Fehintola Disu, Elykah Tejol and Sandhya Adhikari.

CMPUT 291
Mini Project 2 Report

(A) General overview of the system

This mini project works with two json files: messages.json and senders.json.

- task1.py builds a normalized document store for the database, as well as indexing queries 1 to 3.
- task2.py builds an embedded document store for the database, having a new field named "sender_info" with the information retrieved from messages.json

User guide

Prerequisites before running the code:

- Must have messages.json and senders.json in the same directory as the project files (task1_build.py, task1_query.py, task2_build.py, task2_query.py, task1.py, task2.py) .
- Must know the port number to a MongoDB server, and this must be running.

Execution of code

For task1:

Run the python file named task1.py
It will ask for the post number, please provide your MongoDB server's port number
Then task1_build.py and task1_query.py will be called inside this file.
The required database will be created and the queries will run and the required output will be displayed.

For task2:

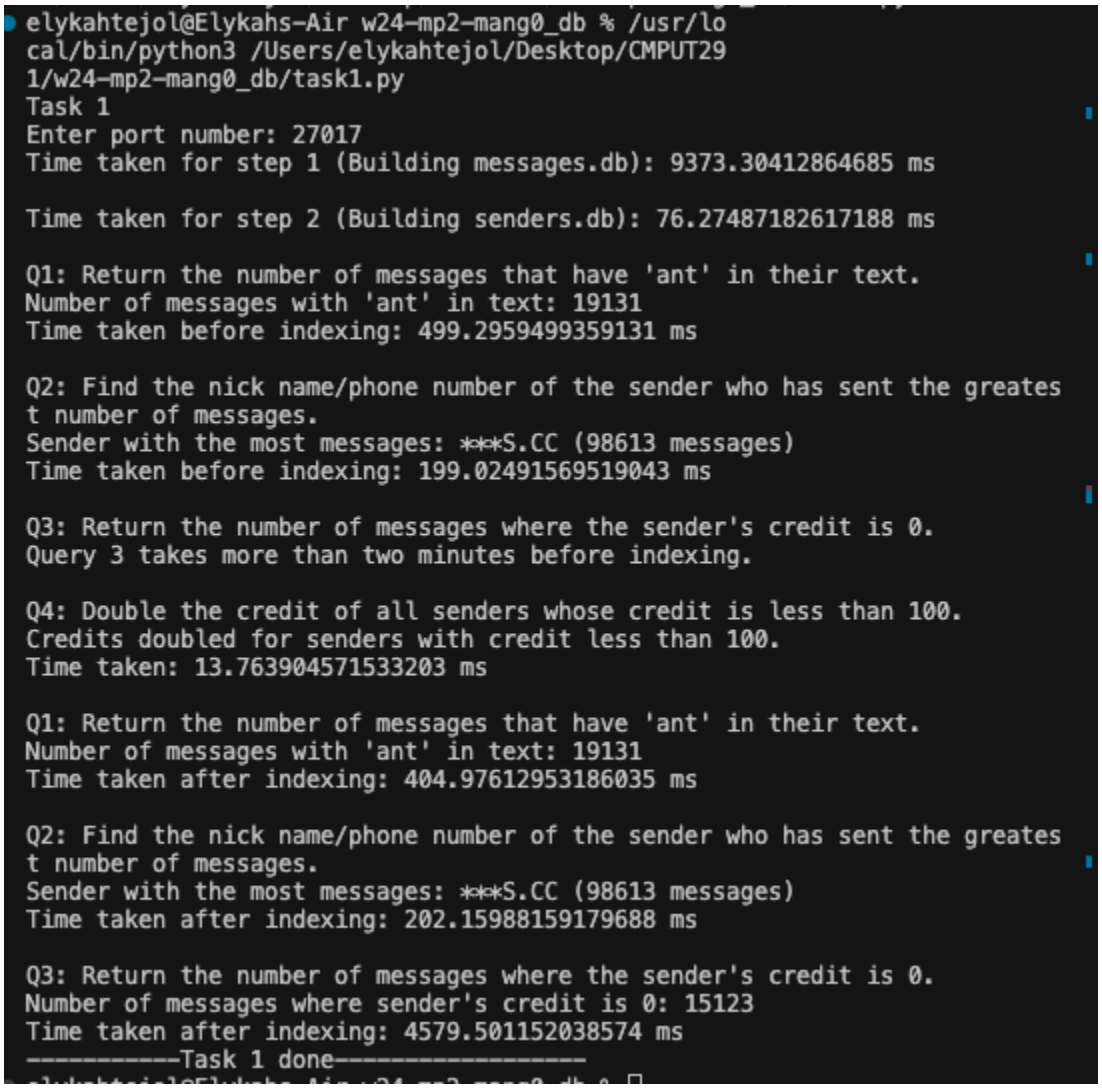
Run the python file named task2.py
It will ask for the post number, please provide your MongoDB server's port number
Then task2_build.py and task2_query.py will be called inside this file.
The required database will be created and the queries will run and the required output will be displayed.

