Overview:

- Our project consists of two main components: data ingestion and data querying. The data ingestion component reads JSON-formatted SMS data and sender information, normalizes, and embeds this data as necessary, and stores it in MongoDB collections. The data querying component retrieves and manipulates this data based on specific analytical questions, showcasing the impact of indexing on query performance.

User guide:

System Requirements:

- MongoDB server
- Python 3.x+
- Pymongo Library

Running the script:

1. Ensure Python 3 is installed with the pymongo library. If not, install using pip install pymongo.
2. Navigate to the directory containing the scripts
3. To run the query and see the impact of indexing run in the order:
    a. python3 task1_build.py <port number>
    b. python3 task1_query.py <port number>
    c. python3 task2_build.py <port number>
    d. python3 task2_query.py <port number>

Task 1 Analysis:

(a) Your strategy to load large json files into mongodb:

- Our strategy for loading large JSONs for this file was to use a double while loop. On the first while loop we make sure the line is not EOF, then create a list to hold the data and a counter. We then enter the second for loop which checks for EOF as well as how many lines are in the current batch. We iterate through the file adding each valid line to a list until EOF or batch size is reached, where then we exit the loop and add the list to the database and clear it as well as reset the counter. Then we check the conditions for the outer for loop if the line is not EOF, where we loop if there is still more.

(b) The output for each query and a complete and accurate answer to all of the questions asked in

```
qdinh@ug22:~/CMPUT291/miniproject2/w24-mp2-nuhuhsql>python3 task1_build.py 27017
creating database...
line cannot be parsed to a json: [

line cannot be parsed to a json: ]
time to create database: 12.67 seconds
qdinh@ug22:~/CMPUT291/miniproject2/w24-mp2-nuhuhsql>python3 task1_query.py 27017
Q1. Number of messages with 'ant': 19551
Time taken: 0.6975 seconds
Q2. Top sender: ***S.CC with 98613 messages
Time taken: 0.5171 seconds
Q3. Number of messages from senders with 0 credit: 15354
Time taken: 0.5447 seconds
Q4. Updated 970 senders' credit.
Time taken: 0.0130 seconds

Creating indices...
Q1. Number of messages with 'ant': 19551
Time taken: 0.6934 seconds
Q2. Top sender: ***S.CC with 98613 messages
Time taken: 0.5157 seconds
Q3. Number of messages from senders with 0 credit: 15354
Time taken: 0.0180 seconds
```

- **Query 1 Analysis:**
    i. Query 1 ran to count the number of mails that included the term 'ant'. Before indexing, Query 1 found 19,551 messages containing 'ant' in 0.6975 seconds. After indexing, the count stayed constant, and the execution time was almost comparable at 0.6934 seconds. The absence of significant changes indicates that the text index does not dramatically enhance the performance for regex-based searches.
- **Query 2 Analysis:**
    i. Query 2 involves an aggregate to determine which sender had the most messages. For Query 2, the top sender, identified by a disguised phone number for privacy reasons, sent 98,613 messages. After indexing, the procedure took 0.5157 seconds, down from 0.5171 seconds before. The minor gain in time suggests that the sender field's index marginally streamlined the grouping operation.
- **Query 3 Analysis**
    i. Query 3 was run to determine the number of messages from senders having a credit value of zero. Query 3 returned 15,354 messages from senders with 0 credit. The indexing significantly lowered the query time from 0.5447 seconds to 0.0180 seconds. This large reduction in time after indexing demonstrates the effectiveness of a proper index, in this example in the credit area. The findings clearly demonstrate how a well-placed index may speed up search and retrieval processes, allowing quick access to certain data subsets.
- **Query 4 Analysis**
    i. Query 4 focused on updating the senders collection to double the credit for those with credit scores less than 100. The script updated the credit for 970 senders in Query 4, with the operation taking 0.0130 seconds. Given that this is an update operation and that the count of affected senders would be dependent on the current state of the data (which can vary if the query is

run multiple times), the focus here is more on the correctness of the operation rather than performance improvement through indexing. The quick execution time indicates that the database effectively processes updates on fields even without index support.

Task 2 Analysis:

(a) Your strategy to load large json files into mongodb:

- When managing large JSON file uploads to MongoDB, we adopted a smooth and efficient approach. First, we use file streaming to read the data, ensuring memory efficiency by processing one line at a time. This method is important for handling big file sizes without overloading the system's memory.
- For insertion, we used batch processing, grouping multiple records into single insert operations. This technique significantly improves performance by reducing the number of write operations to the database. We choose a batch size of 5000 based on the assignment requirement. • Robust error handling is important to this process. It ensures that any bad data or parsing errors don't stop the entire operation. Instead, the script logs such incidents and continues processing, maintaining overall data integrity and upload continuity.
- We utilized Python's built-in JSON library for parsing, given its reliability and efficiency in handling JSON data. For larger-scale operations, I consider parallel processing, which can speed up the upload process by utilizing multiple processors to handle different segments of the data simultaneously.
- Throughout this process, detailed logging is crucial for monitoring progress and identifying any potential issues. This strategy ensures an efficient, reliable, and scalable solution for loading large JSON files into MongoDB.

