

TDT4225 - Very Large, Distributed Data Volumes Assignment 2

Group 85

Fredrik Sundt-Hansen and Daniel Pietrzykowski Sarjomaa

October 2024

Contents

1	Introduction	3
2	Results	4
2.1	Part 1	4
2.2	Task1	5
2.3	Task 2	6
2.4	Task 3	6
2.5	Task 4	7
2.6	Task 5	8
2.7	Task 6a	9
2.8	Task 6b	10
2.9	Task 7	10
2.10	Task 8	12
2.11	Task 9	13
2.12	Task 10	13
2.13	Task 11	14
3	Discussion	16
3.1	Things done differently from task description	16
3.2	Data Insertions	16
3.3	Use of Python and SQL	19
3.4	Use of AI	19
3.5	What we learned	19
4	Appendices	21
A	Part 1 output	21

A Task 1 Output	22
A Task 2 Output	22
A Task 3 Output	22
A Task 4 Output	23
A Task 5 Output	23
A Task 6a Output	24
A Task 6b Output	24
A Task 7 Output	24
A Task 8 Output	24
A Task 9 Output	25
A Task 10 Output	29
A Task 11 Output	30

1 Introduction

This is the second assignment for the course TDT4225 - Very Large, Distributed Data Volumes, completed by Fredrik Sundt-Hansen and Daniel Pietrzykowski Sarjomaa. The assignment is divided into two parts:

- Part 1 involved inserting a dataset of user trajectories into a MySQL database. This process included parsing the data and designing the appropriate database schema for efficient storage and querying.
- Part 2 consisted of 11 tasks, each requiring the execution of SQL queries to retrieve specific information from the database. These tasks tested our ability to write and optimize SQL queries to handle large data volumes effectively.

The assignment focused on understanding the process of data insertion, query formulation, and handling very large, distributed datasets in a database system.

Since part 1 was challenging to split into distinct tasks, the group decided to work on it individually, later coming together to discuss and refine the logic each person had developed. After reviewing the various approaches, the group agreed on the final implementation. For part 2, the workload was divided more evenly: one member handled the even-numbered tasks, while the other focused on the odd-numbered tasks. This division allowed both group members to work in parallel on part 2.

The source code is publicly available in the GitHub Repository. It primarily consists of four main files:

- **geolife_db.py**: This file contains all methods used to interact with the database.
- **geolife_data_handler.py**: This file handles the logic for navigating through the dataset folder and extracting relevant data. The reason behind splitting the code is further explained in section 2.13.
- **main.py**: The main entry point of the application, where both database interaction and data handling functions are used to retrieve and insert data into the database.
- **tasks.py**: This file organizes the tasks of the assignment as unit tests, ensuring correctness and validation of the implemented functionality.

Additionally, there are other supporting files, including Docker-related configuration files (e.g., Docker Compose), and **file_counter.py**, which is used to verify the number of files and track points in the dataset against those stored in the database. The dataset is considered too big to be included in the GitHub repository. Therefore, in order to test this application, one has to copy the dataset over themselves.

2 Results

Below are all the tasks ordered in subsections. It includes a picture of the result, and a code snippet showing the SQL query, and if necessary, an explanation or comment. All tasks include screenshots if the result is small enough to be included into one screenshot, else they are included in appendices, section 4.

2.1 Part 1

Screenshots from part 1, tables after insert, showing the first 10 rows.

User table:

```
mysql> SELECT * FROM user LIMIT 10;
```

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)

Figure 1: User table

Activity table:

```
mysql> SELECT * FROM activity LIMIT 10;
```

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
5	135	NULL	2009-01-14 12:17:57	2009-01-14 12:30:53
6	135	NULL	2009-01-12 01:41:22	2009-01-12 02:14:01
7	135	NULL	2008-12-24 14:42:07	2008-12-24 15:26:45
8	135	NULL	2008-12-28 10:36:05	2008-12-28 12:19:32
9	132	NULL	2010-02-15 10:56:35	2010-02-15 12:22:33
10	132	NULL	2010-04-30 23:38:01	2010-05-01 00:35:31

10 rows in set (0.00 sec)

