REPORT - CMPUT 291, MINI PROJECT 2 - Team LastMinuteSubmission

(a) USER GUIDE:

STEP 1: CHECK PREREQUISITES

- Make sure you have Python installed on your system
- Install MongoDB, https://www.mongodb.com/docs/manual/installation/
- Make sure to install the 'pymongo' library, https://pypi.org/project/pymongo/
- Make sure the files are in the same directory

STEP 2: MongoDB CONNECTION

- Set up a MongoDB connection in the same directory
- Note the down the port number

STEP 3: FILES AND THEIR USES

- task1_build.py: this file is used to fill and populate the MongoDB database with data from JSON files and creates 2 collections: 'messages' and 'senders'
- task2_build.py: this script is used to fill and populate the 'messages' collection in MongoDB with data from a JSON file. Also embeds sender information from another JSON file.
- task1_query.py & **task2_query:** these files are used to execute commands and queries. They create indices for better performance and also measures their execution time. The first file runs on normalized datastore, while the second one uses embedded datastore.

STEP 4: MAKING TASK 1 WORK

- Open Terminal on your machine
- Write python3 task1_build.py "port number" in order to build the normalized datastore
- In the above command, replace "port number" with the actual 5-digit port number
- After creating the normalized datastore, write python3 task1_query.py "port number", to display the requested information in the project's steps 3 and 4
- In the above commands, use the actual port number

STEP 5: MAKING TASK 2 WORK

- Open Terminal on your machine
- Write python3 task2_build.py "port number" in order to build the embedded datastore
- In the above command, replace "port number" with the actual 5-digit port number
- After creating the embedded datastore, write python3 task2_query.py "port number", to display the requested information in the project's steps 3 and 4
- Remember to use the actual port number instead of "port number"

(b) LOADING LARGE JSON FILES - OUR STRATEGY:

For both the tasks provided in the question, our team used similar strategies. Our main aim was to read and process large JSON files in chunks and not together. We did not try to load the entire file into memory all at once. Below is our detailed strategy for each of the tasks:

Task 1: We did not want to return the entire JSON file as a list. Instead, we used the function read_json_array(filename). It gives individual items iteratively from the JSON array using a generator. This allows for lazy loading and efficient memory usage. Also in the “main” function, we have used the “insert_in_batches()”. This makes sure that the large files are processed in batches or chunks.

Task 2: This task also uses the function “read_json_file(filename)”, however, the files are not processed iteratively like in the previous task. The “main” of this task uses the function “read_json_file()”, and uses the insert_in_batches like the previous task.

Handling large JSON files is one of the core fundamentals of this project, and both tasks use slightly different but similar methods of handling. But there are some effects on the runtime and the overall complexity, which we will talk about in the analysis below.

(c) OUTPUT FOR ALL QUERIES

Task 1 Output:

```
Executing Query 1 (before indexing):
Result: 19551, Time taken: 885.61 milliseconds
Executing Query 2 (before indexing):
Result: Sender: ***S.CC, Messages: 98613, Time taken: 1276.49 milliseconds
Executing Query 3 (before indexing):
Result: 15354, Time taken: 653.38 milliseconds
Creating indices...

After creating indices, re-executing Query 1:
Result: 19551, Time taken: 829.71 milliseconds
After creating indices, re-executing Query 2:
Result: Sender: ***S.CC, Messages: 98613, Time taken: 892.41 milliseconds
After creating indices, re-executing Query 3:
Result: 15354, Time taken: 25.93 milliseconds
Executing Query 4 to double credits for senders with credit less than 100:
Query 4 - Double credit for senders Time taken: 13.05 milliseconds
```

Task 2 Output:

```
Executing queries 1, 2, and 3 before creating indices:
Q1: Count messages containing 'ant' Result: 19551, Time taken: 834.68 milliseconds
Q2: Sender with the greatest number of messages Result: Sender: ***S.CC, Messages: 98613,
Time taken: 927.37 milliseconds
Q3: Count messages where sender's credit is 0 Result: 15354, Time taken: 653.27 milliseconds
Creating indices...
Indices creation completed.

Re-executing queries 1, 2, and 3 after creating indices to observe performance changes:
Q1: Count messages containing 'ant' Result: 19551, Time taken: 890.83 milliseconds
```

Q2: Sender with the greatest number of messages Result: Sender: ***S.CC, Messages: 98613, Time taken: 736.29 milliseconds
Q3: Count messages where the sender's credit is 0 Result: 15354, Time taken: 11.90 milliseconds
Executing query 4 to double credits for senders with credit less than 100:
Q4: Double the credit for senders with credit less than 100 Time taken: 4653.37 milliseconds

(d) Q&A + RELEVANT ANALYSIS

Task 1 Analysis:

Query 1 Analysis: A slight improvement in runtime after creating a text index on the "text" field is expected. Text indexing optimizes text search queries by creating a special index that supports efficient text search operations across string content. The reduction in time is relatively small, which might be due to the nature of the data, the efficiency of text search even without indexing, or the overhead involved in using a text index.

Query 2 Analysis: This query shows a more significant improvement in runtime after indexing. The creation of an index on the "sender" field allows MongoDB to aggregate messages by sender more efficiently. Instead of scanning every document in the collection, MongoDB utilizes the index to quickly locate and group documents by sender, significantly reducing the time required to perform the aggregation and sort operations.

Query 3 Analysis: The crazy improvement in runtime for this query is the most notable. By creating an index on the "sender_info.credit" field, MongoDB can rapidly filter documents that match the query condition without scanning every document in the collection. This optimization is particularly effective for queries that filter documents based on specific criteria, as evidenced by the drastic reduction in execution time.

Task 2 Analysis:

Q1) Is the performance different for normalized and embedded model? Why?

Answer: The normalized model tends to offer better performance for update-heavy operations, as seen in Query 4. It also benefits from indexing, particularly for queries that involve joins or lookups. The embedded model shows its strength in read operations, especially those that can be satisfied with a single document lookup, such as Queries 2 and 3, where data relatedness and document structure allow for faster retrieval without the need for joins.

Conclusion: The choice between a normalized and an embedded model depends on the specific requirements of your application. If your use case involves heavy read operations where data relatedness is critical, and minimizing joins can lead to performance gains, an embedded model might be preferable. Conversely, if your application involves frequent updates or operations where data normalization and integrity are paramount, a normalized model could offer better performance and flexibility.

Q2) Which model is a better choice for the query and why?

Answer: For queries 1, 2, and 3, the embedded model is a better choice. This is because the queries are “read-heavy” operations and the embedded model appears to perform better than the normalized model in this case, which can be seen in our outputs as well.

For query 4, the normalized is preferrable because it is updating the data, and normalized model appears to have a better runtime than the embedded one in this case, as also seen in our output.