(b) The output for each query and a complete and accurate answer to all of the questions asked in

```
qdinh@ug22:~/CMPUT291/miniproject2/w24-mp2-nuhuhsql>python3 task2_build.py 27017
Time to load and process senders: 0.0099 seconds
Skipping invalid JSON at line 1
Skipping invalid JSON at line 1174886
Time to load and insert messages: 14.5609 seconds
qdinh@ug22:~/CMPUT291/miniproject2/w24-mp2-nuhuhsql>python3 task2_query.py 27017
Executing queries before indexing:
Q1 completed in 0.7139 seconds
Q1: Number of messages containing 'ant': 19551
Q2 completed in 1.1608 seconds
Q2: Top sender: [{'_id': '***S.CC', 'count': 98613}]
Q3 completed in 0.4985 seconds
Q3: Messages from senders with 0 credit: 15354
Q4 completed in 1.7706 seconds
Q4: Doubled the credit of all senders whose credit is less than 100!

Creating indices...

Executing queries after indexing:
Q1 completed in 0.6981 seconds
Q1: Number of messages containing 'ant': 19551
Q2 completed in 1.2444 seconds
Q2: Top sender: [{'_id': '***S.CC', 'count': 98613}]
Q3 completed in 0.0080 seconds
Q3: Messages from senders with 0 credit: 15354
```

- **Query 1 Analysis:**

    i. The execution of Query 1 both before and after indexing clocked in at approximately 0.71 seconds, consistently returning 19,551 messages containing 'ant'. This uniformity in performance indicates that the text index has minimal impact on regex searches within the 'text' field, likely because regex patterns do not fully exploit the text index's tokenization capabilities.

- **Query 2 Analysis:**
    i. Query 2 took about 1.16 seconds to ascertain the top sender before indexing and slightly longer at 1.24 seconds after indexing. This slight increase in execution time post-indexing suggests that the indexing might introduce some overhead or that the complex nature of the aggregation operation does not gain significant benefit from indexing.
- **Query 3 Analysis:**
    i. Initially, Query 3 required 0.49 seconds to find messages from senders with zero credit, but this time reduced to just 0.08 seconds after indexing. This significant acceleration highlights the effectiveness of indexing on the 'sender_info.credit' field for direct match queries, which bypasses the need for full collection scans and directly accesses relevant documents.
- **Query 4 Analysis:**
    i. Query 4's performance, at 1.77 seconds to update sender credits, affected a large number of message entries, indicating a potential issue with the update scope. Instead of updating individual sender records, it seems to have updated multiple messages linked to each sender. Correcting the scope to target unique sender documents should yield a much lower modified count, aligning with the actual number of senders with credit below 100.

Comprehensive Analysis:

Here in task 2, we compare the query results from task one on the normalized model with the results from task 2, using the embedded model.

Query 1: Task 1 time: 0.6975s Task 2 time: 0.7139s

We can observe that query 1 stays relatively between task 1 and 2. Query 1 doesn't benefit or hurt from the embedded model as it still must query the same amount of content (the whole messages document) to find the answer, so the time is approximately equal.

Query 2: Task 1 time: 0.5171s Task 2 time: 1.1608s

For Query 2, the normalized model of Task 1 (0.5171s) outperformed the embedded model of Task 2 (1.1608s) in aggregating and identifying the top sender. The indexed search in Task 1 allowed for faster retrieval, suggesting that when aggregation and grouping by a field are required, the normalized model is more efficient. This is likely due to the direct and indexed access to the sender's information without the overhead of navigating through embedded

documents. So, the normalized model is better for Query 2 because it allows for quicker and more efficient aggregation operations, which benefit from having a separate index for the sender field.

Query 3: Task 1 time: 0.5447s Task 2 time: 0.4985s

For query 3, task 2 is slightly faster than task 1 unindexed, and much faster when indexed. This is because on the embedded model, we only need to use one collection, allowing us to check the credit in a single operation. For this query we should use the embedded model. For the normalized model we must first check the message and then the sender for the credits.

Query 4: Task 1 time: 0.0130s Task 2 time: 1.7706s

For query 4 the answer is significantly slower now (over 100 times slower) in the embedded model, as to double the sender's credit, it must iterate through the entire messages collection instead of the sender's collection. Because messages are so much larger than senders, and senders' information is now embedded in each message, there is a lot more that needs to be done to double the credit of those under 100. For this operation we should use the normalized model.