# Mini Project 2 Report

**General Overview + User Guide:**

Overview:

Our overall programs are a demonstration of different document storage systems and indexing methods available in a MongoDB database. It incorporates having to load large files into a database (which are too large for the machine's RAM), comparing times to upload into the database and times of various queries with and without indexing.

These scripts are all written in Python, making use of the pymongo library to connect to a running MongoDB database. Other libraries include sys to read command-line arguments, json to convert text from the json files into valid json documents, and time to measure the time it takes for queries or database inserts to happen.

*Build Scripts:*

task1_build.py:

- This script loads all of the contents from senders.json and messages.json into the MP2Norm database on a running MongoDB server with a given port number. The documents are NORMALIZED, not embedded, i.e. a senders collection and messages collection is created. The time it takes to read and insert all of the data is printed when running the script.

task2_build.py:

- This script loads all of the contents from senders.json and messages.json into the MP2Embd database on a running MongoDB server with a given port number. However, these documents ARE EMBEDDED, so in a message document the information about its sender is stored inside the document itself. As such, only a messages collection is created and populated. The time it takes to read and insert all of the data is printed when running the script.

*Query Scripts*:

task1_query.py

- This script connects to the running MongoDB server with a given port number and runs four queries on the MP2Norm schema. The queries are first ran without any special indices (printing the outputs and time taken to execute), and then indices are added for 'sender', 'text', and 'sender_id'. The queries are then re-ran without the indices and the new outputs and times are printed.

task2_query.py

- This script connects to the running MongoDB server with a given port number and runs four queries on the embedded MP2Embd schema. No indices are made, so the four queries are simply ran and their results and times are printed.

User guide (code instructions):

**First: Have a running MongoDB server instance:**

- Ex) mongod --port <portnum> --dbpath ~/mongodb_data_folder &

**And manually download senders.json and messages.json into the root of the repository!**

Now all python scripts can be ran using 'python3 <scriptname> <portnum>'

- Ex) python3 task1_build.py <portnum>
  python3 task1_query.py <portnum>
  python3 task2_build.py <portnum>
  python3 task2_query.py <portnum>

*Task 1*:

- First, run task1_build.py to populate the MP2Norm database
  - The time it takes to load and insert the data to the database is printed
- Then you can run task1_query.py to execute the queries
  - This will run them without indices, then insert indices, then run them with indices. All queries are timed and times are printed along with results

*Task 2*:

- First, run task2_build.py to populate the MP2Embd database
  - The time it takes to load and insert the data to the database is printed
- Then you can run task1_query.py to execute the queries
  - The queries are run without indices and the times are printed with results

Handling large json files:

Because the messages.json file is so large, it is not possible to simply load the entire file into RAM and read it that way to insert into the database. To account for this, we had to insert the data into the database in "small chunks" of about 5000 documents at a time. This way, we only ever have at most 5000 documents loaded into RAM at any given time, and once we insert these 5000 documents we can free up the memory space so that we don't use more than the computer can handle. Our implementation was this:

- (Repeat this process until every line from the file has been read)
- Read in a single line from the file, convert it to json format
- Append it to the list of currently loaded documents
- If 5000 documents are currently loaded (list length is 5000):
  - Upload all loaded documents to the database
  - Free up all loaded documents from memory

Example of implementation in task1_build.py:

```
    # If 5000 messages loaded, insert into the database and clear from memory
    if len(loadedDocuments) == 5000:
        messagesCol.insert_many(loadedDocuments)
        loadedDocuments = []
```

Example of implementation in task2_build.py

```
def process_messages(filepath, messagesCol, sender_lookup, batch_size=5000):
    """
    Processes messages from a file and inserts them into the messages collection.

    Parameters:
        filepath (str): The path to the file containing messages.
        messagesCol (pymongo.collection.Collection): The collection to insert messages into.
        sender_lookup (dict): A dictionary mapping sender IDs to sender information.
        batch_size (int): The size of each batch of messages to insert.
    """
```

**<u>Query Outputs</u>:**

**TASK 1 build, normal non embedded document store**

```
jgourley@ug01:~/cmput291/w24-mp2-mongomaxxers>python3 task1_build.py 54321
Total time elapsed to read and populate into senders collection: 0.24423813819885254 seconds
Total time elapsed to read and populate into messagess collection: 12.058806419372559 seconds
jgourley@ug01:~/cmput291/w24-mp2-mongomaxxers>
```

The script is designed to display proper usage if the user input command is wrong. If usage is right, builds the database “MP2Norm” on the Mongodb server running on the given port. Runtime will change depending on the size of the dataset and each run can have a differing runtime due to hardware fluctuations and connection to lab machines if run through ssh.

