Contents

1 TDT4225 - Report

1.1 Assignment 2 - MySQL

- **Group**: 89
- **Students**: Deidier Simone

1.1.1 Introduction

This project focuses on analyzing the **Geolife GPS Trajectory dataset** using *MySQL* and *Python*. The dataset, which tracks the outdoor movements of **182 users**, requires setting up a database on a *MySQL server* and performing **data cleaning** before insertion. The tasks include creating and defining tables for **users**, **activities**, and **trackpoints**, integrating and cleaning data from the dataset, and ensuring only activities with fewer than **2,500 trackpoints** are included. Additionally, we develop **SQL queries** to extract insights about user activities, transportation modes, distances covered, and altitude changes. The project aims to uncover patterns in user behavior by efficiently managing and querying large amounts of **GPS data**.

1.1.2 Results

- **DATA INSERTION**

As we can see from the results, the time taken to enter data into the database is very high when there are large volumes of data, such as the **TrackPoint table** which contains more than **nine million data points**. The time was greatly reduced for the entry of track points by using a *batch entry* of points instead of a *single point by point entry*, in fact it went from about **1700 seconds taken to enter all points** (*about half an hour*) to just over **180 seconds** (*3 minutes*).

```
simonedeidier@dhcp-10-24-147-88 src % python3 main.py
Connected to: 8.0.32
You are connected to the database: ('vlddv_ass',)
-----------------------------------------------

Creating tables...
Tables created in 0.02 seconds!
Inserting data...
        Inserting User data...
        User data inserted in 0.04 seconds!
        Inserting Activity data...
        Activity data inserted in 34.28 seconds!
        Inserting TrackPoint data...
        TrackPoint data inserted in 180.91 seconds!
Data inserted!
```

First 10 rows of User table:

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

First 10 rows of Activity table:

id	user_id	transportation_mode	start_date_time	end_date_time
1	135		2009-01-03 01:21:34	2009-01-03 05:40:31
2	135		2009-01-02 04:31:27	2009-01-02 04:41:05
3	135		2009-01-27 03:00:04	2009-01-27 04:50:32
4	135		2009-01-10 01:19:47	2009-01-10 04:42:47
5	135		2009-01-14 12:17:57	2009-01-14 12:30:53
6	135		2009-01-12 01:41:22	2009-01-12 02:14:01
7	135		2008-12-24 14:42:07	2008-12-24 15:26:45
8	135		2008-12-28 10:36:05	2008-12-28 12:19:32
9	132		2010-02-15 10:56:35	2010-02-15 12:22:33
10	132		2010-04-30 23:38:01	2010-05-01 00:35:31

First 10 rows of TrackPoint table:

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

- **QUERIES**

As can be seen from the resulting times of the first seven queries, the relational database is **very convenient** when executing **very simple queries** involving a few joins of different tables or counting or displaying data organised in different ways. In fact, the times are practically less than a few seconds, with queries executed almost instantaneously. The relational model is **very powerful** in this respect, since the DBMS allows a **high degree of optimisation** of the search for data in the tables.

Task 2.1: How many users, activities and trackpoints are there in the dataset? (Executed in 1.33 seconds)
Number of users: 182
Number of activities: 16046
Number of trackpoints: 9676756

Task 2.2: Find the average number of activities per user. (Executed in 0.00 seconds)
Average number of activities per user: 92.75

Task 2.3: Find the top 20 users with the highest number of activities. (Executed in 0.00 seconds)

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

```
Task 2.4: Find all users who have taken a taxi. (Executed in 0.01 seconds)
+-----------+
|   User ID |
|-----------|
|       010 |
|       058 |
|       062 |
|       078 |
|       080 |
|       085 |
|       098 |
|       111 |
|       128 |
|       163 |
+-----------+
```

```
Task 2.5: Find all types of transportation modes and count how many activities that are tagged with these transportation mode labels. (Executed in 0.00 seconds)
+---------------------+----------------+
| Transportation Mode |   Activity Count |
|---------------------+----------------|
| walk                |            480 |
| car                 |            419 |
| bike                |            263 |
| bus                 |            199 |
| subway              |            133 |
| taxi                |             37 |
| airplane            |              3 |
| train               |              2 |
| run                 |              1 |
| boat                |              1 |
+---------------------+----------------+
```

```
Task 2.6: Find the year with the most activities. Is this also the year with most recorded hours? (Executed in 0.00 seconds)
Year with the most activities: 2008 with 5894 activities
Year with the most recorded hours: 2009 with 9161 hours (Executed in 0.00 seconds)
No, the year with the most activities is not the year with the most recorded hours.
```

```
Task 2.7: Find the total distance (in km) walked in 2008, by user with id=112. (Executed in 0.06 seconds)
Total distance walked in 2008 by user 112: 115.47 km
```