Figure 2: Activity table

Track point table:

```
mysql> SELECT * FROM track_point LIMIT 10;
```

id	activity_id	lat	lon	altitude	date_days	date_time
68153593	1	39.974294	116.399741	492	39816.0566435185	2009-01-03 01:21:34
68153594	1	39.974292	116.399592	492	39816.0566550926	2009-01-03 01:21:35
68153595	1	39.974309	116.399523	492	39816.0566666667	2009-01-03 01:21:36
68153596	1	39.974320	116.399588	492	39816.0566898148	2009-01-03 01:21:38
68153597	1	39.974365	116.399730	491	39816.0567013889	2009-01-03 01:21:39
68153598	1	39.974391	116.399782	491	39816.0567361111	2009-01-03 01:21:42
68153599	1	39.974426	116.399735	491	39816.0567824074	2009-01-03 01:21:46
68153600	1	39.974458	116.399700	491	39816.0568402778	2009-01-03 01:21:51
68153601	1	39.974491	116.399732	490	39816.0568981481	2009-01-03 01:21:56
68153602	1	39.974530	116.399758	489	39816.0569560185	2009-01-03 01:22:01

10 rows in set (0.00 sec)

Figure 3: Track point table

2.2 Task1

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

Solution:

```
mysql> select count(*) from user;
```

count(*)
182

1 row in set (0.01 sec)

Figure 4: Count all elements in user table

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

```
mysql> select count(*) from activity;
```

count(*)
16048

1 row in set (0.02 sec)

Figure 5: Count all elements in activity table

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

```
mysql> select count(*) from track_point;
+-----+
| count(*) |
+-----+
| 9681756 |
+-----+
1 row in set (0.88 sec)
```

Figure 6: Count all elements in activity table

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

2.3 Task 2

Find the average number of activities per user.

Solution:

```
SELECT AVG(activity_count) AS avg_activities_per_user
FROM (
    SELECT COUNT(*) AS activity_count
    FROM activity
    GROUP BY user_id
) AS user_activity_count;
```

```
mysql> SELECT AVG(activity_count) AS avg_activities_per_user
-> FROM (
->     SELECT COUNT(*) AS activity_count
->     FROM activity
->     GROUP BY user_id
-> ) AS user_activity_count;
+-----+
| avg_activities_per_user |
+-----+
| 92.7630 |
+-----+
1 row in set (0.01 sec)
```

Figure 7: Average number of activities per user

2.4 Task 3

Find the top 20 users with the highest number of activities.

Solution:

```
SELECT u.id, COUNT(*) AS count_activity
FROM user u
JOIN activity a ON u.id = a.user_id
GROUP BY u.id
ORDER BY count_activity DESC
LIMIT 20;
```

```
mysql> SELECT u.id, COUNT(*) AS count_activity
->      FROM user u
->      JOIN activity a ON u.id = a.user_id
->      GROUP BY u.id
->      ORDER BY count_activity DESC
->      LIMIT 20;
```

id	count_activity
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

20 rows in set (0.00 sec)

Figure 8: Top 20 users with the highest number of activities

2.5 Task 4

Find all users who have taken a taxi.

Solution:

```
SELECT DISTINCT user_id
FROM activity
WHERE transportation_mode = 'taxi';
```

Output:

```
mysql> SELECT DISTINCT user_id
-> FROM activity
-> WHERE transportation_mode = 'taxi';
```

user_id
010
058
062
078
080
085
098
111
128
163

```
10 rows in set (0.12 sec)
```

Figure 9: All users who have taken a taxi

2.6 Task 5

Find all types of transportation modes and count how many activities that are tagged with these transportation mode labels. Do not count the rows where the mode is null.

Solution:

```
SELECT transportation_mode, COUNT(*) AS count_transportation_mode
FROM activity
WHERE transportation_mode IS NOT NULL
GROUP BY transportation_mode;
```



```
mysql> SELECT transportation_mode, COUNT(*) AS count_transportation_mode
-> FROM activity
-> WHERE transportation_mode IS NOT NULL
-> GROUP BY transportation_mode;
```

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

```
10 rows in set (0.02 sec)
```

Figure 10: All types of transportation modes and count how many activities that are tagged with these transportation mode labels

2.7 Task 6a

Find the year with the most activities.

