                |
| lon         | double   | NO   |     | NULL    |                |
| altitude    | int      | YES  |     | NULL    |                |
| date_days   | double   | NO   |     | NULL    |                |
| date_time   | datetime | NO   |     | NULL    |                |
+-------------+----------+------+-----+---------+----------------+
```

trackpoint table

```
mysql> desc user;
+-----------+------------+------+-----+---------+-------+
| Field     | Type       | Null | Key | Default | Extra |
+-----------+------------+------+-----+---------+-------+
| id        | varchar(3) | NO   | PRI | NULL    |       |
| has_label | tinyint(1) | NO   |     | 0       |       |
+-----------+------------+------+-----+---------+-------+
```

user table

Top 10 rows of Activity:

id	user_id	transportation_mode	start_date_time	end_date_time
18002	000	NULL	2008-10-23 02:53:04	2008-10-23 11:11:12
18003	000	NULL	2008-10-24 02:09:59	2008-10-24 02:47:06
18004	000	NULL	2008-10-26 13:44:07	2008-10-26 15:04:07
18005	000	NULL	2008-10-27 11:54:49	2008-10-27 12:05:54
18006	000	NULL	2008-10-28 00:38:26	2008-10-28 05:03:42
18007	000	NULL	2008-10-29 09:21:38	2008-10-29 09:30:28
18008	000	NULL	2008-10-29 09:30:38	2008-10-29 09:46:43
18009	000	NULL	2008-11-03 10:13:36	2008-11-03 10:16:01
18010	000	NULL	2008-11-03 23:21:53	2008-11-04 03:31:08
18011	000	NULL	2008-11-10 01:36:37	2008-11-10 03:46:12

"id" is a primary key and is automatically generated through auto_increment as data is inserted. "user_id" is a foreign key referring to the user table's primary key "id". By default, transportation_mode is NULL and does not need to be stated when inserting data.

Top 10 rows of Trackpoint:

id	activity_id	lat	lon	altitude	date_days	date_time
10870492	18002	39.984702	116.318417	492	39744.1201851852	2008-10-23 02:53:04
10870493	18002	39.984683	116.31845	492	39744.1202546296	2008-10-23 02:53:10
10870494	18002	39.984686	116.318417	492	39744.1203125	2008-10-23 02:53:15
10870495	18002	39.984688	116.318385	492	39744.1203703704	2008-10-23 02:53:20
10870496	18002	39.984655	116.318263	492	39744.1204282407	2008-10-23 02:53:25
10870497	18002	39.984611	116.318026	493	39744.1204861111	2008-10-23 02:53:30
10870498	18002	39.984608	116.317761	493	39744.1205439815	2008-10-23 02:53:35
10870499	18002	39.984563	116.317517	496	39744.1206018519	2008-10-23 02:53:40
10870500	18002	39.984539	116.317294	500	39744.1206597222	2008-10-23 02:53:45
10870501	18002	39.984606	116.317065	505	39744.1207175926	2008-10-23 02:53:50

Same as activity, "id" is automatically generated by auto_increament and is a primary key, "activity_id" is a foreign key refering to the activity table's primary key "id". If altitude's value is -777 (a value stated as not valid) the altitude is changed to the previous row's altitude, this is due to low changes of the altitude between two rows next to each other and to prevent excessive rise and fall in altitude making the attribute more usable later.

Top 10 rows of User:

id	has_label
000	0
001	0
002	0
003	0
004	0
005	0
006	0
007	0
008	0
009	0

id is the primary key.

When we inserted data into the tables we started with the user-table. We assumed a folder within `./dataset/dataset/Data` represents a single user, with the name of the folder representing a user's id. All users are generated by default with a false state on has_label, however this value becomes true if a user id is identified within the `labeled_ids.txt` document.

As the dataset contained trackpoint files within each user folder, we inserted activities and trackpoints simultaneously. First, we read a .plt file. We assumed that lines 7 and beyond were rows to be inserted into the trackpoint table (the first 6 lines were header lines). We checked if the file was more than 2506 lines long; if it was, we ignored it. Otherwise, we began creating an activity. We assumed that each .plt file represented one activity; thus, the first row (line 7) contained the activity's first trackpoint and thus start date, the file's last row and last trackpoint is the activity's end date. The activity's user was determined by the user whose folder the file was located within. If the user had labels, we searched that user's labels.txt file for matching start and end dates; If a match was found, the activity was assigned a transportation mode. With these assumptions, we created an activity. After creating an activity, we captured the activity id, and each row from the .plt file was then inserted into the trackpoint table with the activity id as a foreign key.

Part 2 - Queries

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

We used these queries:

- `SELECT COUNT(*) FROM user;`
- `SELECT COUNT(*) FROM activity;`
- `SELECT COUNT(*) FROM trackpoint;`

which gave the following result with python:

