Task 1:

General Overview:

task1_build.py
Our code at the beginning of the program is set up so we expect a port number entered to the command line by the user. Then we try connecting to the database. The next section of code is reading and inserting into the collections. First we read from messages.json and insert to the message collection then we read from senders.json and insert into the senders collection.

task1_query.py
In task1_build.py we have two functions read_messages and read_senders these are intended to read the information from the two given jsons messages.json and senders.json and then insert into the collections. These two functions will later be used in step 4. Next, we verify the user has entered in a port number and we then attempt a connection. The code for step 3 is our queries which returns the expected output and prints the execution times of the queries. Step 4 is where we use indices for the queries. We start off by resetting the collections and then creating the proper indices. Execution times for the queries after creating indices is noticeably improved for query 3. Lastly we drop the indices that were created.

Step 1:
The strategy to handle the large json file from messages.json was to json.load each line individually instead of loading the entire data set. We then stored those lines into batches of 5k. This batch was then being loaded onto the collection once it hit the threshold. This process would continue until all lines have been processed.

Step 2: Time required to read data and create the collection

Execution time for messages: 9.231380939483643 seconds
Execution time for senders: 1.5277297496795654 seconds

Step 3:

Query 1 execution time = ~2.1993870735168457 seconds
- Number of messages that have “ant” in their text: 19551

Query 2 execution time = ~0.508289098739624 seconds
- ***S.CC sent the most messages, total messages: 98613

Query 3 execution time = ~41.37273597717285 seconds
- Number of messages where the sender's credit = 0 is 15354

Query 4 execution time = ~0.020133018493652344 seconds
- Number of senders whose credit was less than 100: 970

Step 4: Explain how and why indexing affects your runtime for each query.

We created indexes for both collections. Sender and text fields in messages collection. Sender_id field for senders collection. The purpose of indexes is so that those specific fields are sorted, such that searching through it would be faster and more efficient. The type of indexing we had for both the sender and sender_id of messages and senders collection respectively, was of default type (ascending order) so for those fields, the values would be sorted in increasing order. For the text field of messages collection, we had text indexing, which is primarily for full text search.

Query 1 execution time = ~2.1715288162231445 seconds
- Number of messages that have "ant" in their text: 19551

Query 2 execution time = ~0.5070221424102783 seconds
- ***S.CC sent the most messages, total messages: 98613

Query 3 execution time = ~0.05642294883728027 seconds
- Number of messages where the sender's credit = 0 is 15354

1. The run times for query 1 did not change. This is simply due to the fact that using regex to find certain key phrases in words doesn't really help. The message could be anything which is why indexing the text field is practically useless. It would benefit most if we used search for a whole word instead, but search does not find specific texts inside of complete words. Text indexing is not fully optimized with $regex
2. The run time for query 2 also did not change. This is because the stage "$sortbycount" is already a powerful aggregation stage. It groups all the senders and counts and takes the top choice via limit stage, which should be the sender with the most messages sent. So with indexing, the pipeline will perform exactly the same
3. The run time for query 3 however, did change drastically. It went from 41 seconds to half a second because of the indexing. The senders, and the sender_id were ordered in ascending order. Meaning that when joining the two collections via the $lookup stage, it's only processing what it has to process, and nothing more. So matching the sender_id with sender will greatly improve the performance of the query.

Task 2:

General Overview:

task2_build.py
Similar to task 1, our code at the beginning of the program is set up so we expect a port number entered to the command line by the user. Then we try connecting to the database. Once the connection is successful, with a simple if statement we check whether or not the messages database already exists, and if it does we drop it. We then read the senders file, and then in batches read the messages file line

by line and embedded senders info into json into messages. We read messages in batches of 5000 and this is how we handled the possibility of having large json files which are too large to fit in memory as inputs.

The process we used to embed senders into messages in batches can be broken down into 4 steps:

1. We created a dictionary using the senders data, and indexed it by sender id. This allowed our program to run much quicker as the use of the dictionary made it easier and faster to look up senders information
2. We then iterated through each message in the messages batch and retrieved the senders info from the dictionary we created earlier
3. Once we retrieved the senders info we embedded it into the message
4. Once we finished embedding the senders info into each message of the batch, we inserted the embedded messages data into the messages file

task2_query.py
We performed our queries on the messages collection with embedded sender info. Before running the queries we verified the user has entered in a port number and we then attempt a connection. We then made methods for each query and ran each one of them, making sure to output their execution times.

Step 1: Time taken to embed sender info into messages collection: ~15.138920068740845

Step 2:

Query 1: Number of messages containing "ant"
Task 1 - Without indexes: ~1.51695322 seconds
Task 1 - With indexes: ~1.5239217813 seconds
Task 2: ~1.5508879716 seconds

The performance improvement from indexing is minimal for this query. Both normalized and embedded models show similar performance because the operation scans through message contents, which is not drastically optimized by indexing structures.

Choice of Model: For this query, the choice between normalized and embedded models is relatively inconsequential since the performance difference is marginal. However, we think the normalized model would still be the better choice since it has a slightly faster time frame, and less documents to scan through.

Query 2: Messages sent by the most active sender
Task 1 - Without indexes: ~0.507258415 seconds
Task 1 - With indexes: ~0.5159549 seconds

Task 2: ~1.032551288 seconds

This query involves aggregating messages by sender and identifying the sender with the most messages. Due to the fact that we had to use a sort stage in task 2, the consequences would suggest a slower process time. This is because it would take a longer time to process more documents while also sorting them at the same time.

Choice of Model: For Query 2 and similar aggregation operations where grouping and analyzing data across multiple documents are common, the normalized model is a better choice. The embedded model may be suitable for scenarios where documents are small and self-contained, and there is a strong one-to-one relationship between embedded documents. However, for queries involving aggregation, normalization would be the correct choice of model.

Query 3: Messages where the sender's credit is 0
Task 1 - Without indexes: ~37.46565151214 seconds
Task 1 - With indexes: ~0.03952598571 seconds
Task 2: ~0.4876780 seconds

Here, indexing shows a dramatic improvement in performance. The drastic reduction in execution time from ~37 seconds to less than a second with indexing indicates that operations involving sender fields (like sender_info) and messages fields (like senders and text) are highly optimized by indexes, due to efficient lookup capabilities via sorting.

Choice of Model: The model with normalized data including the presence of indexing is clearly superior for this query. This is because, due to the indexing choices of sender and text from messages collection and sender_info from senders collection, this allows the documents to be sorted in such a way that the lookup times are greatly reduced. The system also knows that, since it's in default type indexing (ascending) it knows when to stop processing the collection.

Query 4: Doubling credit for senders with less than 100 credit
Task 1 - Without indexes: ~0.0126500129 seconds
Task 1 - With indexes: not needed
Task 2: ~1.426425838470459 seconds

In query 4, we have noticed that the runtime for the normalized version is significantly faster. When we embedded the two collections, we didn't account for duplicates, signifying that senders appeared more than once, making the whole collection a lot larger. Due to this fact, it takes a lot longer to process the documents.

Choice of model: Since the normalized model is significantly faster than the embedded model, going with the normalized would only make more sense. Searching through significantly more documents would only make it less efficient.