Solution:

2008 is the year with most activities.

```
SELECT YEAR(start_date_time) AS year, COUNT(*) AS count
FROM activity
GROUP BY year
ORDER BY count DESC
LIMIT 1;
```

Output:

```
mysql> SELECT YEAR(start_date_time) AS year, COUNT(*) AS count
-> FROM activity
-> GROUP BY year
-> ORDER BY count DESC
-> LIMIT 1;
```

year	count
2008	5895

```
1 row in set (0.01 sec)
```

Figure 11: Year with the most activities

2.8 Task 6b

Is this also the year with most recorded hours?

Solution:

No, that year with most recorded hours is 2009.

```
SELECT YEAR(start_date_time) AS year, SUM(TIMESTAMPDIFF(HOUR,
    start_date_time, end_date_time)) AS hours
FROM activity
GROUP BY year
ORDER BY hours DESC
LIMIT 1;
```

Output:

```
mysql> SELECT YEAR(start_date_time) AS year,
-> SUM(TIMESTAMPDIFF(HOUR, start_date_time, end_date_time)) AS hours
-> FROM activity
-> GROUP BY year
-> ORDER BY hours DESC
-> LIMIT 1;
+-----+-----+
| year | hours |
+-----+-----+
| 2009 | 9165 |
+-----+-----+
1 row in set (0.02 sec)
```

Figure 12: Year with most recorded hours

2.9 Task 7

Find the total distance (in km) walked in 2008, by user with id=112.

Solution:

To find the total distance, SQL and Python was used. One could only use SQL, but doing so require quite a long and extensive SQL query. This way a short and concise query was used, with the precision of float numbers from python.

SQL query:

```
SELECT lat, lon
FROM track_point
JOIN activity ON track_point.activity_id = activity.id
WHERE user_id = '112'
AND transportation_mode = 'walk'
AND YEAR(start_date_time) = 2008
AND YEAR(end_date_time) = 2008
ORDER BY date_time;
```

Python code:

```
def test_task7(self):
    query = """
    SELECT lat, lon
    FROM track_point
    INNER JOIN activity ON track_point.activity_id = activity.id
    WHERE user_id = '112'
    AND transportation_mode = 'walk'
    AND YEAR(start_date_time) = 2008
    AND YEAR(end_date_time) = 2008
    ORDER BY date_time ASC;
    """
    res = self.db.exec_query(query)
    assert len(res) != 0

    previous_point = None
    total_dist = 0
    for lat, lon in res:
        current_point = (lat, lon)
        if previous_point:
            total_dist += haversine(previous_point, current_point,
                                     unit=Unit.KILOMETERS)
            previous_point = current_point

    assert total_dist is not None
    assert total_dist > 0

    print(f"\nTask 7, total distance walked by user 112 in 2008: {
        total_dist:.3f} km")
```

Output:

Task 7, total distance walked by user 112 in 2008: 141.217 km

2.10 Task 8

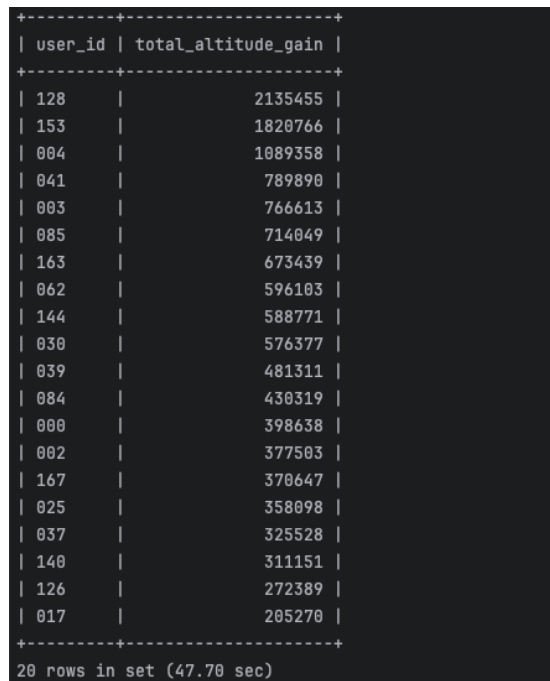
Find the top 20 users who have gained the most altitude meters.