(B) Strategy to load large json files into mongodb

- To handle the potentially massive ‘messages.json’ file, the code reads it line by line, avoiding loading the entire file into memory at once. It skips lines containing ‘[‘ or ‘]’ characters and processes the remaining lines by stripping trailing commas and newlines, loading the JSON object, and appending it to a list. The list is then inserted into the MongoDB collection in batches of 5000 documents, using the ‘insert_many’ operation to overcome memory limitations and improve performance.

(C) Output for each query and analysis

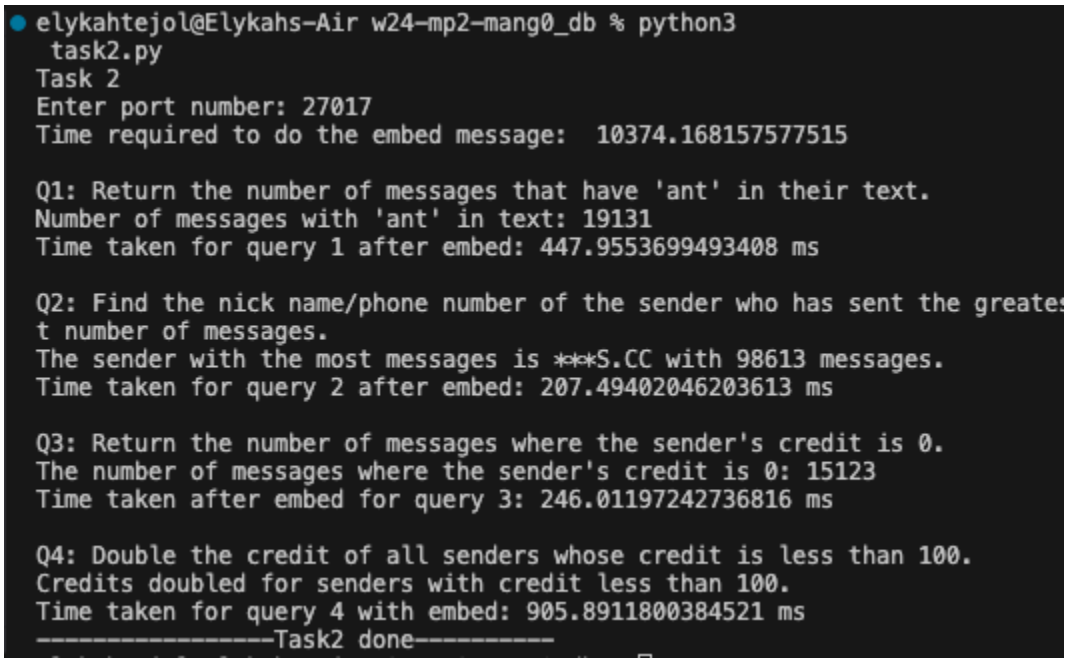
Output for Task 1 queries, with indexing state specified:



Question from Task 1: *Repeat query 1, 2, and 3. How does the run time change? In your report, explain how and why indexing affects your runtime for each query.*

The runtime decreases, and this is due to the indexing. Indexing allows a more efficient query, as instead of scanning every single document and seeing for matches. Indexing lowers the amount of documents to scan, allowing for a faster run time.

Output for task2 queries:



Question:

*In your report, do a comprehensive analysis of your run times and **for each query** explain:*

1) Is the performance different for normalized and embedded model? Why?

2) Which model is a better choice for the query and why?

1. Is the performance different for normalized and embedded model? Why?

    - The difference in performance for normalized and embedded model depends on what the query wants. For queries 1 and 2, the performance is very similar for both, but it does prove to be faster with the normalized model with indexing. In that case, indexing allows the query to be faster, and this is due to the nature of indexing, it reduces the amount of documents to scan.
    - For query 3, the performance is significantly better with an embedded model. The reason is because there is no need to access two collections anymore, as the information we need from senders is now all in one embedded document.
    - Query 4, however, proves to have a significantly higher performance with the normalized model. We believe this is due to the nesting in the embedded document, as we would have to access document > sender_info field > search for the credit and update accordingly. For the normalized model, there's no nesting. So, accessing the document would suffice for searching and updating the relevant information in a short amount of time.

2. Which model is a better choice for the query? Why?

    - For query 1 and query 2, both models had similar run times. However, indexing proved to reduce the run time. So, the normalized document with indexing would be the better choice due to the more efficient run time.
    - For query 3, the embedded model proved to be the better choice as this gives the advantage of only accessing one collection. So, in the case of information needed from two collections, it would be a smart choice to just create an embedded model for that as the normalized model without indexing took longer than 2 minutes that it wouldn't even run according to the requirements for this project.
    - For query 4, a normalized model would be the smart choice. On average, this was around 10ms for the normalized model. For the embedded, this would yield around 800ms, around 80x more than the normalized model.