hetbhara, Het Bharatkumar Patel
kjemhus, Nole Kjemhus
dmhoang, Manh Duong Hoang
pdadhani, Parth Dadhania

Report

(a) A general overview of your system with a user guide

System Overview:

The system is designed to load and query data from JSON files into MongoDB collections, analyzing the impact of different document models and indexing on query performance. It consists of four main scripts:

1. task1_build.py: Builds a normalized document store with separate collections for messages and senders.
2. task2_build.py: Builds an embedded document store with sender information embedded within each message document.
3. task1_query.py: Executes and analyzes queries on the normalized document store.
4. task2_query.py: Executes and analyzes queries on the embedded document store.

Setup:

1. Download all files from the github repository to a folder
2. In that same folder, copy in the messages.json file and senders.json file.
3. By now, you should have all these files in one folder (task1_build.py, task2_build.py, task1_query.py, task2_query.py, messages.json, senders.json)

Initiate MongoDB Server:

1. Create a data directory: mkdir ~/mongodb_data_folder
2. Start MongoDB server: mongod --port <portNumber> --dbpath ~/mongodb_data_folder &

To run task1:

1. In the command line, in the folder that includes all the files, the syntax to run the build file is:

   python3 task1_build.py <portNumber>
2. Make sure you run task1_build.py before continuing. Next, we will run task1_query.py with the same portNumber, run:

   python3 task1_query.py <portNumber>

To run task2:

1. In the command line, in the folder that includes all the files, the syntax to run the build file is:

   python3 task2_build.py <portNumber>

2. Make sure you run task2_build.py before continuing. Next, we will run task2_query.py with the same portNumber, run:
   python3 task1_query.py <portNumber>

Explore Database using MongoDB client:

1. Run the client listening to the same port of server (make sure the server is running in the background): mongosh --port <portNumber>
2. Navigate the database using following commands:
   a. Open database: use DATABASE_NAME
   b. List databases: show dbs
   c. Drop database: db.dropDatabase()
   d. Create collection: db.createCollection(name, options)
   e. Drop collection: db.COLLECTION_NAME.drop()
   f. Query collection: db.COLLECTION_NAME.find()

(b) <u>Your strategy to load large json files into mongodb</u>

Our strategy for loading large JSON files, such as messages.json, into MongoDB for both task 1 and task 2 involves reading the file line by line. Each line corresponds to a Python dictionary. For each line, we convert it into a Python dictionary and append it to an array called “batch.” We continue this process until the “batch” array contains 5000 dictionaries, representing 5000 messages. Once the “batch” reaches this size, we use the insert_many operation to add them to the collection in MongoDB. After that, we reset the “batch” array to an empty array and repeat this procedure until all lines have been processed.
Since the “senders.json” file is stated to be relatively small, we read the whole file at once using json.load, then insert_many the returned array in one batch.

(c) <u>The output for each query and a complete and accurate answer to all of the questions asked in Task 1 and 2 along with the relevant analysis</u>

Task 1 queries’ output:

Q1: The number of messages that have 'ant' in their text: 19551
Q1: Time needed to run the query: 1.2172069549560547 seconds, or 1217.2069549560547 milliseconds
Q2: The sender who has sent the greatest number of messages is: ***S.CC with 98613 messages
Q2: Time needed to run the query: 0.7574937343597412 seconds, or 757.4937343597412 milliseconds
Q3: Number of messages from senders with 0 credit: 15354
Q3: Time needed to run the query: 47.71829271316528 seconds, or 47718.29271316528 milliseconds
Q4: Time needed to run the query: 0.02294158935546875 seconds, or 22.94158935546875 milliseconds

Queries after creating indices:
Q1: The number of messages that have 'ant' in their text: 19551
Q1: Time needed to run the query: 1.188262701034546 seconds, or 1188.262701034546 milliseconds
Q2: The sender who has sent the greatest number of messages is: ***S.CC with 98613 messages
Q2: Time needed to run the query: 0.7203769683837891 seconds, or 720.3769683837891 milliseconds
Q3: Number of messages from senders with 0 credit: 15354
Q3: Time needed to run the query: 0.05900382995605469 seconds, or 59.00382995605469 milliseconds

Task 1 answers to questions:
Strategy to handle the large json file was discussed in (b)
Runtime to read the data and create the messages collection (step 1): 12.263454675674438 seconds
Runtime to read the data and create the senders collection (step 2): 0.08963251113891602 seconds
For query 1 and 2, the runtime didn't change much. For query 3, the runtime changed significantly.
For query 1, the runtime remained largely unchanged since both the pre-indexing and post-indexing queries rely on '$regex' for text matching and fail to fully utilize the text index via '$text'.
For query 2, the runtime remained somewhat the same because the index for 'sender' here didn't help much in this specific query as we're grouping and counting the number of documents for each sender.
For query 3, the runtime significantly changed because we added an index for 'sender_id' and 'sender', thereby substantially improving the matching of documents between the two collections via '$lookup'.