Solution:

```
WITH altitude_differences AS (  
  SELECT a.user_id,  
         (tp_next.altitude - tp_current.altitude) AS altitude_gain  
  FROM track_point tp_current  
  JOIN track_point tp_next  
    ON tp_current.activity_id = tp_next.activity_id  
    AND tp_next.id = tp_current.id + 1  
  JOIN activity a ON tp_current.activity_id = a.id  
  WHERE tp_next.altitude > tp_current.altitude  
)  
SELECT user_id, SUM(altitude_gain) AS total_altitude_gain  
FROM altitude_differences  
GROUP BY user_id  
ORDER BY total_altitude_gain DESC  
LIMIT 20;
```

By having altitude as None if not valid, there is no need to check it in the SQL query.

Output:



user_id	total_altitude_gain
128	2135455
153	1820766
004	1089358
041	789890
003	766613
085	714049
163	673439
062	596103
144	588771
030	576377
039	481311
084	430319
000	398638
002	377503
167	370647
025	358098
037	325528
140	311151
126	272389
017	205270

20 rows in set (47.70 sec)

Figure 13: Top 20 users who have gained the most altitude meters

2.11 Task 9

Find all users who have invalid activities, and the number of invalid activities per user. An invalid activity is defined as an activity with consecutive track point where the timestamps deviate with at least 5 minutes.

Solution:

See appendix A for the complete output of the query.

SQL query:

```
WITH consecutive_track_point AS (  
    SELECT tp_current.activity_id  
    FROM track_point tp_current  
    JOIN track_point tp_next ON tp_next.activity_id = tp_current.  
        activity_id  
    AND tp_next.id = tp_current.id + 1  
    WHERE TIMESTAMPDIFF(Minute, tp_current.date_time, tp_next.date_time)  
        >= 5  
)  
SELECT a.user_id, COUNT(DISTINCT a.id) AS count_invalid_activities  
FROM activity a  
JOIN consecutive_track_point cpt ON a.id = cpt.activity_id  
GROUP BY a.user_id;
```

Using "DISTINCT a.id" to only count unique activities, else it would count occurrences of the same activity multiple times.

2.12 Task 10

Find the users who have tracked an activity in the Forbidden City of Beijing. In this question you can consider the Forbidden City to have coordinates that correspond to: lat 39.916, lon 116.397.

Solution:

This task involved calculating distances to determine if users were within the vicinity of the Forbidden City. Instead of calculating distances directly in SQL, the SQL query was used to retrieve data, and Python's *haversine* library was utilized to compute the distances.

The SQL query extracts track points and uses rounding to reduce latitude and longitude to three decimal places. This prevents small variations in precision from blocking a match and is more effective at capturing points that are close enough to the specified coordinates, ensuring that nearby locations within the same area are included.

```
SELECT user_id, lat, lon  
FROM track_point tp  
JOIN activity a ON a.id = tp.activity_id  
WHERE ROUND(lat, 3) = 39.916  
AND ROUND(lon, 3) = 116.397;
```

Using exact coordinates to determine if a user visited the Forbidden City would be inaccurate because users may not have tracked activities exactly at those coordinates. To address this, an area-based approximation was used. The Forbidden City covers approximately 72 hectares, or 0.72 square kilometers (source). The square root of 0.72 is roughly 0.849, meaning a radius of 0.424 km (half the square root) was chosen as the threshold for determining if a user has visited the Forbidden City.

