TDT4225 Assignment 3

Group: 78 Students: Simen Omholt-Jensen

October 19, 2021

Introduction

In this assignment, the Geolife project dataset was parsed from files, similarly as in the previous assignment, and inserted into a MongoDB database by the use of the PyMongo python package. The design of the collections is summarized in Listings 1,2, and 3, as well as in tables 1,2, and 3 found in the appendix. A public repository containing the source code and a description of how to reproduce the results can be found at Github.

After insertion, a combination of MongoDB queries and data manipulation in python was utilized to answer the given questions. This report summarizes the results, and describes some of the methods used for data cleaning before insertion into the database. These data cleaning steps were performed based on the group's interpretation of the given assignment text, and is similar to the data processing steps used in assignment 2. Finally, this report highlights some of the differences between the two databases used in assignment 2 and assignment 3.

As the group consisted of a single member, there was no need for work coordination. Some of the work from assignment 2 could be reused, for example work needed for setting up the code structure and the parsing of the data files. This allowed the group to quickly start focusing on the challenging aspect of the assignment, namely getting to know the inner working of MongoDB. As the group had no previous experience working with this database, a considerable amount of time was spent reading up on the documentation and understanding how to pick an efficient collection design. Even though this resulted in more time spent on assignment 3 than on assignment 2, the single group member was still content with his decision of going at it alone.

Results

The tables and figures containing the answers to the questions can be found in the appendix. Questions 1 through 10 have the exact same answers as those found in assignment 2. However, question 11 and question 12 differ.

For question 11, the group forgot to convert the altitudes from feet to meters in assignment 2. As such, the answers provided here are correctly converted into meters. More interestingly, an insight was discovered about the nature of the altitude data. In assignment 2, the altitude data was modeled as integers, but it turns out that the dataset consists of a considerable amount of altitude decimal values. As such, the altitude data was modeled with doubles in assignment 3, which lead to a second difference in the answers provided in the two assignments. An example of the difference in altitude values in the two databases can be seen in figure 1 below. The resulting difference of the number of elevation gain in meters can be seen in figure 2.

activity_id	trackpoint_id	altitude	from_database
1244	1028931	85.30	MongoDB
1244	1028931	85.00	MySQL

Figure 1: Different modeling of altitude values in MySQL and MongoDB.

user_id	total_elevation_gain	from_database
128	650969.28	MySQL
128	650952.00	MongoDB
153	554964.60	MySQL
153	554960.62	MongoDB
004	332036.32	MongoDB
004	332036.32	MySQL

Figure 2: Difference in total elevation gained in meters between the two databases.

For question 12, the group realized that in assignment 2, the difference between datetime values was not calculated correctly. In order to get the correct difference between two datetime values in seconds, it is necessary to use the function `TIMESTAMPDIFF()`. A single subtraction of two timestamps in MySQL will overestimate the number of seconds between the datetime values as it calculates the difference using the centesimal numerical system and not the sexagesimal

numerical system. MongoDB datetime subtraction yields the correct answer out of the box, but when comparing answers between the two assignments, the group realized their mistake in assignment 2.

Discussion

Part 1

- First, the `main.py` script queries the user for its local MongoDB server login information.
- Then, the `main.py` script calls the `create_user` function in the `database.py` module to connect to a local MongoDB server by using the `pymongo` package. The function then proceeds to drop any database with the name `TDT4225ProjectGroup78` if it exists, then creates a new database with the same name, and finally adds a user with the given login information for the created database.
- Next, the `main.py` script calls the `insert_data` function in the `database.py` module which again calls the `parse_data` function to read the data from the `.plt` files into Pandas DataFrames.
- The `parse_data` function finds the user ids from the folder names and stores them as strings as some of the user ids are prefixed with zeros (e.g. '007'). Information about which users that have labels is then read from the `labeled_ids.txt` file and stored as a boolean value together with the user ids in the `user_df` Pandas DataFrame.
- The `parse_data` function then loads each `.plt` file, stores an activity record as a Pandas Series object, and stores the trackpoint information as a Pandas DataFrame. These are then appended to their respective lists.
- Special care had to be taken to ensure that no trackpoint with more than 2500 entries were read. Further, the user `'020'` had labeled an activity twice, with labels walk and bike. To ensure that all the data was reflected in the database. The group decided to create separate activities for both labels, as this ensures that the database reflects the recorded data.
- The lists containing information about each activity and corresponding trackpoint information is then concatenated into Pandas DataFrames. One for the activities, and one for the trackpoints.
- Further, the altitude value of `-777` was replaced by `NULL` values.
- A list of associated `activity_id` were added for each row in the user DataFrame.
- Finally, the DataFrames were converted into record dictionaries that could be inserted into the MongoDB database collections.
- Once the data is parsed, the `insert_data` establishes a connection to the local MongoDB server by using the Python package `pymongo`, and inserts the data.