**TASK 2 build, embedded document store**

```
dricmoy@ug08:~/w24-mp2-mongomaxxers>python3 task2_build.py
Usage: python3 task2_build.py <database_port>
dricmoy@ug08:~/w24-mp2-mongomaxxers>python3 task2_build.py 54321
Total time to read data and create the collection: 14.185860872268677 seconds
```

The script is designed to display proper usage if the user input command is wrong. If usage is right, builds the database “MP2Embd” on the Mongodb server running on the given port. Runtime will change depending on the size of the dataset and each run can have a differing runtime due to hardware fluctuations and connection to lab machines if run through ssh.

**Task 1 query usage:**

```
dricmoy@ug08:~/w24-mp2-mongomaxxers>python3 task1_query.py
Usage: python script.py <port>
```

**Task 1 query outputs in order:**

```
dricmoy@ug08:~/w24-mp2-mongomaxxers>python3 task1_query.py 54321
Connected successfully to MongoDB
Number of messages containing 'ant': 19551
Time taken: 688.9400482177734 milliseconds
----------------------------------------
Sender with the most messages: ***S.CC (Messages sent: 98613)
Time taken: 519.1056728363037 milliseconds
----------------------------------------
Number of messages from senders with credit 0: 15354
Time taken: 37556.947469711304 milliseconds
----------------------------------------
Double credit of senders that have credit less than 100: 12.754201889038086 milliseconds
----------------------------------------
Indices created successfully.
Time taken: 17667.64545440674 milliseconds
Index have been created

Number of messages containing 'ant': 19551
Time taken: 723.4091758728027 milliseconds
----------------------------------------
Sender with the most messages: ***S.CC (Messages sent: 98613)
Time taken: 519.4265842437744 milliseconds
----------------------------------------
Number of messages from senders with credit 0: 15354
Time taken: 49.479007720947266 milliseconds
----------------------------------------
```

**Task 2 query usage:**

```
dricmoy@ug08:~/w24-mp2-mongomaxxers>python3 task2_query.py
Usage: python script.py <port>
```

**Task 2 query outputs in order:**

```
dricmoy@ug08:~/w24-mp2-mongomaxxers>python3 task2_query.py 54321
Connected successfully to MongoDB
Number of messages containing 'ant': 19551
Time taken: 696.5334415435791 milliseconds
----------------------------------------
Sender with the most messages: ***S.CC (Messages sent: 98613)
Time taken: 1033.0758094787598 milliseconds
----------------------------------------
Number of messages from senders with credit 0: 15354
Time taken: 491.43075942993164 milliseconds
----------------------------------------
Double credit of senders that have credit less than 100: 1744.4605827331543 milliseconds
```

## Reasons for runtime differences between queries for non-indexed vs indexed:

Query 1: No change

The runtimes are similar: **~688.94 ms** without an index and **~723.41 ms** with an index. Indexing on the 'text' field is not effective when counting messages containing 'ant'. Substring searches can't use an index efficiently because the index is designed for whole string matches or prefix searches, not for locating substrings within text. Therefore, MongoDB has to scan through each document in the collection to check for the presence of the substring in either case, so there is no improvement with indexing.
**Better choice:** non-indexed, do not need to create indices which require additional resources with no improvement in query runtime.

Query 2: No change

The runtimes are similar: **~519.11 ms** without an index and **~519.41 ms** with an index. The task involves grouping all messages with their senders and counting them; then identifying the sender with the maximum number of messages. Given the need to group and then find the max—indexing is ineffective. Indexes can help with direct searches or sort operations on indexed fields, but don't when finding aggregations across grouped data.
**Better choice:** non-indexed, do not need to create indices which require additional resources with no improvement in query runtime.