Using the *haversine* formula, the distance between each track point and the coordinates of the Forbidden City was calculated, with a threshold of 0.424 km to identify relevant users:

```
def test_task10(self):
    query = """
    SELECT user_id, lat, lon
    FROM track_point tp
    JOIN activity a ON a.id = tp.activity_id
    WHERE ROUND(lat, 3) = 39.916
    AND ROUND(lon, 3) = 116.397;
    """

    res = self.db.exec_query(query)
    assert len(res) > 0

    forbidden_city = (39.916, 116.397)
    valid_users = set()

    for user_id, lat, lon in res:
        current_point = (lat, lon)
        dist = haversine(forbidden_city, current_point, Unit.KILOMETERS)
        if dist <= 0.424:
            valid_users.add(user_id)

    print(f"\nTask 10, number of users who have been in The Forbidden
    City of Beijing: {len(valid_users)}\n"
          f"Users: {valid_users}")
```

Output

```
Task 10, number of users who have been in
The Forbidden City of Beijing: 3
Users: {'131', '018', '004'}
```

2.13 Task 11

Find all users who have registered transportation_mode and their most used transportation_mode. The answer should be on format (user_id, most_used_transportation_mode) sorted on user_id. Some users may have the same number of activities tagged with e.g. walk and car. In this case it is up to you to decide which transportation mode to include in your answer (choose one). Do not count the rows where the mode is null.

Solution:

See appendix A for the complete output of the query.

The group chose alphabetical in the case where a user has the same number of activities tagged with multiple modes (e.g., walk and car)

SQL query:

```
WITH user_transportation_modes AS (  
  SELECT  
    user_id, transportation_mode, COUNT(transportation_mode) AS  
      transport_mode_count  
  FROM activity  
  WHERE transportation_mode IS NOT NULL  
  GROUP BY user_id, transportation_mode  
) ,  
most_used_modes AS (  
  SELECT user_id, transportation_mode, transport_mode_count,  
    ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY  
      transport_mode_count DESC) AS rn  
  FROM user_transportation_modes  
)  
SELECT user_id, transportation_mode AS most_used_transportation_mode  
FROM most_used_modes  
WHERE rn = 1  
ORDER BY user_id;
```

If a user has 5 activities tagged as walk and 5 tagged as car, since the counts are the same, the ROW_NUMBER() function will break the tie by selecting car over walk because car comes before walk alphabetically.

3 Discussion

3.1 Things done differently from task description

While the task description was quite clear, there were some assumptions and choices made under the development:

- **Handling of altitude values:** When inserting data into the `track_point` table, if the altitude was recorded as `-777` (an invalid value), we chose to insert the data but set the altitude value to `NULL`. This decision was made to preserve the integrity of the other data points while acknowledging that the altitude was not valid.
- **Indexes and optimization:** The assignment description did not specifically mention creating additional indexes for performance optimization. The group assumed that indexes were not required to complete the tasks, as query optimization was not emphasized. However, had performance become a significant concern, adding indexes on frequently queried fields would have been considered.
- **Retention of `date_days` column:** Although the `date_days` column in the `track_point` table was not utilized in any of the assigned tasks, the group decided to retain this column. The reason was that this information might be relevant in future assignments or queries, and removing it prematurely could lead to data loss.
- **Python for calculations, SQL for data retrieval:** In tasks requiring distance calculations, we used SQL to retrieve the necessary data points and Python (via the `haversine` library) to calculate distances. This hybrid approach simplified implementation, but also raised the question of why not perform the calculations entirely in SQL. The decision to use Python for calculations was made due to its ease of implementation and flexibility, though it could have been done with more complex SQL queries.

3.2 Data Insertions

Perhaps the most challenging aspect of this assignment was ensuring the correct insertion of data from the dataset. Although the final solution was successful, arriving at it required considerable effort. One notable deviation from the example code provided was that we decided to create our own files rather than rely on `example.py`.

Specifically, we structured the code by creating two separate files: one for handling database operations (`geolife-db.py`) and another for retrieving and processing the dataset (`geolife_data_handler.py`). This approach was guided by best coding practices, particularly separation of concerns, which made debugging easier and ensured better code maintainability. Additionally, should this code need to be reused in future assignments, the modular structure would allow for easier modifications and extensions.

Initially, the data handler code was quite messy and difficult to read. However, after several rounds of refactoring, the code became more streamlined, easier to understand, and more efficient to debug. This process was necessary due to the complexity involved in applying filters, cleaning the data, and determining the most efficient way to insert large amounts of data into the database. Ultimately, this restructuring improved the overall clarity and functionality of the code.