Part 2

- On completion of the data insertion, the `main.py` script calls the `query_database` function in the `database.py` module. It establishes a connection to the local MongoDB server, and calls the relevant query functions found in the `queries.py` module to answer the given questions.
- The query functions for the different questions behave in a similar manner, but some includes additional data manipulation using python packages. Most commonly, the functions query the local MongoDB server with some declarative query of which the result is stored in printed to the console using the `pprint` python package.
- Questions 1 through 5, 8, 11 and 12 are answered with MongoDB queries only.
- Questions 7 and 10 make use of application side joins to speed up the queries.
- To answer question 6, the `activity` and `trackpoint` collections are downloaded and merged into a Pandas DataFrame. Next, the `scikit-learn` python package's implementation of the `DBSCAN` algorithm is used to find clusters of users that are close in time (trackpoints records within 60 seconds). Once these clusters are found, the `DBSCAN` algorithm is run again on each of the time-clusters to find users that are close in space and time (trackpoint records within 100 meters). The haversine distance is used to approximate the earth as a sphere with a radius of 6371.0008 km [1]. Lastly, the number of users is counted as the total number of users that are close in time and space. This means that some users are double counted as they are found in multiple time-space-clusters. Due to the size of the dataset, the group ran into performance issues when trying to query the database directly. The advantage of the use of the `scikit-learn`'s implementation of the `DBSCAN` algorithm is that it uses KDTrees or BallTrees to avoid having to compute the full distance matrices.
- Question 9 is answered by querying the local MongoDB server for all the `user` and `activity` collections. The result is then merged, and further manipulated in Pandas DataFrames to find the most common year-month. Next, the two users with the most recorded activities in this month are found. To ensure that the number of recorded activities is correct, it is asserted that these two users did not start and activity in the most common year-month and end it in the subsequent.
- The result of the question 10 MongoDB query is processed using the `haversine` python package to find the total number of kilometers.

Considerations, problems, and learning outcomes

The collection design implemented in this project followed what was suggested in [2], [3] and [4]. As the number of activities per user had a maximum of 2102, activity ids were modeled as embedded id references to the activity collection. In this assignment, we were asked to not include those activities that consisted of more than 2500 trackpoints. This indicates that a similar collection design can be used for activities, using embedded references to trackpoint as a field in the activity collection. However, as the group wanted the database to allow for the removal of the imposed restriction, it found it more suitable to model the trackpoint collection with a reference to the activity collection and omitting the reference field from the activity collection. With the exception of the altitude field, as mentioned before, the data fields were modeled with the same types as suggestion in the assignment text.

As from assignment 2, due to the way the assignment text explained how to match labels to trackpoints, on exact matches on start time and end time, there was a significant number of activities that ended up without a transportation mode. As mentioned above, the implementation had to be adjusted for users that had different labels for the same activity.

The assignment did not pose too many problems as a decent amount of work could be reused from assignment 2. For example, code to solve question 6 could easily be adjusted to work with MongoDB. Pain points were mostly related to spending a lot of time on learning a new query syntax, and getting to know the MongoDB database.

The group found this assignment interesting as it presented an opportunity to learn a new database structure, and a new database query syntax. It was fascinating to see how much faster insertion of data was for MongoDB than for the MySQL database. Further, learning about the MongoDB aggregation pipelines proved to be rewarding as it provided a sequential window into each operation needed to answer the questions. As before, working with such a large dataset was interesting as it not often encountered in an academic setting, but is otherwise very relevant to a real-life work setting.

MySQL vs. MongoDB

Relational databases have a data structure based on tables, where it is necessary to define a schema. NoSQL databases are based on document structures with key-value pairs and do not require any predefined schema. This allows NoSQL databases to be more flexible and handle changing data sets in better way than relational databases.

While one-to-many relations are modeled in relation databases by having a foreign key in the many table that references the primary key in the one table, NoSQL databases offer a multitude of different way to model such relationships depending on the cardinality of the different tables. Expensive join operations can be mitigated in the NoSQL databases by chosen an efficient way of modeling such relationship.

For this assignment, working with MySQL provided a familiar environment as the group had experience from working with relational databases from before. However, several features of MongoDB resulted in the group preferring working with MongoDB over MySQL, even though the user interface initially was less intuitive. First, while insertion of the data into the MySQL database took about 7 minutes, this was reduced to about 90 seconds when using MongoDB. Further, MySQL requires that a predefined schema is defined for the data. As explained above, this can lead to the false assumptions about the data, e.g. the type of altitudes. As NoSQL does not impose such a requirement, fewer assumptions has to be made about the data before insertion. While expensive joins were quite fast when working with MySQL, such joins could be mitigated when working with MongoDB by utilizing the application side to query the collections twice. This was utilized in question 7 and 10 in assignment 3.

In conclusion, even though MySQL offered a more familiar and user friendly interface, in terms of performance the group ended up favoring MongoDB.

References

[1] "Earth radius - Wikipedia." https://en.wikipedia.org/wiki/Earth_radius#Arithmetic_mean_radius.

[2] "6 Rules of Thumb for MongoDB Schema Design: Part 1 | MongoDB." https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-1.

[3] "6 Rules of Thumb for MongoDB Schema Design: Part 2 | MongoDB." https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-2.

[4] "6 Rules of Thumb for MongoDB Schema Design: Part 3 | MongoDB." https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-3.