Query 3) Change

With no indexing: the query takes **~37556.95 ms**. With indexing: the query takes **~49.48 ms** The reduction in time for this query post-indexing is because MongoDB can utilize the index to quickly locate all senders with credit equal to 0 instead of scanning through every document. With the use of an index, MongoDB avoids a full collection scan and directly accesses the relevant documents, thereby reducing the execution time.
**Better choice:** Indexing approach is significantly better

Query 4) No change

The runtimes are similar: **~742.16 ms** without an index and **~740.95 ms** with an index.
The reason for no change in runtime is due to the lack of a specific index on the 'credit' field. For this query, the system must iterate over every sender to assess whether their credit is less than 100 before applying the update. Without an index directly targeting the 'credit' field, the MongoDB cannot bypass the step of checking each sender's credit. Therefore, the query must go over the entire collection.
**Better choice:** non-indexed, do not need to create indices which require additional resources with no improvement in query runtime.

## Reasons for runtime differences between queries for normalized (+ non indexed) vs embedded:

NOTE: we are comparing to non-indexed normalized document store

Query 1) No change

In the embedded scenario provided, it took **696.53 ms**. Comparing this to the non-indexed normalized scenario where the runtime was **688.94 ms**, the difference is minimal. This slight variation is likely due to normal fluctuations in query execution rather than any impact from the database schema since Q1 doesn't involve sender information and thus doesn't utilize sender embedding or indexing. The query asks to Return the number of messages that have "ant" in their text. However, this is not affected by any changes made to the senders' collection. Regardless of if the senders' collection is maintained as a separate collection or if it is embedded within messages.

**Better choice:** Embedded model can be a better choice considering the fact that it requires less storage space due to avoiding separate collections but runtime wise we did not find a significant change between the models.

Query 2) Changed

Given the embedded model's query execution time of **~1033.08 ms** for Q2, and comparing it to the significantly lower runtime of **~519.11 ms** in the non-indexed normalized model, we can conclude there is indeed a significant difference. The increased runtime in the embedded model likely stems from the complexity of aggregating over nested fields, as opposed to the more straightforward aggregation over top-level fields in the normalized model. This supports the idea that for operations like grouping, which is central to Q2, an embedded model can introduce additional processing overhead due to the nested document structure, thus leading to slower performance compared to a normalized schema where sender information is stored in a separate collection and can be aggregated more directly.
**Better choice:** Normalized model, since there is no complexity of aggregating over nested fields

Query 3) Changed

This query asks to return the number of messages sent by users with a credit of 0. In task 1, you must first find all of the senders with a credit of 0 and count the number of messages sent by that user. However, in task 2, because the sender info is embedded into messages. You must go through just the messages, and then check if the sender credit associated with that message is 0.

In a normalized design with separate collections (or tables, if thinking in terms of a relational database), you would typically need to perform a join operation between the messages collection and the sender collection to filter messages based on sender credit. The query might involve a lookup to match sender IDs in the messages with sender records, and then filter those records where the credit is 0. When sender information is embedded directly in the messages collection, each message document contains the sender's credit information within it. This eliminates the need for a join or lookup operation. The query can directly access and filter the messages based on the embedded credit attribute. This simplifies the query and improves its performance.
**Better choice:** Embedded, due to directly being able to access and filter messages based on the sender's credit from the embedded collection and not have to lookup a different collection.

Query 4) Changed

Normalized model runtime is **~12.75 ms** while Embedded model runtime is **~1744.46 ms.**
The normalized model has way better performance compared to the embedded model. The Embedded model does not require a separate senders table, instead, it incorporates sender information directly into the messages collection as an attribute. However, this insertion process requires significant overhead. In the normalized model, senders and messages are maintained in separate collections.

This distinction is why normalized runs faster than embedded. In Task 1 (normalized model), senders are stored in a separate collection. If we have 970 senders with less than 100 credits, updating their information involves modifying the records of these 970 senders directly. However, in Task 2 (embedded model), sender details are embedded within each message. This means if these 970 senders, each with less than 100 credits, have sent multiple messages, updating their information requires adjusting the sender details in every message they've sent.
**Better choice:** Normalized model does significantly better, since the embedded model needs to go through each and every message by a sender whose credit is less than 100. For example, 10 senders with 10k messages will lead to 100,000 updates vs Normalized where there would be 10 updates only.