Another important consideration during data insertion was the creation and structuring of the database tables. The group followed a design closely inspired by the assignment description, maintaining three core tables (user, activity, and track_point) and preserving all suggested fields. As mentioned earlier, we retained the date_days column even though it wasn't used in the current tasks, allowing for potential future queries or assignments that might require this data.

One challenge we encountered was determining the most suitable primary keys for the activity and track_point tables. Specifically, we had to ensure that each track_point was associated with the correct activity. Initially, we considered using an auto-increment integer as the primary key for the activity table. However, to simplify tracking and ensure accuracy, we opted for a manual incrementing integer for the activity_id field.

As shown below, before processing the activity and track_point data, an activity_id counter was manually incremented. This allowed us to easily maintain a consistent relationship between activities and their respective track points:

```
activity_id += 1
first_line = lines[0]
last_line = lines[-1]
process_activity_data(activity_data, activity_id, user_id,
    labeled_timestamp, first_line, last_line)
process_track_point_data(track_point_data, activity_id, lines)
```

This manual increment ensured that the activity_id assigned to each activity was correctly and consistently associated with the related track points. By incrementing the activity_id counter prior to data insertion, we could reliably match activities and track points across the dataset.

For the track_point table, however, we utilized AUTO_INCREMENT for the primary key, as no other data was directly dependent on individual track points. This decision proved useful, particularly in tasks like 8 and 9, where we needed to join tables and take advantage of the fact that the AUTO_INCREMENT values for the track points increased sequentially. This approach allowed us to write efficient queries that rely on consecutive id values, as shown below:

```
FROM track_point tp_current
JOIN track_point tp_next
ON tp_current.activity_id = tp_next.activity_id
AND tp_next.id = tp_current.id + 1
```

The only thing to consider is the maximum size of an INT limits the number of unique values you can store. In MySQL, the INT data type can store

values from -2,147,483,648 to 2,147,483,647, meaning you can have a maximum of approximately 2.1 billion rows if you are using an unsigned INT AUTO_INCREMENT as the primary key. Once you exceed this limit, you would no longer be able to insert new rows using the AUTO_INCREMENT mechanism. However, this assignment did not need over two billion rows anyway.

A key challenge we faced was the bulk insertion of many track points. Initially, we attempted to insert all the data at once using the `executemany` function (`cursor.executemany(query, data)`). However, this approach caused the MySQL database to disconnect, as we exceeded the default maximum packet size (16 MB).

A simple solution would have been to increase the max packet size in the MySQL configuration. However, this approach would only work in the short term. If the dataset were to grow (e.g., if an additional 9 million track points were added), we would have to increase the packet size again, which is not a scalable solution.

To address this issue, we decided to batch the data inserts by calculating the size of a single row in the `track_point` table and dividing the total by the default packet size. This way, the data could be inserted in smaller, manageable batches, as shown in the code snippet below:

```
"""
One row track_point table =
id activity-id lat lon altitude date_days date_time
INT(4 bytes) + INT(4 bytes) + DOUBLE(8 bytes) + DOUBLE(8 bytes) + INT
(4 bytes) + DOUBLE(8 bytes) + DATETIME(8 bytes)
= 44 bytes

Default max packet size in MySQL = 16 MB

Batch size = 16 MB / 44 B = 381300
"""
def bulk_insert_track_point(self, data: list, batch_size: int = int
(381300)):
    query = f"INSERT IGNORE INTO {TRACK_POINT_TABLE_NAME} ({
        TRACK_POINT_TABLE_INSERT}) VALUES (%s, %s, %s, %s, %s, %s)"
    for i in range(0, len(data), batch_size):
        batch_data = data[i:i + batch_size]
        self.cursor.executemany(query, batch_data)
        self.db_connection.commit()
```

By calculating the batch size based on the row size and MySQL's packet size limit, this solution is scalable and capable of handling large datasets without exceeding MySQL's packet size limit. This approach provides a more robust method for handling large data inserts, regardless of how much the dataset might grow in the future.

3.3 Use of Python and SQL

One of the key decisions made was to combine SQL for data retrieval and Python for calculations, rather than relying solely on SQL. Python's extensive library ecosystem, such as the *haversine* function for calculating distances, made it an ideal choice for certain tasks that involve more complex logic. This is evident in Task 10, where we calculated whether users had been within the radius of the Forbidden City in Beijing. Although SQL could handle some spatial computations, Python provided better flexibility for handling the approximations required for the task.