Appendix

Collections

```
{
  _id: '000',
  has_labels: false,
  activity_id: [
  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
  12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
  24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
  36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
  48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
  60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
  72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83,
  84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,
  96, 97, 98, 99,
  ... 55 more items
  ]
}
```

Listing 1: User collection sample

Field	Type
_id	string
has_labels	boolean
activity_id	array of integers

Table 1: Type description of the user collection.

```
{
  _id: 1235,
  start_date_time: ISODate("2008-12-01T00:34:31.000Z"),
  end_date_time: ISODate("2008-12-01T00:53:34.000Z"),
  transportation_mode: 'taxi'
}
```

Listing 2: Activity collection sample

Field	Type
_id	int
start_date_time	ISODate
end_date_time	ISODate
transportation_mode	string

Table 2: Type description of the activity collection.

```
{
  _id: 1028931,
  lat: 39.9811083,
  lon: 116.2990083,
  altitude: 85.3,
  date_days: 39771.3097106481,
  date_time: ISODate("2008-11-19T07:25:59.000Z"),
  activity_id: 1244
}
```

Listing 3: TrackPoint collection sample

Field	Type
_id	int
lat	double
lon	double
altitude	double
date_days	double
date_time	ISODate
activity_id	int

Table 3: Type description of the trackpoint collection.

Question 1:

Users	Activities	TrackPoints
182	16049	9.682e+06

Table 4: Number of users, activities, and trackpoints in the database after insertion.

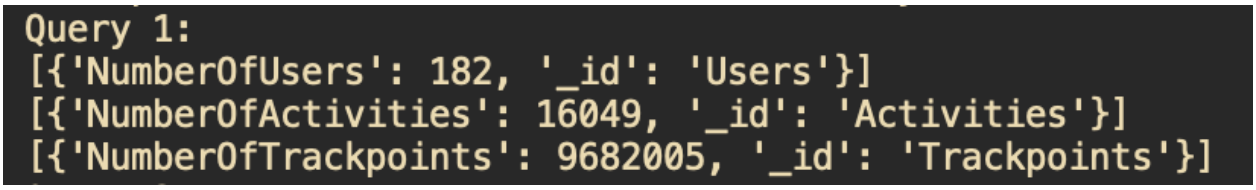

Figure 3: Console printout

Question 2:

Average	Minimum	Maximum
88.1813	0	2102

Table 5: Average, minimum, and maximum number of activities per user.

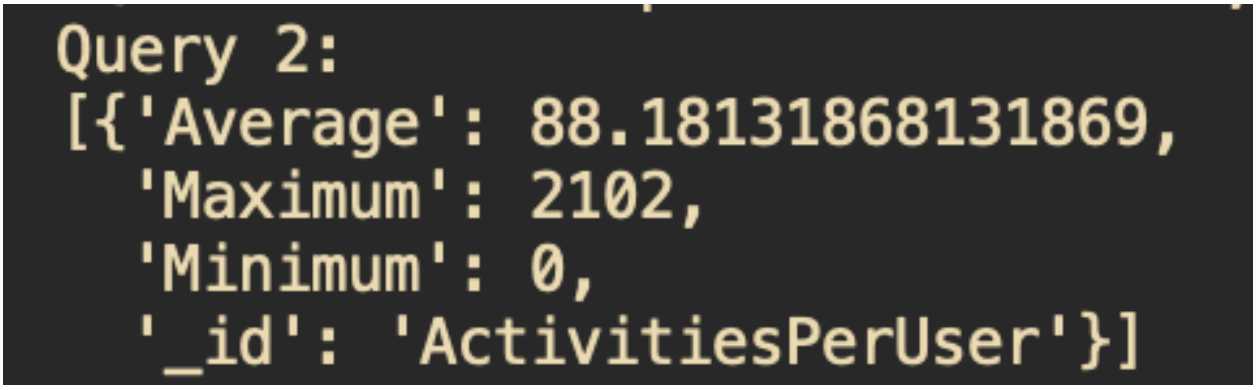

Figure 4: Console printout

Question 3:

user_id	Number of Activities
128	2102
153	1793
025	715
163	704
062	691
144	563
041	399
085	364
004	346
140	345

Table 6: The top 10 users with the highest number of activities.