As for more complex queries that, for example, require **several joins** between different tables (*including the TrackPoint table that contains more than **nine million data***), we can see from the times that the **optimisation of the DBMS** manages to cut the time down to a certain point. In fact, some queries take up to **one minute** to execute.

Task 2.8: Find the top 20 users who have gained the most altitude meters. (Executed in 37.88 seconds)

User ID	Total Altitude Gain (kilometers)
128	2135.03
153	1820.58
004	1089.36
041	789.622
003	766.613
085	714.041
163	673.296
062	595.932
144	587.632
030	576.377
039	481.311
084	430.319
000	398.638
002	377.503
167	370.646
025	357.995
037	325.561
140	311.073
126	272.374
017	205.265

Task 2.9: Find all users who have invalid activities. (Executed in 68.07 seconds)

User ID	Invalid Activity Count
128	720
153	557
025	263
062	249
163	233
004	219
041	201
085	184
003	179
144	157
039	147
068	139
167	134
017	129
014	118
030	112
126	105
000	101
092	101
037	100
084	99
002	98
104	97
034	88
140	86
112	67

```
Task 2.10: Find the users who have tracked an activity in the Forbidden City of Beijing. (Executed in 3.05 seconds)
+-----------+
|   User ID |
|-----------|
|       131 |
|       018 |
|       019 |
|       004 |
+-----------+
```

The **last query** is a good way to cross-check the **correctness** of the data in the relational schema, since the result of the query is the data contained in the file “*labeled_ids.txt*”. In addition, the query is executed via a **sub-query**, demonstrating the **efficiency** of the relational schema and the DBMS to execute several **optimised queries** and merge the data.

```
Task 2.11: Find all users who have registered transportation_mode and their most used transportation_mode. (Executed in 0.00 seconds)
+-----------+--------------------------------+
|   User 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 | bus                            |
|       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                            |
+-----------+--------------------------------+
```

1.1.3 Discussion

As far as the **execution** and **writing of the code** is concerned, the **text of the assignment** is very clear and helps a lot by guiding step by step through the various points to be executed. I did not adopt any methods other than those suggested, as I was comfortable with them due to my experience with both Python and relational databases, particularly MySQL, even with very complex queries. Some **critical points** of this code are certainly, as already mentioned, the **complex queries**, especially those using **window functions** such as **ROW_NUMBER()**, which can be slow on large datasets. Likewise, the insertion of more than **nine million records** into the TrackPoint table is very **slow and memory-intensive**. Although **batch insertion** helps, it may not be sufficient for datasets of this size. A **non-relational schema** might help with some of these critical points, but it also presents challenges. There would certainly be more **data flexibility**: formats such as **XML** and **JSON** allow for more flexibility in data structure, which can be useful for **unstructured or semi-structured data**. In addition, inserting large amounts of data into a **NoSQL database** (*such as MongoDB*) can be faster than a relational database, *especially if the data is already in JSON format.* On the other hand, **complex queries**, such as those using window functions, can be more difficult to execute in a **NoSQL database**. **Aggregation** and **sorting operations** may be less efficient, and above all, many NoSQL databases do not support **ACID transactions**, which can complicate the handling of operations that need to be **atomic**. In my academic experience, I had never worked with a dataset of this size in an SQL database (*I have worked on large datasets, but always imported them into memory as dataframes in Python*). This assignment certainly helped me to better appreciate the ability of such widely used systems, like **SQL DBMSs**, to perform **complex queries** and operations in a **short time** thanks to various **optimisations**. However, I also better understood the **limitations** of these DBMSs, and perhaps found **alternative methods** of addressing these problems (*such as batch data entry into the DBMS*).