However, this approach also posed a challenge: it involved shifting between SQL and Python, which could potentially add overhead and complexity to the workflow. Using SQL for all calculations might have streamlined some processes, but it would have required additional time and resources to implement complex calculations that Python handles easily.

3.4 Use of AI

In Part 2 of the assignment, AI helped us troubleshoot and fix various errors in our code, especially when working with database queries. AI was used to fix various syntax issues, ineffective code and other problems in some of the part 2 tasks. Additionally, tools like ChatGPT were used to refine and enhance the clarity and flow of certain written paragraphs. However, it is important to emphasize that AI was not used to solve any of the tasks or generate results. The group fully stands behind all the solutions presented in this report.

3.5 What we learned

The group learned a lot of about data insertion, how the process work, and how difficult it can be to get it right. There were numerous times we got different amount of activities and track points in the database by changing the logic of dataset traversing.

Additionally, we learned a lot of creating and refining queries. How to attack the problem, initially we tried to tackle problems by writing complex queries in one go, but we quickly learned that using a divide-and-conquer strategy was more effective. By breaking down each problem into smaller, testable parts, we could isolate issues, test intermediate results, and gradually build towards the final solution. This method proved to be particularly useful for challenging tasks, such as Task 7, 9, 10, and 11, which required several iterations to get correct.

Furthermore, the group could have adopted another strategy for diving the work. Instead of both doing task 1 simultaneously, one could have worked on part 1 while the other on part 2. Since the queries would hardly change based on the fixed structure of the dataset and tables suggested in the task description, as long as the group did not decide to create additionally tables. However, the

process we used worked quite well, and the group is satisfied with solutions for data handling and queries.

4 Appendices

Below are all the outputs from all the tasks. Some have only their outputs here, like task 9 and 11, which were too big to fit into one screenshot.

A Part 1 output

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

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
5	135	NULL	2009-01-14 12:17:57	2009-01-14 12:30:53
6	135	NULL	2009-01-12 01:41:22	2009-01-12 02:14:01
7	135	NULL	2008-12-24 14:42:07	2008-12-24 15:26:45
8	135	NULL	2008-12-28 10:36:05	2008-12-28 12:19:32
9	132	NULL	2010-02-15 10:56:35	2010-02-15 12:22:33
10	132	NULL	2010-04-30 23:38:01	2010-05-01 00:35:31

track_point table:

id	activity_id	lat	lon	altitude	date_days	date_time
87517105	1	39.974294	116.399741	492	39816.0566435185	2009-01-03

87517106	1	39.974292	116.399592	492	39816.0566550926	2009-01-03
87517107	1	39.974309	116.399523	492	39816.0566666667	2009-01-03
87517108	1	39.974320	116.399588	492	39816.0566898148	2009-01-03
87517109	1	39.974365	116.399730	491	39816.0567013889	2009-01-03
87517110	1	39.974391	116.399782	491	39816.0567361111	2009-01-03
87517111	1	39.974426	116.399735	491	39816.0567824074	2009-01-03
87517112	1	39.974458	116.399700	491	39816.0568402778	2009-01-03
87517113	1	39.974491	116.399732	490	39816.0568981481	2009-01-03
87517114	1	39.974530	116.399758	489	39816.0569560185	2009-01-03

A Task 1 Output

```

+-----+
| COUNT(*) |
+-----+
|      182 |
+-----+

```

```

+-----+
| COUNT(*) |
+-----+
|    16048 |
+-----+

```

```

+-----+
| COUNT(*) |
+-----+
|   9681756 |
+-----+

```

A Task 2 Output

```

+-----+
| avg_activities_per_user |
+-----+
|                92.7630 |
+-----+

```

A Task 3 Output

```

+-----+-----+
| id | count_activity |
+-----+-----+

```

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

-----+

20 rows in set (0.01 sec)

A Task 4 Output

-----+
user_id
-----+
010
058
062
078
080
085
098
111
128
163
-----+

10 rows in set (0.11 sec)