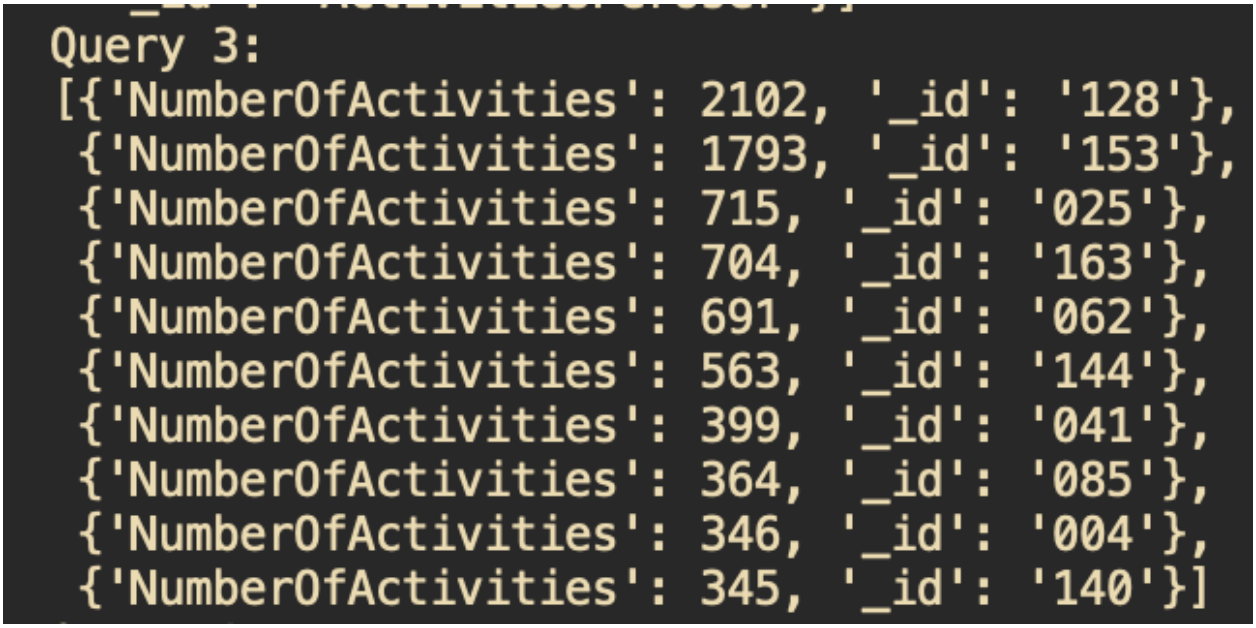

Figure 5: Console printout

Question 4

Number of Users
98

Table 7: Number of users that have started an activity during one day and ended the activity during the next day.

Figure 6: Console printout

Question 5:

user_id	start_date_time	end_date_time	count	activity_ids
020	2011-11-02 02:04:29	2011-11-02 02:09:31	2	[2367, 2366]

Table 8: Duplicate activities.

```
Query 5:
[{'_id': {'end_date_time': datetime.datetime(2011, 11, 2, 2, 9, 31),
          'start_date_time': datetime.datetime(2011, 11, 2, 2, 4, 29),
          'userId': ['020']},
  'activityIds': [2367, 2366],
  'count': 2}]
```

Figure 7: Console printout

Question 6:

```
Number of close users: 776
```

Figure 8: Console printout

Question 7:

user_id	user_id	user_id	user_id
000	044	092	138
001	045	093	139
002	046	094	140
003	047	095	141
004	048	096	142
005	049	097	143
006	050	099	144
007	051	100	145
008	052	101	146
009	053	102	147
011	054	103	148
012	055	104	149
013	056	105	150
014	057	106	151
015	059	107	152
016	060	108	153
017	061	109	154
018	063	110	155
019	064	112	156
020	065	113	157
021	066	114	158
022	067	115	159
023	068	116	160
024	069	117	161
025	070	118	162
026	071	119	164
027	072	120	165
028	073	121	166
029	074	122	167
030	075	123	168
031	076	124	169
032	077	125	170
033	079	126	171
034	081	127	172
035	082	129	173
036	083	130	174
037	084	131	175
038	086	132	176
039	087	133	177
040	088	134	178
041	089	135	179
042	090	136	180
043	091	137	181

Table 9: All users that have never taken a taxi.