```
total amount of rows in database
ant users: 182
ant activities: 16048
ant trackpoint: 9681756
```

These queries ran directly in mysql:

```
mysql> select count(*) from user;
+----------+
| count(*) |
+----------+
|      182 |
+----------+
```

```
mysql> select count(*) from activity;
+----------+
| count(*) |
+----------+
|    16048 |
+----------+
```

```
mysql> select count(*) from trackpoint;
+----------+
| count(*) |
+----------+
|  9681756 |
+----------+
```

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

We used this query:

```
SELECT AVG(total_trackpoints) as avg_trackpoints, MIN(total_trackpoints) as
min_trackpoints, MAX(total_trackpoints) as max_trackpoints
FROM (SELECT activity.user_id, COUNT(trackpoint.id) AS total_trackpoints
FROM trackpoint JOIN activity ON trackpoint.activity_id = activity.id
GROUP BY activity.user_id) AS subquery;
```

which gave the following result with python:

```
avg trackpoints:   55963.9075
min trackpoints:   17
max trackpoints:   1010325
```

2.3. Find the top 15 users with the highest number of activities.

We used this query:

```
SELECT activity.user_id, COUNT(activity.id) AS total_activities
FROM activity
GROUP BY activity.user_id
ORDER BY total_activities DESC
LIMIT 15;
```

which gave the following result in python:

```
Top 15 users highest activites

  User    Activities
------  ------------
   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
```

2.4. Find all users who have taken a bus.

We used this query:

```
SELECT user_id
FROM activity
WHERE transportation_mode = "Bus"
GROUP BY user.id
```

and got the following results with python:

```
total amount of rows in database
ant users: 182
ant activities: 16048
ant trackpoint: 9681756
```

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

We used this query:

```
SELECT user_id,COUNT(DISTINCT transportation_mode) as distinct_mode
FROM activity
GROUP BY user_id
ORDER BY distinct_mode DESC
LIMIT 10;
```

and got the following result with python:

Top 10 users by the amount of different transportation modes

User	Different transportation modes
128	9
062	7
085	4
084	3
058	3
163	3
078	3
081	3
112	3
065	2

"Different transportation modes" are the amount of different transportation modes the user has taken.

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

We used this query:

```
SELECT user_id, start_date_time, end_date_time, transportation_mode
FROM activity
GROUP BY user_id, start_date_time, end_date_time, transportation_mode
HAVING (COUNT(user_id) > 1) and (COUNT(start_date_time) > 1) and
(COUNT(end_date_time) > 1) and (COUNT(transportation_mode) > 1);
```

and got this response in mysql

```
mysql> SELECT user_id, start_date_time, end_date_time, transportation_mode, COUNT(*) FROM activity
GROUP BY user_id, start_date_time, end_date_time, transportation_mode HAVING COUNT(*)>1;
Empty set (0.05 sec)
```

The empty set is returned, meaning there are zero duplicates. The print in python is thus empty.

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

We used this query:

```
SELECT COUNT(DISTINCT user_id) as Overnight_users
FROM activity
WHERE DATEDIFF(start_date_time, end_date_time) <> 0 AND
DATEDIFF(start_date_time, end_date_time) <= 1;
```

and got this result in python:

```
Total users who have an activity which last until the next day: 98
```

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

We used this query:

```
SELECT IFNULL(transportation_mode, "-"), user_id,
TIMEDIFF(end_date_time, start_date_time) as duration
FROM activity
WHERE DATEDIFF(start_date_time, end_date_time) <> 0 AND
DATEDIFF(start_date_time, end_date_time) <= 1
LIMIT 20 OFFSET 620;
```

We applied a limit since the output is exceedingly large, along with an offset to pinpoint a more intriguing data point in the result table for display. In Python, we got:

```
All activites which last until the next day (limit 20, offset 630):
Transportation mode      User  Duration
---------------------  ------  ----------
-                         125  0:41:29
-                         125  5:15:38
-                         126  1:09:45
-                         126  4:56:18
walk                      126  0:07:20
-                         128  8:49:25
-                         128  6:10:56
-                         128  3:02:49
-                         128  13:03:51
-                         128  10:44:49
-                         128  2:24:12
-                         128  12:53:24
-                         128  1:40:05
-                         128  5:51:12
car                       128  1:04:19
car                       128  0:44:11
bus                       128  0:58:44
subway                    128  0:30:57
car                       128  0:33:08
car                       128  1:40:04
```

2.8. 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)

The result here is a combination of python code and sql query. The answer was 39 pairs as shown in the top left corner in the screenshot to the right down below, with a time limit for comparisons of 1.0 second. This is not an accurate answer. The runtime for solving this question accurately was too long, so we added a limit, where it would do row comparisons between each pair of users for just 0.01 seconds. This reduced the runtime substantially, and we were able to find a result within 2.5 minutes as shown in the screenshot to the left. By increasing allowed time on run comparison, you will get a more accurate answer. Without this limit, you will find the fully correct answer, but the run time will be substantially longer.

Queries used in python code to make dictionary to connect user_ids to belonging trackpoints:

Query 1 used to get all ids:

```
SELECT id from user;
```

Query 2 used to connect all user ids to their belonging trackpoints:

```
SELECT lat, lon, date_time
FROM trackpoint
WHERE activity_id IN (SELECT id
FROM activity
WHERE user_id = {user_id});
```

Result after executing whole python code:

Total elapsed time for pair comparisons: 2 minutes 34.03 seconds
Time limit for comparing each pair: 0.01
Amount of pairs: 23
The pairs:

Pair	Timediff	Distance	Rows compared until match
('003', '004')	2	36.8	1
('011', '088')	0	0	1
('012', '127')	0	0	1
('013', '070')	0	0	1
('051', '104')	18	25.6	9
('057', '061')	1	35.7	5617
('057', '094')	0	0	1
('057', '150')	0	0	1
('057', '157')	9	26.6	5004
('081', '125')	0	0	1
('081', '136')	0	0	1
('092', '109')	8	27.5	5956
('094', '150')	0	0	1
('094', '157')	9	26.6	5004
('101', '109')	0	0	1
('101', '173')	5	40.9	1943
('104', '109')	5	17.6	5361
('108', '139')	0	0	870
('109', '173')	5	40.9	1943
('125', '136')	0	0	1
('126', '167')	0	0	5248
('150', '157')	9	26.6	5004
('164', '176')	8	10.3	1412

Total elapsed time for pair comparisons: 24 minutes 44.22 seconds
Time limit for comparing each pair: 0.1
Amount of pairs: 34
The pairs:

Pair	Timediff	Distance	Rows compared until match
('003', '004')	2	36.8	1
('011', '088')	0	0	1
('012', '127')	0	0	1
('013', '070')	0	0	1
('047', '055')	28	42.5	12688
('051', '104')	18	25.6	9
('051', '110')	14	46.7	16566
('057', '061')	1	35.7	5617
('057', '094')	0	0	1
('057', '150')	0	0	1
('057', '157')	9	26.6	5004
('061', '094')	24	33.5	36326
('061', '150')	24	33.5	36326
('061', '157')	19	7.3	20001
('069', '129')	5	1.4	7262
('081', '125')	0	0	1
('081', '136')	0	0	1
('084', '085')	0	0	27612
('092', '109')	8	27.5	5956
('094', '150')	0	0	1
('094', '157')	9	26.6	5004
('101', '109')	0	0	1
('101', '173')	5	40.9	1943
('104', '109')	5	17.6	5361
('104', '173')	11	42.4	7765
('108', '139')	0	0	870
('109', '173')	5	40.9	1943
('114', '175')	22	33.6	11132
('125', '136')	0	0	1
('126', '167')	0	0	5248
('134', '161')	9	46.5	26726
('142', '161')	10	22.7	13592
('150', '157')	9	26.6	5004
('164', '176')	8	10.3	1412

Total elapsed time for pair comparisons: 240 minutes 38.00 seconds
Time limit for comparing each pair: 1
Amount of pairs: 39
The pairs:

Pair	Timediff	Distance	Rows compared until match
('003', '004')	2	36.8	1
('011', '088')	0	0	1
('012', '127')	0	0	1
('013', '070')	0	0	1
('047', '055')	28	42.5	12688
('051', '104')	18	25.6	9
('051', '110')	14	46.7	16566
('056', '175')	9	29.9	137099
('057', '061')	1	35.7	5617
('057', '094')	0	0	1
('057', '150')	0	0	1
('057', '157')	9	26.6	5004
('061', '094')	24	33.5	36326
('061', '150')	24	33.5	36326
('061', '157')	19	7.3	20001
('061', '162')	8	33.8	179702
('069', '129')	5	1.4	7262
('081', '125')	0	0	1
('081', '136')	0	0	1
('084', '085')	0	0	27612
('092', '101')	8	27.5	137386
('092', '104')	4	31.6	177202
('092', '109')	8	27.5	5956
('094', '150')	0	0	1
('094', '157')	9	26.6	5004
('101', '104')	5	17.6	147675
('101', '109')	0	0	1
('101', '173')	5	40.9	1943
('104', '109')	5	17.6	5361
('104', '173')	11	42.4	7765
('108', '139')	0	0	870
('109', '173')	5	40.9	1943
('114', '175')	22	33.6	11132
('125', '136')	0	0	1
('126', '167')	0	0	5248
('134', '161')	9	46.5	26726
('142', '161')	10	22.7	13592
('150', '157')	9	26.6	5004
('164', '176')	8	10.3	1412

Here you are also able to see that the program compares a fair amount of rows between the users within the small time interval of 0.01 seconds until it finds a match, as shown in the leftmost picture. Increasing the time interval, will increase the amount of rows compared, and thereby increasing the accuracy of the result and may give additional unique pairs not yet discovered. On the other hand, increasing the time interval will increase the overall runtime, so depending on the accuracy of the result you want, choose a suitable time interval. We can see that increasing the time interval from 0.01 to 1.00 second, increases the runtime substantially, from 2 minutes, to 240 minutes. As a result we found 16 new pairs.

Full program can be viewed in python code in file `task2.py`.

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

We used this query:

```
SELECT
    user_id,
    SUM(sq.hightgain) * 0.3048 AS user_highgain
FROM
    (
        SELECT
            ssq.user_id,
            ssq.activity_id,
            SUM(CASE WHEN ssq.diff > 0 THEN ssq.diff ELSE 0 END) AS hightgain
        FROM
            (
```

```
                SELECT
                    user_id,
                    activity_id,
                    altitude,
                    altitude - LAG(altitude) OVER (ORDER BY activity_id) AS diff
                FROM
                    trackpoint
                JOIN
                    activity ON trackpoint.activity_id = activity.id
            ) AS ssq
        GROUP BY
            ssq.activity_id
    ) AS sq
GROUP BY
    sq.user_id
ORDER BY
    user_highgain DESC
LIMIT 15;
```

and got this result with python:

```
Users whw have gained the most altitude (meters):

  User    Altitude gain (m)
------  -------------------
   128               688412
   153               675369
   004               357836
   041               280459
   062               258200
   003               249477
   163               246264
   085               239369
   144               224735
   030               183820
   039               161123
   025               149781
   084               145681
   167               130406
   000               130186
```

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

We solved this task mainly through using python code, and not as a single nested SQL query.

The following query was used identify all different transportation modes:

```
SELECT distinct(transportation_mode) as modes
FROM activity;
```

With this information we looped through the various modes (we ignore mode NULL). For each transportation mode, we used the query below to get all trackpoint with a given mode including a user id to identify a user and activity id to separate the trackpoint into their own activity.

Second query used:

```
SELECT activity.user_id, activity.id, activity.transportation_mode, trackpoint.lat,
trackpoint.lon, trackpoint.date_time
FROM trackpoint
JOIN activity ON activity.id = trackpoint.activity_id
WHERE activity.transportation_mode = "%s";
```

%s is replaced by the current transportation mode.

First, we created a result dataframe to log data. This dataframe consisted of user id, distance, and datetime, with one row per user.

We looped through each user, filtering the data for that user with the goal of finding the distance traveled for a given transportation mode. We split each activity out of the main data, excluding activities that crossed over to a new day. Then, we analyzed each activity by creating coordinates from the latitude and longitude values. We measured the distance between the coordinates of the current row and the previous row using *geopy* to find the distance between both points. The distances were accumulated until we obtained a total distance for the entire activity.

This total distance was then compared with the result dataframe. If the activity's start date for the given user matched the date in the results dataframe, the total distance was added to the existing value. If the dates did not match, but the newly calculated distance was greater, it replaced the result's distance value, and the date was updated to the new date. Otherwise, no changes were made.

lastly we identify the highest distance value in the results dataframe and extract the row with said value, this is then displayed giving this result:

```
Users who have traveled the furthest using a transportation mode
bus      - user: 128, distance: 207.322125km
taxi     - user: 128, distance: 40.175591km
walk     - user: 108, distance: 22.857899km
bike     - user: 128, distance: 52.533836km
car      - user: 128, distance: 398.884409km
run      - user: 062, distance: 0.033292km
train    - user: 062, distance: 277.798911km
subway   - user: 128, distance: 23.282272km
airplane         - user: 128, distance: 1442.088587km
boat     - user: 128, distance: 65.60205km
```

2.11. Find all users who have invalid activities, and the number of invalid activities per use.

We used this query:

```
SELECT activity.user_id, COUNT(DISTINCT(activity.id)) AS 'invalid activities'
FROM (SELECT t1.activity_id AS activity_id, ABS(t2.date_days - t1.date_days) AS
time_diff
FROM  trackpoint t1
JOIN trackpoint t2 ON t1.activity_id = t2.activity_id AND t1.id+1 = t2.id
HAVING time_diff >= 0.00347222) AS subquery
JOIN activity on activity.id = subquery.activity_id
GROUP BY activity.user_id
LIMIT 20;
```

We limited the result to show 20 users and their total amount of invalid activities, due to the result being very large. we got this result with python:

```
All users how have invalid activities (limit 20):
  User    ant invalid
------  -------------
   000            101
   001             45
   002             98
   003            179
   004            219
   005             45
   006             17
   007             30
   008             16
   009             31
   010             50
   011             32
   012             43
   013             29
   014            118
   015             46
   016             20
   017            129
   018             27
   019             31

---------------------------------
```

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

We used this query to get all activities with a transportation mode:

```
SELECT user_id, transportation_mode
FROM activity
WHERE transportation_mode IS NOT NULL;
```

Then we use the dataset with pandas to filter each user and then use the value_count() function in pandas to find the most commonly used transportation mode for the user. We assume the most common transformation mode is the mode most often registered with an activity, if two are equally common we display the activity which is first encountered.

This gives the following result:

```
The users with labeles most common transportation mode:
User: 010, most common transport mode: taxi
User: 020, most common transport mode: bike
User: 021, most common transport mode: walk
User: 052, most common transport mode: bus
User: 056, most common transport mode: bike
User: 058, most common transport mode: taxi
User: 060, most common transport mode: walk
User: 062, most common transport mode: walk
User: 064, most common transport mode: bike
User: 065, most common transport mode: bike
User: 067, most common transport mode: walk
User: 069, most common transport mode: bike
User: 073, most common transport mode: walk
User: 075, most common transport mode: walk
User: 076, most common transport mode: car
User: 078, most common transport mode: walk
User: 080, most common transport mode: taxi
User: 081, most common transport mode: bike
User: 082, most common transport mode: walk
User: 084, most common transport mode: walk
User: 085, most common transport mode: walk
User: 086, most common transport mode: car
User: 087, most common transport mode: walk
User: 089, most common transport mode: car
User: 091, most common transport mode: bus
User: 092, most common transport mode: bus
User: 097, most common transport mode: bike
User: 098, most common transport mode: taxi
User: 101, most common transport mode: car
User: 102, most common transport mode: bike
User: 107, most common transport mode: walk
User: 108, most common transport mode: walk
User: 111, most common transport mode: taxi
User: 112, most common transport mode: walk
User: 115, most common transport mode: car
User: 117, most common transport mode: walk
User: 125, most common transport mode: bike
User: 126, most common transport mode: bike
User: 128, most common transport mode: car
User: 136, most common transport mode: walk
User: 138, most common transport mode: bike
User: 139, most common transport mode: bike
User: 144, most common transport mode: walk
User: 153, most common transport mode: walk
User: 161, most common transport mode: walk
User: 163, most common transport mode: bike
User: 167, most common transport mode: bike
User: 175, most common transport mode: bus
```

See the exact details in our attached python code in file `task2.py`.

Discussion

Things we did differently

We primarily followed the assignment sheet, setting up the virtual machine and database as specified.

On certain queries we limited the number of returned items, as some queries take a considerable amount of time to perform and/or produce a large output. Moreover, this could also be a sign that the queries might need to be reformulated to perform more efficiently, however running time was not of concern to this assignment. A more detailed breakdown of assumptions and/or what we did differently are stated at the relevant task.

Doing mathematical calculations can be time consuming with SQL, and frustrating when doing them repeatedly for debugging. By storing values in a file as a dictionary, we were able to skip repeating the same comparisons and reduce run time. In comparison by just using SQL we were able to find just 1 pair of users who satisfies the conditions of task 2.8 in 10 minutes, while with python we were able to find 23 pairs in just 2.5 minutes. This shows that by doing this differently, by using files and dictionaries, you are able to optimize the program and get a more correct result.

Pain points

We found the tasks from 2.8 and out particularly hard.

Task 2.8 requires an extensive search throughout the database, doing comparisons between all users' trackpoints. This takes a lot of time, and makes the code hard to debug and validate. First we tried just with one big SQL query, but the runtime was too long. For context, it took around 10 minutes to identify a single pair of users who met the task's conditions.

Considering that we had to check pairs for all users, we understood we needed to find a more efficient way to do it. Doing the time difference calculations and haversine through SQL is time consuming.

To optimize the process we split the problems in more parts to later be able to reduce the amount of unnecessary comparisons. Saving all user trackpoints in one dictionary would allow for faster access for the trackpoints. By saving this in a file, we could skip doing this long process again when finding user pairs that satisfy the limits given in the task. This reduced the runtime substantially, and made debugging easier and less time consuming. After this we started comparing with time difference and haversine, and noticed the runtime was still considerable. Therefore we set an additional limit, where it was allowed to find comparisons between one certain pair of users for a given amount of time before it had to check for a new pair. This reduced the runtime a lot, and by allowing the program to compare trackpoints between each user pair for just 0.01 seconds, we were able to find an answer in approximately 2.5 minutes giving us 23 unique user pairs.

We used the file to accelerate the processing, and it can be outdated if the data changes. Therefore this file works for this exact problem. Adding other users, would demand a new file to be made.

Otherwise, we originally aimed to complete the task using pure SQL commands. we quickly found out that this would require us to create intricate nested SQL queries that were difficult to read, and even harder to write. We especially found math heavy tasks painstakingly complicated in SQL. As such, we split these tasks into a data gathering phase and data processing phase. Most of the processing phase was done in python using libraries such as *pandas*, *geopy*, and *pickle*. The SQL queries were used for the data gathering.

Setting up the VM also caused some frustrations, as explained under the Feedback-section in this report.

What we learned

Through this assignment, we learnt how to set up an Ubuntu Virtual Machine. We honed our skills in utilizing Python for MySQL database management, and significantly enhanced our ability to formulate precise and efficient SQL queries. This experience provided us with a deeper comprehension of database architectures, allowing us to grasp the intricacies of organizing and managing data effectively.

A crucial takeaway from this project was the realization that labels do not always align seamlessly with existing data. Initially, we assumed that all labels would neatly correspond with the provided data, but this assumption proved incorrect. We had to filter and discern ourselves which data matched with what label. This experience underscored the importance of recognizing that even professionally presented data may not be flawless or comprehensive.

Calculating distance between two coordinates requires more complex math equations than previously assumed. We were surprised by the intricacies of calculating geographical distance, and quickly discovered that this measure was best solved outside of SQL. Our main takeaway from this experience is to not overly rely on one tool, other tools might be just as good if not better at solving a problem.

Another valuable lesson we learnt was the importance of writing efficient SQL statements. Certain queries we formulated were time-consuming and involved extensive comparisons within the datasets. Drawing from our programming experience, we recognized that nested for-loops can quickly become unwieldy. We realized the importance of filtering the data rather than engaging in exhaustive comparisons, which led to noticeable performance improvements in some cases. Additionally, we found that employing Python was often the optimal approach. It allowed us to pinpoint crucial data within a smaller subset before requesting more data based on our discoveries.

In addition, we understood that storing connecting values in a file and using dictionaries was a much better and faster approach for not repeating unnecessary comparisons, doing lookups and reducing runtime.

Overall, this task has broadened our understanding of real-world database scenarios.

Feedback

We spent several hours setting up the VM due to insufficient instructions provided in the assignment sheet. After seeking help on Piazza, we learned that we needed to comment out specific lines from a file within the VM to get it working. Including this information in the assignment text would have been greatly appreciated, as it would have spared us unnecessary frustration. Alternatively, providing a VM which does not include these code lines.

If the goal is to teach SQL queries it is unwise to create tasks which demand a query with a very long run time. This leads to frustration, as debugging and validation becomes hard and time consuming.