

# Report

## Assignment 3 - MongoDB

**Group:** 69

**Students:** Hauk Aleksander Olaussen, Vidar Michaelsen, Noran Baskaran

### Disclaimer

In Assignment 2 we used the wrong way of finding the size of an activity. When we were to check if the file length was over 2500 lines (trackpoints), we used `pandas.DataFrame.size` which does not give the proper line count, but the number of elements in the data frame. This caused our preprocessor to exclude a lot of data that should have been included. We have since fixed this issue in this assignment – and all the data should be present. You can see the place where we went wrong on line 117 in `Preprocessor.py`.

### NOTE!

The queries and solutions work as intended when all the data is present.

### Introduction

We used the same file architecture as we did in Assignment 2.

The code for preprocessing of the data can be found in the python file `Preprocessor.py`. This class will read the data from the provided dataset, and place the preprocessed versions of it into two massive lists of dictionaries – which the `DbHandler` will later use to insert the data to MongoDB. We have decided to only use two collections this time – one for users, and one for trackpoints. As none of the activities are shared by multiple users, this makes sense. We initially tried to only use one collection (users) with the trackpoints being an attribute of the activities as well. More on this in the discussion part.

All the answers for part 2 can be found within the `Queries.py`. We solved the tasks by relying heavily on python. Most of the solutions fetches the data needed to solve the task and uses python code almost exclusively to find the result. Running the `Queries.py` file will run all the task functions and print their answers to the terminal. Images and runtimes for each function/query can be found in the next section.

For this assignment we have worked together physically at campus in a very similar fashion as Assignment 2. Hauk and Noran pair programmed the whole part 1 of the

assignment and sent the data to a database which is (once again) located at his home desktop.

For part 2 of the assignment, we all worked together for all the solutions, with most of them solved using almost only python. We tried as best we could to find efficient and scalable solutions.

Link to repo: [GitHub](#)

To connect to the database, fill a `.env` file with the following data:

HOST=84.202.106.55

## Results from Part 1

Image below shows the user collections. This collection also contains all the activities that belongs to the user.

🏠 users			
	<code>_id</code> String	<code>has_labeled</code> Boolean	<code>activities</code> Array
1	"000"	false	[] 155 elements
2	"001"	false	[] 57 elements
3	"002"	false	[] 146 elements
4	"003"	false	[] 261 elements
5	"004"	false	[] 346 elements
6	"005"	false	[] 73 elements
7	"006"	false	[] 24 elements
8	"007"	false	[] 40 elements
9	"008"	false	[] 23 elements
10	"009"	false	[] 37 elements

This image shows how the activity object is structured inside the users.

```

_id: "000"
has_labeled: false
activities: Array
  0: Object
    _id: "20081023025304000"
    transportation_mode: null
    start_date_time: 2008-10-23T02:53:04.000+00:00
    end_date_time: 2008-10-23T11:11:12.000+00:00
  1: Object
  2: Object
  3: Object
  4: Object
  5: Object

```

This image the first 10 trackpoints from our collection trackpoints.

#	trackpoints					
	_id ObjectId	activity_id String	location Object	altitude Int32	date_days Double	date_time D
1	616f0e75df96e8b9b2430c2a	"20081023025304000"	{ } 2 fields	492	39744.1201851852	2008-10-23T
2	616f0e75df96e8b9b2430c2b	"20081023025304000"	{ } 2 fields	492	39744.1202546296	2008-10-23T
3	616f0e75df96e8b9b2430c2c	"20081023025304000"	{ } 2 fields	492	39744.1203125	2008-10-23T
4	616f0e75df96e8b9b2430c2d	"20081023025304000"	{ } 2 fields	492	39744.1203703704	2008-10-23T
5	616f0e75df96e8b9b2430c2e	"20081023025304000"	{ } 2 fields	492	39744.1204282407	2008-10-23T
6	616f0e75df96e8b9b2430c2f	"20081023025304000"	{ } 2 fields	493	39744.1204861111	2008-10-23T
7	616f0e75df96e8b9b2430c30	"20081023025304000"	{ } 2 fields	493	39744.1205439815	2008-10-23T
8	616f0e75df96e8b9b2430c31	"20081023025304000"	{ } 2 fields	496	39744.1206018519	2008-10-23T
9	616f0e75df96e8b9b2430c32	"20081023025304000"	{ } 2 fields	500	39744.1206597222	2008-10-23T
10	616f0e75df96e8b9b2430c33	"20081023025304000"	{ } 2 fields	505	39744.1207175926	2008-10-23T