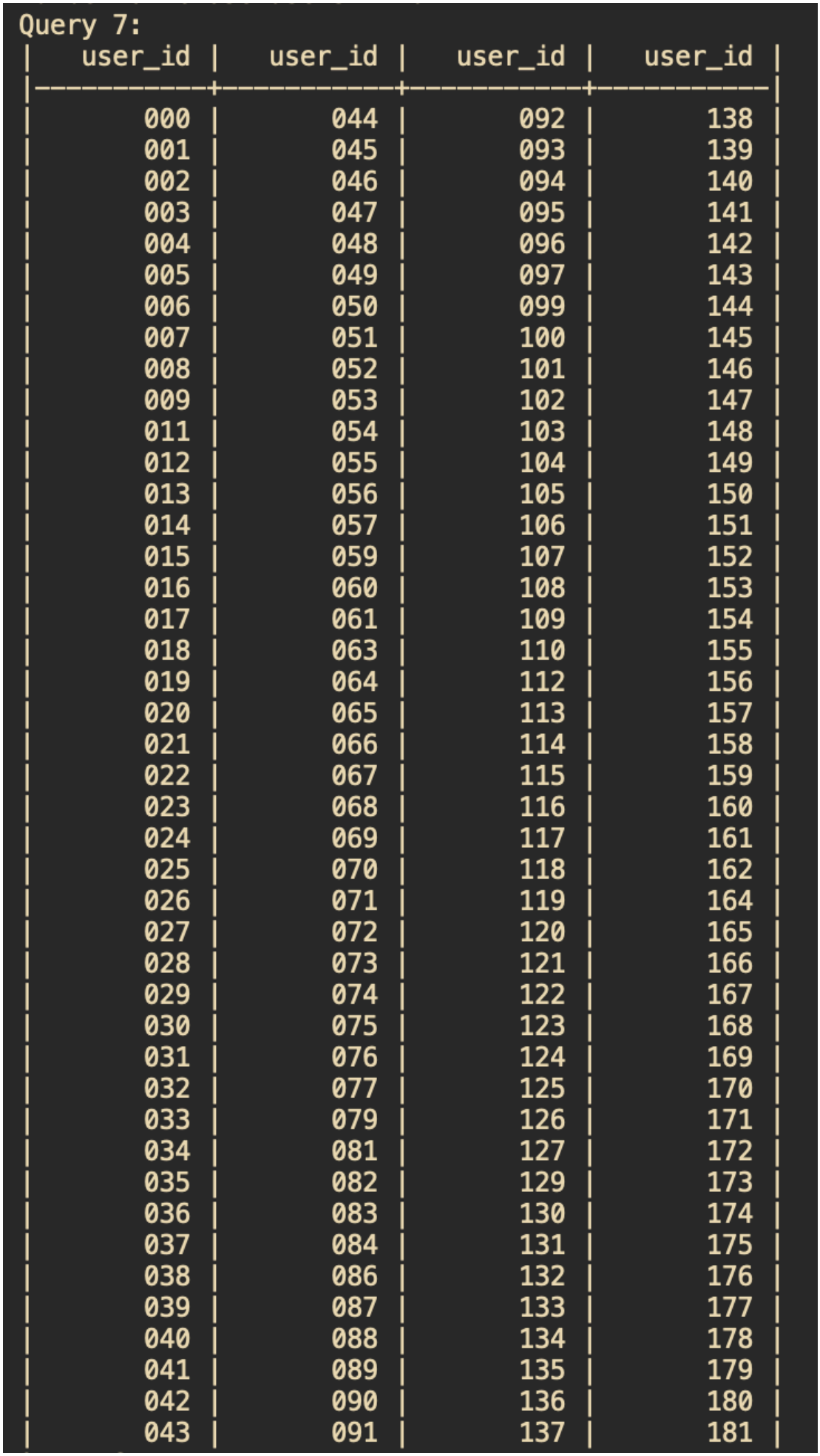

Figure 9: Console printout

Question 8:

transportation_mode	Number of Distinct Users
walk	34
bike	19
bus	13
taxi	10
car	8
subway	4
train	2
airplane	1
boat	1
run	1

Table 10: Number of distinct users that have used the different transportation modes.

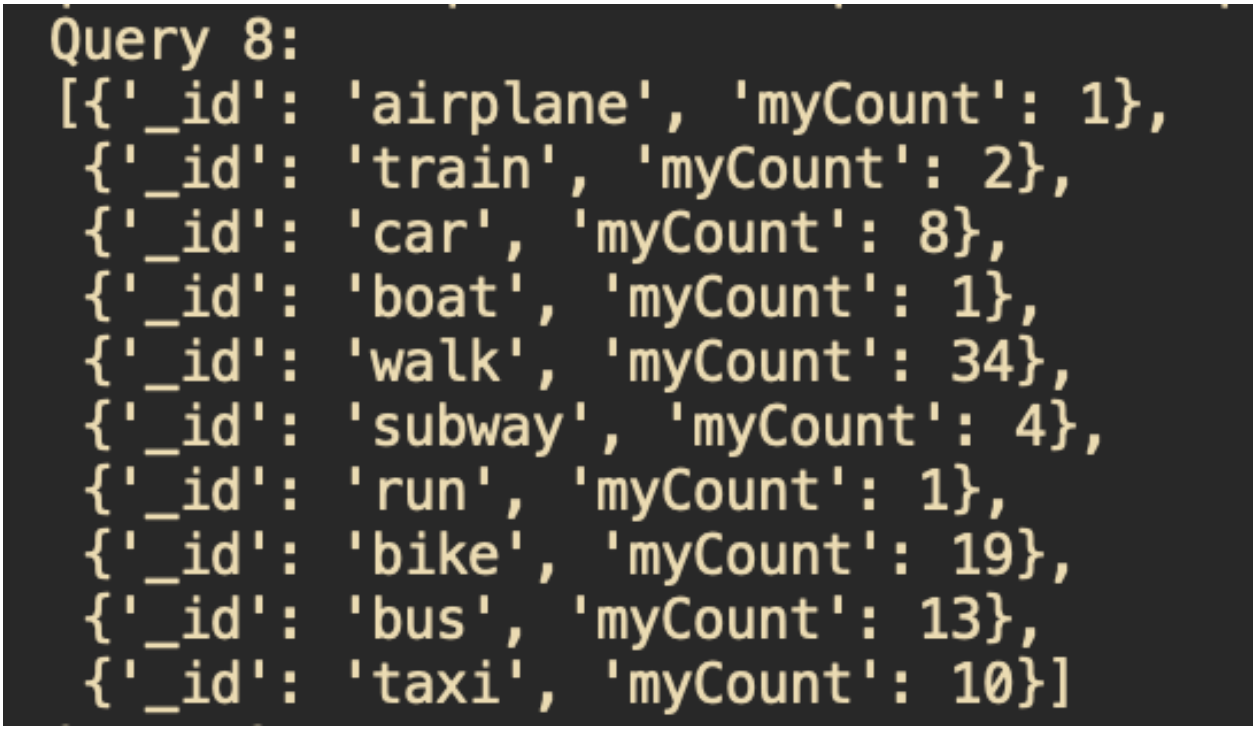

Figure 10: Console printout

Question 9:

user_id	recorded_hours	number_of_activities	year_month
062	47.3136	130	2008-November
128	68.2211	75	2008-November

Table 11: The top two users with the most number of recorded activities for the year-month with the most number of recorded activities.