A Task 5 Output

-----+

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

10 rows in set (0.01 sec)

A Task 6a Output

year	count
2008	5895

A Task 6b Output

year	hours
2009	9165

A Task 7 Output

Task 7, total distance walked by user 112 in 2008: 141.217 km

A Task 8 Output

user_id	total_altitude_gain
128	2135455
153	1820766
004	1089358

041		789890	
003		766613	
085		714049	
163		673439	
062		596103	
144		588771	
030		576377	
039		481311	
084		430319	
000		398638	
002		377503	
167		370647	
025		358098	
037		325528	
140		311151	
126		272389	
017		205270	

-----+

20 rows in set (51.70 sec)

A Task 9 Output

user_id		count_invalid_activities	
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	

020		20	
021		7	
022		55	
023		11	
024		27	
025		263	
026		18	
027		2	
028		36	
029		25	
030		112	
031		3	
032		12	
033		2	
034		88	
035		23	
036		34	
037		100	
038		58	
039		147	
040		17	
041		201	
042		55	
043		21	
044		32	
045		7	
046		13	
047		6	
048		1	
050		8	
051		36	
052		44	
053		7	
054		2	
055		15	
056		7	
057		16	
058		13	
059		5	
060		1	
061		12	
062		249	
063		8	
064		7	
065		26	
066		6	

067		33	
068		139	
069		6	
070		5	
071		29	
072		2	
073		18	
074		19	
075		6	
076		8	
077		3	
078		19	
079		2	
080		6	
081		16	
082		27	
083		15	
084		99	
085		184	
086		5	
087		3	
088		11	
089		40	
090		3	
091		63	
092		101	
093		4	
094		16	
095		4	
096		35	
097		14	
098		5	
099		11	
100		3	
101		46	
102		13	
103		24	
104		97	
105		9	
106		3	
107		1	
108		5	
109		3	
110		17	
111		26	
112		67	

113		1	
114		3	
115		58	
117		3	
118		3	
119		22	
121		4	
122		6	
123		3	
124		4	
125		25	
126		105	
127		4	
128		720	
129		6	
130		8	
131		10	
132		3	
133		4	
134		31	
135		5	
136		6	
138		10	
139		12	
140		86	
141		1	
142		52	
144		157	
145		5	
146		7	
147		30	
150		16	
151		1	
152		2	
153		557	
154		14	
155		30	
157		9	
158		9	
159		5	
161		7	
162		9	
163		233	
164		6	
165		2	
166		2	

167		134	
168		19	
169		9	
170		2	
171		3	
172		9	
173		5	
174		54	
175		4	
176		8	
179		28	
180		2	
181		14	

-----+

171 rows in set (48.01 sec)

A Task 10 Output

SQL output:

-----+
user_id lat lon
-----+
131 39.916445 116.396603
131 39.916335 116.396623
131 39.916237 116.396640
131 39.916133 116.396640
131 39.916037 116.396620
131 39.915933 116.396618
131 39.915842 116.396645
131 39.915758 116.396605
131 39.915668 116.396580
131 39.915568 116.396598
018 39.916195 116.396508
018 39.916055 116.396537
018 39.915917 116.396535
018 39.915812 116.396553
018 39.915710 116.396587
018 39.915578 116.396602
018 39.916368 116.396570
018 39.916208 116.396580
018 39.916108 116.396593
018 39.916003 116.396597
018 39.915910 116.396597
018 39.915818 116.396603
018 39.915723 116.396613

018	39.915630	116.396620	
018	39.915763	116.396927	
018	39.916388	116.397207	
018	39.916198	116.397188	
018	39.916077	116.397155	
018	39.915923	116.397163	
018	39.915795	116.397123	
018	39.915678	116.397090	
018	39.915558	116.397060	
004	39.915737	116.396561	
004	39.916239	116.396568	

-----+

34 rows in set (1.56 sec)

Python output:

Task 10, number of users who have been in The Forbidden City of Beijing: 3
Users: {'131', '004', '018'}

A Task 11 Output

user_id	most_used_transportation_mode
010	taxi
020	bike
021	walk
052	bus
056	bike
058	car
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	

+-----+

48 rows in set (0.01 sec)