Task 2 queries' output:
Q1: The number of messages that have 'ant' in their text: 19551
Q1: Time needed to run the query: 0.687436580657959 seconds, or 687.436580657959 milliseconds
Q2: Sender with most messages: ***S.CC with 98613 messages
Q2: Time needed to run the query: 0.7895348072052002 seconds, or 789.5348072052002 milliseconds
Q3: Number of messages from senders with 0 credit: 15354
Q3: Time needed to run the query: 0.49136877059936523 seconds, or 491.36877059936523 milliseconds

Q4: Time needed to run the query: 571.0980207920074 seconds, or 571098.0207920074 milliseconds

Task 2 answers to questions:
Strategy to handle the large json file was discussed in (b)
Runtime for embedding sender information and loading messages: 13.891347169876099 seconds

Comprehensive Analysis of Query Run Times
Query 1 (Q1): Messages Containing 'ant'

- Normalized Model:
  - Before Indexing: 1.22 seconds
  - After Indexing: 1.19 seconds
- Embedded Model:
  - 0.69 seconds
- **Performance Difference:** Both models perform a similar operation, searching within a single collection. The improved performance in the embedded model (0.69 seconds) might be due to differences in document structure and indexing efficiency, even though both queries are conducted on a single collection without joining operations.

Query 2 (Q2): Sender with Most Messages

- Normalized Model:
  - Before Indexing: 0.76 seconds
  - After Indexing: 0.72 seconds
- Embedded Model:
  - 0.79 seconds
- **Performance Difference:** The performance is comparable between models, slightly favoring the normalized model after indexing. This indicates that MongoDB's aggregation framework efficiently handles grouping and sorting operations across both data models.

Query 3 (Q3): Messages from Senders with 0 Credit

- Normalized Model:
  - Before Indexing: 47.72 seconds
  - After Indexing: 0.059 seconds
- Embedded Model:
  - 0.49 seconds
- **Performance Difference:** The dramatic improvement in the normalized model after indexing (to 0.058 seconds) highlights the critical role of indexing in optimizing query performance. The embedded model's consistent performance (0.49 seconds) without such

dramatic changes suggests that indexing within embedded documents might not be as effective or needed for this particular query.

Query 4 (Q4): Doubling Credit for Senders with Credit < 100

- Normalized Model:
  - Time Needed: 0.0229 seconds
- Embedded Model:
  - Time Needed: 571.10 seconds
- **Performance Difference:** The significant discrepancy here is primarily due to the operation's nature in each model. In the normalized model, a single update within the senders collection is efficient. In contrast, the embedded model requires iterating through and updating multiple messages for each sender, leading to a much higher execution time.

Analysis and Recommendations:

- Is the performance different for normalized and embedded models? Why?
  Yes, the performance differs notably between the two models. The key reasons include:
  - **Data Locality:** In Q1, the embedded model shows better performance (0.69 seconds) than the normalized model before indexing (1.23 seconds). This suggests that even for operations within a single collection, the structure and organization of documents in the embedded model can offer performance benefits, likely due to reduced overhead in document retrieval.
  - **Indexing Efficiency:**The dramatic improvement in the normalized model for Q3 after indexing (from 48.47 seconds to 0.058 seconds) underscores the critical role of indexing in optimizing query performance. The normalized model benefits substantially from indexing, particularly for operations involving fields that can be indexed effectively, like sender and text.
  - **Update Complexity:** Q4 highlights a significant efficiency gap in update operations. The normalized model completes the operation in 0.024 seconds, contrasting starkly with the embedded model's 571.10 seconds. This is due to the normalized model's ability to target updates more directly at a centralized collection of senders, whereas the embedded model requires iterative updates across multiple documents, increasing complexity and execution time.

- Which model is a better choice for the query and why?
  - **For Read Operations (Q1, Q2, Q3):**
    The normalized model generally offers advantages, especially post-indexing, as seen with the drastic improvement in Q3. This model provides better scalability and efficiency for complex queries and benefits significantly from indexing, making it preferable for applications with intensive read operations that can leverage indexed fields.

The embedded model does show competitive or better performance in certain cases (like Q1), which may suit applications with simpler query requirements or those that predominantly access data from a single document context without the need for complex joins or aggregations.

- **For Write/Update Operations (Q4):**
  The normalized model is far more efficient for updates, particularly for operations involving relational data updates, as demonstrated by the quick execution of Q4. This model facilitates straightforward and cost-effective updates to related data, making it ideal for applications with frequent relational data updates or write-heavy workloads.
  Due to the substantial execution time seen in Q4 for the embedded model, it may not be suitable for applications requiring frequent updates to relational data across multiple documents. The complexity and time required for such updates in the embedded model could significantly impact application performance and scalability.