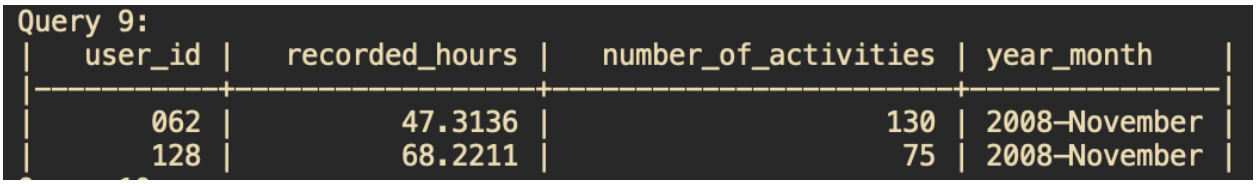

Figure 11: Console printout

Question 10:

```
Total distance walked: 115.475 km
```

```
Query 10:
Total distance walked: 115.47465961508007
```

Figure 12: Console printout

Question 11:

user_id	total_elevation_gain (m)
128	650952
153	554961
004	332036
041	240769
003	233664
085	217643
163	205274
062	181693
144	179442
030	175680
039	146704
084	131161
000	121505
002	115063
167	112974
025	109159
037	99234.6
140	94846.3
126	83025.8
017	62581.4

Table 12: The top 20 users with the most amount of altitude meters gained.

```
Query 11:
[{'_id': ['128'], 'altitudeGained': 650951.99728},
 {'_id': ['153'], 'altitudeGained': 554960.6230530001},
 {'_id': ['004'], 'altitudeGained': 332036.3184},
 {'_id': ['041'], 'altitudeGained': 240768.86568000002},
 {'_id': ['003'], 'altitudeGained': 233663.6424},
 {'_id': ['085'], 'altitudeGained': 217643.38488},
 {'_id': ['163'], 'altitudeGained': 205274.370464},
 {'_id': ['062'], 'altitudeGained': 181693.29168000002},
 {'_id': ['144'], 'altitudeGained': 179441.52448},
 {'_id': ['030'], 'altitudeGained': 175679.7096},
 {'_id': ['039'], 'altitudeGained': 146703.5928},
 {'_id': ['084'], 'altitudeGained': 131161.2312},
 {'_id': ['000'], 'altitudeGained': 121504.86240000001},
 {'_id': ['002'], 'altitudeGained': 115062.91440000001},
 {'_id': ['167'], 'altitudeGained': 112974.15464000001},
 {'_id': ['025'], 'altitudeGained': 109158.57264},
 {'_id': ['037'], 'altitudeGained': 99234.58944},
 {'_id': ['140'], 'altitudeGained': 94846.29935999999},
 {'_id': ['126'], 'altitudeGained': 83025.83576},
 {'_id': ['017'], 'altitudeGained': 62581.35312000001}]
```

Figure 13: Console printout

Question 12:

user_id	number_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
084	99
002	98
104	97
034	88
140	86
112	66
091	63
115	58
038	58
022	55
174	54
042	54
142	52
010	50
015	46
101	46
001	45
052	44
005	44
012	43
089	40
051	36
028	36
096	35
036	34
067	33
011	32
134	31
044	31
019	31
009	31
155	30
007	30
147	30
013	29
179	28
018	27

user_id	number_of_invalid_activities
024	27
071	27
082	27
111	26
125	25
065	25
029	25
103	24
035	23
119	22
043	21
020	20
016	20
078	19
074	19
168	19
026	18
110	17
040	17
073	17
006	17
150	16
008	16
081	16
094	16
057	16
055	15
083	15
181	14
154	14
097	14
102	13
058	13
046	13
139	12
032	12
061	12
099	11
088	11
023	11
138	10
131	10
172	9
162	9
105	9
157	9
158	9
169	9
063	8
130	8
176	8
050	8
076	8
053	7
021	7
146	7
161	7
056	7

user_id	number_of_invalid_activities
064	7
045	7
136	6
047	6
129	6
122	6
080	6
075	6
066	6
164	6
069	6
173	5
108	5
135	5
145	5
098	5
086	5
070	5
159	5
059	5
121	4
124	4
133	4
093	4
095	4
175	4
127	4
123	3
031	3
090	3
077	3
132	3
114	3
171	3
109	3
117	3
087	3
100	3
118	3
106	3
180	2
072	2
166	2
152	2
079	2
027	2
033	2
170	2
165	2
054	2
151	1
113	1
048	1
141	1
060	1
107	1