This image shows how the trackpoints are structured.

```

_id: ObjectId("616f0e75df96e8b9b2430c2a")
activity_id: "20081023025304000"
location: Object
  type: "Point"
  coordinates: Array
    0: 116.318417
    1: 39.984702
altitude: 492
date_days: 39744.1201851852
date_time: 2008-10-23T02:53:04.000+00:00

```

## Result from Part 2

### Task 1:

Screenshot of terminal showing result of function: `count_all_entries()`.

```

TASK 1

Total amount of users: 182
Total amount of activities: 16048
Total amount of trackpoints: 9681756

Time used: 4.20349 seconds

```

### Task 2:

Screenshot of terminal showing result of function: `average_max_min()`.

```

TASK 2

Minimum number of activitites: 0
Maximum number of activitites: 2102
Average number of activitites: 88.17582417582418

Time used: 1.50605 seconds

```

### Task 3:

Screenshot of terminal showing result of function: `top_10_users()`.

```
TASK 3

1| User 128 has 2102 stored activities
2| User 153 has 1793 stored activities
3| User 025 has 715 stored activities
4| User 163 has 704 stored activities
5| User 062 has 691 stored activities
6| User 144 has 563 stored activities
7| User 041 has 399 stored activities
8| User 085 has 364 stored activities
9| User 004 has 346 stored activities
10| User 140 has 345 stored activities

Time used: 1.48831 seconds
```

### Task 4:

Screenshot of terminal showing result of function: `started_one_day_ended_next()`.

```
TASK 4

Amount of users with activities ending the day after starting: 98

Time used: 1.4942 seconds
```

### Task 5:

Screenshot of terminal showing result of function: `duplicate_activities()`.

```
TASK 5

Returned a list of lists containing id of activities with matching 'transportation_mode', 'start_date_time'
and 'end_date_time'
It contains 718 elements

Time used: 30.13804 seconds
```

### Task 6:

Screenshot of terminal showing result of function: `covid_19_tracking()`.

```
TASK 6

User 073 has been within 100m and one minute

Time used: 1.56363 seconds
```

### Task 7:

Screenshot of terminal showing result of function: `user_no_taxi()`.

```
TASK 7

Returned a list containing 172 user ids

Time used: 1.02337 seconds
```

### Task 8:

Screenshot of terminal showing result of function: `transportation_mode_usage()`.

```
TASK 8

bus has been used by 14 users
taxi has been used by 12 users
walk has been used by 36 users
bike has been used by 21 users
car has been used by 11 users
run has been used by 3 users
train has been used by 4 users
subway has been used by 6 users
airplane has been used by 4 users
boat has been used by 3 users

Time used: 1.4371 seconds
```

### Task 9: Screenshot of terminal showing result of function:

`year_month_most_activities()` and `user_with_most_activities_11_2008()`.

```
TASK 9a

The year and month with most activities was month number 11 in year 2008 with a total of 1006 activities

Time used: 2.0786 seconds

TASK 9b

1| User 062 has the most activities in 11-2008, with 130 activities and a total of 47.3136 hours
2| User 128 has the second most activities in 11-2008, with 75 activities and a total of 68.2211 hours, which is more than
   number one

Time used: 2.14182 seconds
```

### Task 10:

Screenshot of terminal showing result of function: `user_112_walk_2008()`.

```
TASK 10

User 112 walked 117.84035175872275 km in 2008

Time used: 3.77091 seconds
```

### Task 11:

Screenshot of terminal showing result of function: `top_20_users_altitude()`.

