User Guide

Before running the code, it is important to set up a MongoDB server on the lab machine. To set up, enter the home directory of your repository and type the following into the terminal:

```
mongod --port 27017 --dbpath ./w24-mp2-dhavaldb &
```

(if necessary, pip3 install pymongo and use python3)

The output of the MongoDB server starting up will be displayed, simply press ‘Enter’ to exit the display loop. Next, enter the following:

```
mongod --port 27017
```

(Note: We are using port **27017**, however other ports can be used as well)

After which, you’ll be directed into MongoDB on the terminal. Since we are using pymongo, we can exit MongoDB by entering ‘exit’ in the terminal:

```
test> exit
```

To run the code, we would first have to read the data and create the collections. After the collections are created, we can run the queries. It’s important to also have the port number in the command line as an argument. Type the following commands to run the program on port **27017**, other ports can be used as well:

```
python3 task1_build.py 27017
python3 task1_query.py 27017
```

The first line creates the collections and the second line runs the queries. The same can be done for Task 2:

```
python3 task2_build.py 27017
python3 task2_query.py 27017
```

Loading JSON Files to MongoDB

Task 1)
Since messages.json and senders.json differ greatly in size, we utilize two different methods in handling. For messages.json, our program reads the JSON file line by line and appends each line into a separate batch. After the batch size reaches 5000, the batch is inserted into the collection and then reset. This method is more memory efficient as we don’t load the entire file

into memory all at once. Our utilization of insert_many() allows us to be more efficient than inserting individual arrays one by one. For senders.json, since it is a smaller file, we are able to read the file using json.load(), without needing to read line by line. Similar to messages.json, we insert the data into the collection in batches and then use insert_many() to insert the entire batch into the collection.

Task 2)
Similar to Task 1. Since we have two JSON files that differ greatly in size, we utilize two different methods. For messages.json, our program reads the JSON file line by line and appends each new line into a separate batch. Once the batch size reaches 5000, the existing batch is inserted into the collection. Like in task 1, we utilize insert_many(), which allows us to be more efficient in the insertion of our documents. For senders.json, similar to task 1, since it is a smaller file, we are able to insert the entire file into memory all at once using json.load(). Utilization of batches and insert_many() allows senders.json to read the entire JSON file and insert into the collection all at once, while being more memory efficient.

Unlike in Task 1, we are required to embed 'sender_info' into the messages collection, we must parse through sender_list, which is created when we read the senders.json file. In order to make our program more efficient in time complexity, we sort the sender_list in alphabetical order based on the sender_id. We then utilize a binary-search type algorithm to find and embed sender_info into a new attribute named 'sender_info' within each entry in messages.

Runtime Analysis

Task 1:
Step 1) Time taken to read messages.json: **14.49 seconds**
Time taken to created "messages" collection: **14.55 seconds**

Step 2) Time taken to read senders.json: **0.0095 seconds**
Time taken to create "senders" collection: **0.1612 seconds**

Step 3) Time taken to run Q1: **4.41 seconds**
Time taken to run Q2: **3.29 seconds**
Time taken to run Q3: **over 2 minutes (timeout exception)**
Time taken to run Q4: **0.096 seconds**

Step 4) Time taken to run Q1 after creating indices: **4.40 seconds**
Time taken to run Q2 after creating indices: **2.21 seconds**
Time taken to run Q3 after creating indices: **55.55 seconds**

When analyzing the runtimes for each query, it is clear that indexing significantly improves the runtime by making the queries run faster. For Q1, initially MongoDB has to search through every

document in the collection and match the query requirements. This is not very efficient, especially as the collection grows in size. After indexing, the search is constrained to only the 'text' field, which speeds up the process. Similarly, for Q2, MongoDB initially has to do a collection scan, which involves sorting and grouping the messages by sender. Once again, indexing targets a specific field, which makes the search performance much more efficient. Regarding Q3, MongoDB has to initially lookup and match the senders' whose credit is equal to 0. By indexing the 'sender' field in the 'messages' collection, the senders can be filtered based on their credit values, optimizing performance. Lastly, in Q4, MongoDB has to scan through the collection, find the documents, and update them. With the use of indexing, this process is executed much faster as the documents can be located and updated in less time. Overall, indexing is crucial for enhancing query performance with MongoDB. By rapidly retrieving documents and modifying them, long collection scans are not required. Indexing also decreases execution time for queries that involve sorting, filtering and grouping actions.

Task 2:

Step 1) Time taken to read messages.json: **21.48 seconds**
Time taken to created "messages" collection: **21.58 seconds**
Time taken to read senders.json: **0.0155 seconds**

Step 2) Time taken to run Q1: **5.65 seconds (5647.95 milliseconds)**
Time taken to run Q2: **0.71 seconds (706.60 milliseconds)**
Time taken to run Q3: **0.53 seconds (526.34 milliseconds)**
Time taken to run Q4: **0.02 seconds (22.94 milliseconds)**

Time taken to run Q1 after creating indices: **1.25 seconds (1247.29 milliseconds)**
Time taken to run Q2 after creating indices: **0.66 seconds (661.97 milliseconds)**
Time taken to run Q3 after creating indices: **0.52 seconds (520.53 milliseconds)**

In Q1, we need to return the number of messages that have "ant" in the text and using the embedded model does not change the query as it does not depend on the attribute of sender_info. Both models are good because the query does not depend on sender_info so the performance is the same. Both models are just as good. In Q2, we need to find the nickname/ phone number of the person who sent the most messages, we need to adjust the groupings in order to consider the nested sender id. The embedded model is a better choice for the query as it allows us to group more efficiently and count the number of messages faster. In Q3, we need to return the number of messages where the sender's credit is 0 and the query remains the same as it targets the sender_info.credit. The embedded model is a better choice as it allows us to directly check the nested attribute sender_info.credit. This allows us to run way faster than the normalized one. The number of messages was 15354 initially, I did not modify anything and now it seems to not be providing the correct output as it gives 0. In Q4, we need to double the credits of senders less than 100, we need to change the update to look at the sender_info.credit instead of credit. Depending on if the data is constantly being updated, the embedded model

would be more efficient as the data would be directly embedded in the sender_info.credit. Next we will look at the queries when indexing. Q1 will have better performance as it allows you to use the ‘text’ field in order to find all of the fields containing “ant” faster. Q2 will also be faster as the ‘sender’ field will allow us to group messages more efficiently in the query. Q3 does not improve by performance as we use the sender_info.credit field.

Query Outputs

Task 1
Q1) Number of messages containing ‘ant’: **19551**

Q2) Sender with the most messages: ***S.CC
Number of messages sent by ***S.CC: **98613**

Q3) Number of messages with sender’s credit as 0: **15354**

Q4) Number of senders with credit less than 100 updated: **970**

Task 2
Q1) Number of messages containing ‘ant’: **19551**

Q2) Sender with the most messages: ***S.CC
Number of messages sent by ***S.CC: **98613**

Q3) Number of messages with sender’s credit as 0: **0**

Q4) Number of senders with credit less than 100 updated: **970**