Table 13: The number of invalid activities per user.

```
Query 12:
[{'_id': ['128'], 'numInvalidActivities': 719},
 {'_id': ['153'], 'numInvalidActivities': 556},
 {'_id': ['025'], 'numInvalidActivities': 263},
 {'_id': ['062'], 'numInvalidActivities': 248},
 {'_id': ['163'], 'numInvalidActivities': 232},
 {'_id': ['004'], 'numInvalidActivities': 219},
 {'_id': ['041'], 'numInvalidActivities': 201},
 {'_id': ['085'], 'numInvalidActivities': 182},
 {'_id': ['003'], 'numInvalidActivities': 179},
 {'_id': ['144'], 'numInvalidActivities': 157},
 {'_id': ['039'], 'numInvalidActivities': 147},
 {'_id': ['068'], 'numInvalidActivities': 139},
 {'_id': ['167'], 'numInvalidActivities': 134},
 {'_id': ['017'], 'numInvalidActivities': 129},
 {'_id': ['014'], 'numInvalidActivities': 118},
 {'_id': ['030'], 'numInvalidActivities': 112},
 {'_id': ['126'], 'numInvalidActivities': 104},
 {'_id': ['092'], 'numInvalidActivities': 101},
 {'_id': ['000'], 'numInvalidActivities': 101},
 {'_id': ['037'], 'numInvalidActivities': 100},
 {'_id': ['084'], 'numInvalidActivities': 99},
 {'_id': ['002'], 'numInvalidActivities': 98},
 {'_id': ['104'], 'numInvalidActivities': 97},
 {'_id': ['034'], 'numInvalidActivities': 88},
 {'_id': ['140'], 'numInvalidActivities': 86},
 {'_id': ['112'], 'numInvalidActivities': 66},
 {'_id': ['091'], 'numInvalidActivities': 63},
 {'_id': ['115'], 'numInvalidActivities': 58},
 {'_id': ['038'], 'numInvalidActivities': 58},
 {'_id': ['022'], 'numInvalidActivities': 55},
 {'_id': ['174'], 'numInvalidActivities': 54},
 {'_id': ['042'], 'numInvalidActivities': 54},
 {'_id': ['142'], 'numInvalidActivities': 52},
 {'_id': ['010'], 'numInvalidActivities': 50},
 {'_id': ['101'], 'numInvalidActivities': 46},
 {'_id': ['015'], 'numInvalidActivities': 46},
 {'_id': ['001'], 'numInvalidActivities': 45},
 {'_id': ['005'], 'numInvalidActivities': 44},
 {'_id': ['052'], 'numInvalidActivities': 44},
 {'_id': ['012'], 'numInvalidActivities': 43},
 {'_id': ['089'], 'numInvalidActivities': 40},
 {'_id': ['028'], 'numInvalidActivities': 36},
 {'_id': ['051'], 'numInvalidActivities': 36},
 {'_id': ['096'], 'numInvalidActivities': 35},
 {'_id': ['036'], 'numInvalidActivities': 34},
 {'_id': ['067'], 'numInvalidActivities': 33},
 {'_id': ['011'], 'numInvalidActivities': 32},
 {'_id': ['044'], 'numInvalidActivities': 31},
 {'_id': ['019'], 'numInvalidActivities': 31},
 {'_id': ['009'], 'numInvalidActivities': 31},
 {'_id': ['134'], 'numInvalidActivities': 31},
 {'_id': ['007'], 'numInvalidActivities': 30},
 {'_id': ['147'], 'numInvalidActivities': 30},
 {'_id': ['155'], 'numInvalidActivities': 30},
 {'_id': ['013'], 'numInvalidActivities': 29},
 {'_id': ['179'], 'numInvalidActivities': 28},
 {'_id': ['024'], 'numInvalidActivities': 27},
 {'_id': ['071'], 'numInvalidActivities': 27},
 {'_id': ['082'], 'numInvalidActivities': 27},
 {'_id': ['018'], 'numInvalidActivities': 27},
 {'_id': ['111'], 'numInvalidActivities': 26},
 {'_id': ['065'], 'numInvalidActivities': 25},
 {'_id': ['029'], 'numInvalidActivities': 25},
 {'_id': ['125'], 'numInvalidActivities': 25},
 {'_id': ['103'], 'numInvalidActivities': 24},
 {'_id': ['035'], 'numInvalidActivities': 23},
 {'_id': ['119'], 'numInvalidActivities': 22},
 {'_id': ['043'], 'numInvalidActivities': 21},
 {'_id': ['016'], 'numInvalidActivities': 20},
 {'_id': ['020'], 'numInvalidActivities': 20},
```