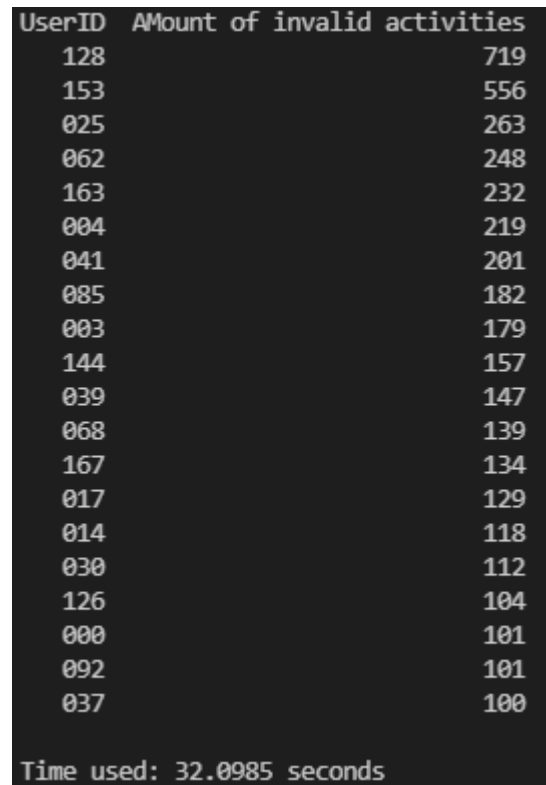
```
UserID  Altitude gained in meters
128      650951.997280
153      554960.623053
004      332036.318400
041      240768.865680
003      233663.642400
085      217643.384880
163      205274.370464
062      181693.291680
144      179441.524480
030      175679.709600
039      146703.592800
084      131161.231200
000      121504.862400
002      115062.914400
167      112974.154640
025      109158.572640
037       99234.589440
140       94846.299360
126       83025.835760
017       62581.353120

Time used: 33.705 seconds
```

**Task 12:**

Screenshot of terminal showing result of function:

`year_invalid_activities()`.



UserID	Amount of invalid activities
128	719
153	556
025	263
062	248
163	232
004	219
041	201
085	182
003	179
144	157
039	147
068	139
167	134
017	129
014	118
030	112
126	104
000	101
092	101
037	100

Time used: 32.0985 seconds

**Discussion****Part 1: Data preprocessing and insertion**

As MongoDB is a NoSQL database using the document model we were – as stated in the introduction – able to combine the users and activities in one document. This differs from the SQL way of doing it, as the activities were located in their own table using foreign keys. The way of structuring the user and activity objects themselves is not changed however, but the activities are contained inside a user's `activities` array. We have changed the way the trackpoints are structured. Instead of having `lat` and `lon` as their own attributes, we have combined them using MongoDB's own GeoJSON objects. This allows for more intricate queries, where one can get information about distance and other geographical data directly in the queries. It is also possible to directly plot the data on a map using MongoDB charts using this datatype.

As stated in the introduction, we initially wanted to try to combine all the data into one collection – as the all the data has a one-to-many structure. We were not allowed to this however, as the maximum size of a document in MongoDB is 16MB. Hence,

---

we decided to combine the users and activities, and leave the trackpoints in their own collection with a key to the activity it belongs to.

## Part 2: Queries and code

Compared to when using SQL we relied more on the application code in python when using MongoDB. This is because some of the complex queries that are easy to find in SQL, are very hard to do in MongoDB. This is because joins are a real pain when using MongoDB, and the NoSQL databases in general. However, this did not make the solutions harder to find, but simply relies more on the computational power of the computer running the code. It also means that most often, the actual MongoDB query returns more unnecessary data from the database than its SQL counterpart. We know that we could have solved more of the tasks can be solved by making more use of aggregation and more complex queries but decided that writing a few lines of code in python made for a simpler solution as it is more readable, and understandable than a query. Task 1 and 7 are examples where we solved the task using just queries.

As in Assignment 2, for task 5, we assume that “duplicate activities” mean that they have the same `start_date_time`, `end_date_time` and `transportation_mode`, as these are the only attributes with actual meaning for the activity itself. We do not check activities within one user, as no users has duplicates. We know this, as they would then share the `_id` attribute, as this is the start date combined into a number. We also do not check activities two ways – meaning that an activity is only counted one time.

As tasks 11 and 12 made us fetch almost all the trackpoints, we created an index in the database on the `activity_id` field, significantly reducing the time it took to solve these tasks.

Note that all the solutions are commented directly in the code. This will give a more rounded understanding of how the task at hand was solved.

For us, the biggest difference between using the MySQL and MongoDB was that this assignment involved a lot more application-level code than the last one. Because of this, we preferred a NoSQL database as we are more familiar with python code. Some tasks that were very easy to solve with the join operation in MySQL, was now taking a lot of resources and lines of code (task 11 and 12). With MongoDB we also did not have to define a schema like we did in MySQL, we just inserted the objects directly from the lists of dictionaries. For this assignment we think MySQL is better because it is more powerful when handling structured data with a clear schema. This fits the assignment dataset and tasks better in our opinion, as many of the tasks are highly relational.