Figure 14: Console printout

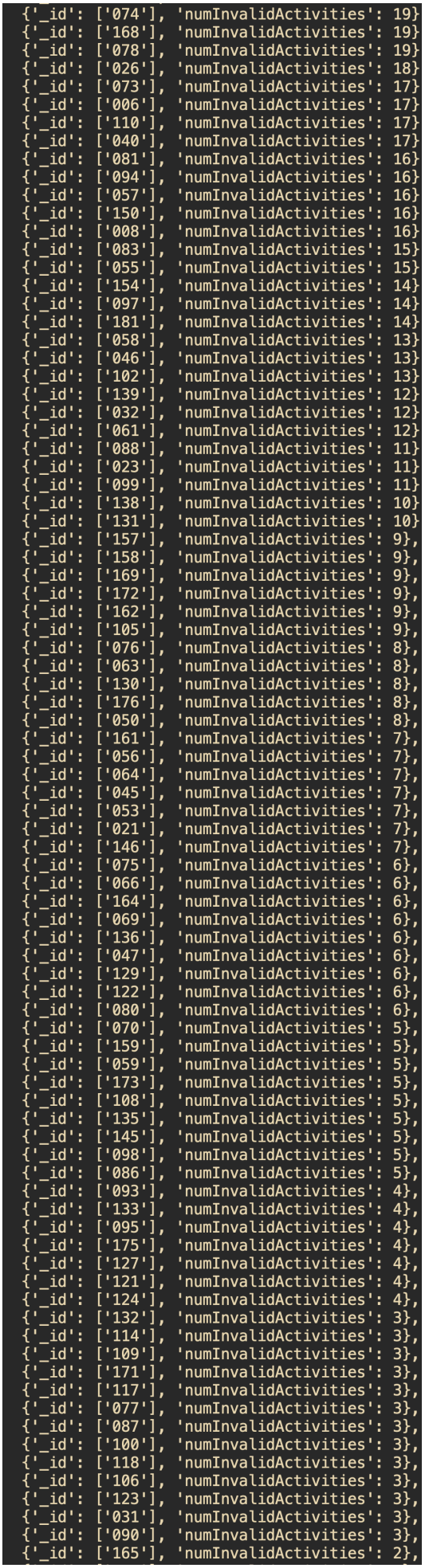

Figure 15: Console printout

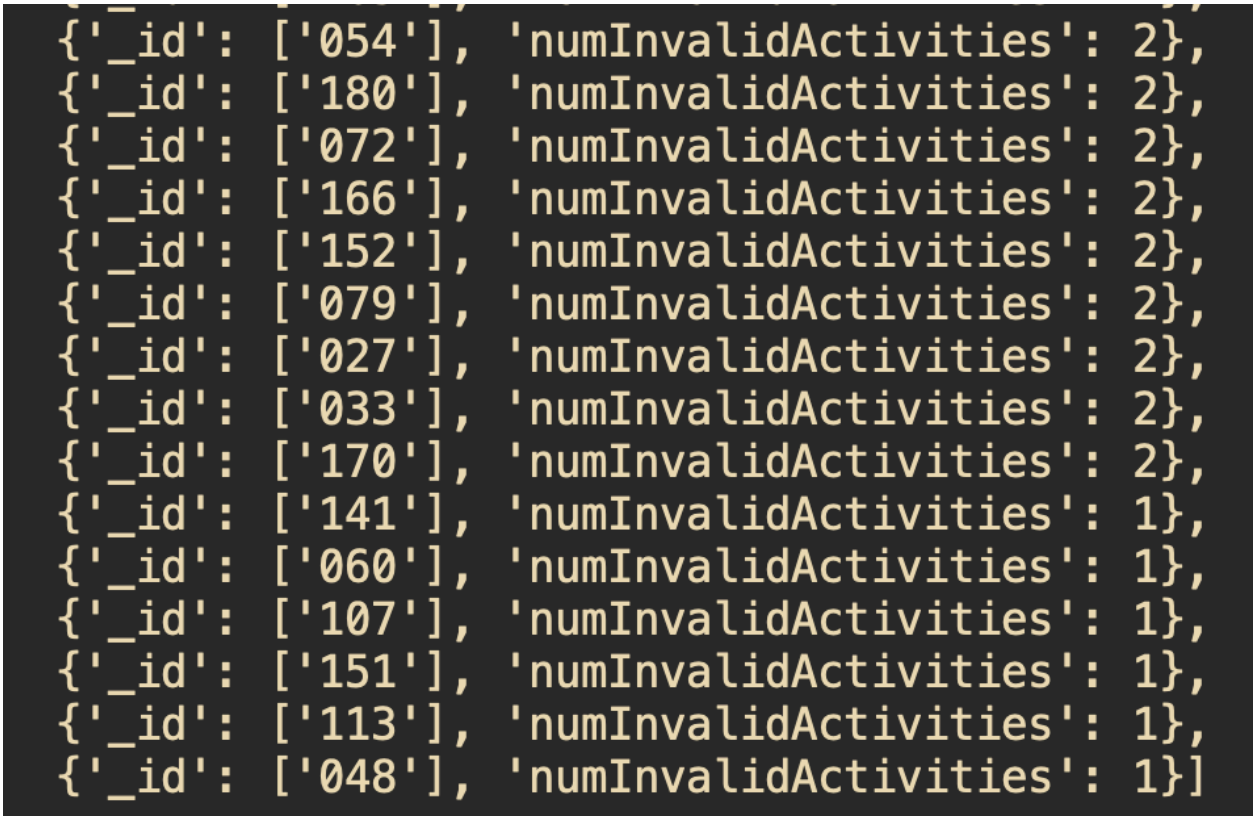

Figure 16: Console printout