Overview

All of the source code is written in Python.
task1_build.py performs steps 1 and 2 of task 1.
task1_query.py performs steps 3 and 4 of task 1.
task2_build.py performs step 1 of task 2.
task2_query.py performs step 2 of task 2.
task1_build.py and task2_build.py assume that two files, “messages.json” and “senders.json”, are in the same folder as the Python application.
task1_query.py and task2_query.py assume that the database is already created.
We also provided two additional files task1_all.py and task2_all.py that perform all of task 1 and task 2 respectively.
All of the above scripts assume that a local MongoDB server is being hosted on a particular port number at run time.
All of the above scripts take exactly one command line argument, which is the port number that a local MongoDB server is being hosted on.

User guide

Set up

1. Ensure you have Python 3.6 or higher, MongoDB, and PyMongo installed on your machine.

2. Clone the repository to your local machine.

3. Navigate to the project directory in the terminal.

4. Create a folder for the database path: mkdir ~/mongodb_data_folder

5. Run the server in the background listening on a specific valid port: mongod --port 27017 --dbpath ~/mongodb_data_folder

Perform the tasks

First, make sure you have the proper “messages.json” and “senders.json” files in the same directory as the program scripts. Then, open a different terminal shell, and navigate to the project directory.

Task 1

To perform steps 1 and 2, run `python3 task1_build.py 27017`. Make sure to change `27017` if you have hosted the MongoDB server on a different port. The program will print the time it took to perform each step in the terminal.

To perform steps 3 and 4, run `python3 task1_query.py 27017`. Make sure to change `27017` if you have hosted the MongoDB server on a different port. The program will print the time it took to perform each query for each step in the terminal.

Task 2

To perform step 1, run `python3 task2_build.py 27017`. Make sure to change `27017` if you have hosted the MongoDB server on a different port. The program will print the time it took to perform the step in the terminal.

To perform step 2, run `python3 task2_query.py 27017`. Make sure to change `27017` if you have hosted the MongoDB server on a different port. The program will print the time it took to perform each query in the terminal.

Strategy to Load JSON Files:

We implemented a helper function in utils.py to read in JSON files character by character. This design is to address any possible line break format of the JSON file with high flexibility and reusability: even if the file is saved all on one line (unfortunately, this is the default behavior of Python's json library), the file will be read in small batches.

The detailed breakdown of our algorithm is as such:

Initialization and variable functionality:

An empty "pool" array is created to hold such characters read; an array will be much quicker than string concatenations for this situation. Three variables has been used to denote the current context:

- Indentation: sections such as object {}, array [] will contribute to this by 1. It initially would start as -1; when it reaches -1 for another time, the list of documents should have been completely identified. Hence, this variable also handles the termination of this algorithm.
- String: flags whether the current character is within a string section. This is very crucial, as characters such as brackets will be present within strings; without this flag, those characters will disrupt the algorithm.
- Escaped: flags whether the current character is escaped. This will be set to true as long as the current character is "\"; otherwise, the Escaped flag will be reset to false, and the current character will be left as-is and appended to the "pool". This has a very high precedence to prevent escaped quotes etc. breaking our String flag.

Read:

The characters are read one by one. The logics to read in the document are shown above.

Note that as the Indentation starts as -1, it means our documents shall have Indentation = 0. In another word:

- After a closing bracket turning indentation into 0, we construct a string using the "pool", parse it into JSON and yield it.
- Furthermore, the "pool" would be cleared before recording an opening bracket while Indentation = 0; in this way, the comma and space-like characters between documents will be elegantly handled and removed.

Termination and Error Handling:

If the JSON file follows the proper format, the code shall yield the document objects one by one then terminate. However, if any mistake is present, there are two mechanisms to inform the user:

- If the end of file has been reached: recall that when the closing bracket of the documents array will turn Indentation back to -1, marking the end of function execution. Hence, for properly formatted JSON files, EOF should never be met.
- If the json library fails to interpret the pooled current document: this will mean that either the function encountered a bug (still technically possible, although it has been carefully tested) or the document section from the file is malformed.

Task 1

Step 1

Reading and inserting messages took 103.04713249206543 seconds

Step 2

Reading and inserting senders took 0.3464374542236328 seconds

Step 3 & 4

		Step 3	Step 4
Q1	Query output	19551	19551(regex), 39(search)
	Time (s)	0.8129444122314453 (regex)	0.761592 (regex), 0.007447 (search)
	Time (ms)	812.9444122314453 (regex)	761.592 (regex), 7.447(search)
Q2	Query output	[{'_id': '***S.CC', 'count': 98613}]	[{'_id': '***S.CC', 'count': 98613}]
	Time (s)	0.7104976177215576	0.7150797843933105
	Time (ms)	710.4976177215576	715.0797843933105
Q3	Query output	[{'message_count': 15354}]	[{'message_count': 15354}]
	Time (s)	43.770519495010376	0.05208563804626465
	Time (ms)	43770.519495010376	52.08563804626465
Q4	Query output	UpdateResult({ 'n': 1063, 'nModified': 970, 'ok': 1.0, 'updatedExisting': True }, acknowledged=True)	N/A
	Time (s)	0.022984027862548828	N/A
	Time (ms)	22.984027862548828	N/A

Q1: Number of Messages Containing "ant" in Their Text

- Analysis: The run time decreased slightly after indexing. However, the $regex operator that we used for finding messages with “ant” in their text did not use the text index. Therefore, indexing did not affect the running time of the query we developed for Q1.

Q2: Sender With the Greatest Number of Messages

- Analysis: The run time increased slightly after indexing. However, the query we developed for Q2 did not use any of the indexes. Therefore, indexing did not affect the running time of the query we developed for Q2.

Q3: Number of Messages Where Sender's Credit is 0

- Analysis: Indexing significantly decreased the run time of the query that we developed for Q3 from 43 s to 0.05 s. In our query, we used an aggregation pipeline on the senders collection where we first found only the senders with 0 credits. Then we perform a lookup operation on the messages collection with the condition senders.sender_id = messages.sender. Before indexing, we had to scan the entire messages collection looking for the messages with a particular sender. After indexing, however, MongoDB creates a B+ tree index on the 'sender' field of the messages collection. This index allows us to find all the messages of a particular sender in logarithmic time, instead of the linear time before indexing.

Task 2

Step 1

Reading and inserting messages took 97.50559592247009 seconds

Step 2

Q1	Query output	19551
	Time (s)	0.868628740310669
	Time (ms)	868.628740310669
Q2	Query output	[{'_id': '***S.CC', 'count': 98613}]
	Time (s)	0.834162712097168
	Time (ms)	834.162712097168
Q3	Query output	15354
	Time (s)	0.6189351081848145
	Time (ms)	618.9351081848145
Q4	Query output	UpdateResult({ 'n': 135138, 'nModified': 119784, 'ok': 1.0, 'updatedExisting': True }, acknowledged=True)
	Time (s)	5.044574499130249
	Time (ms)	5044.574499130249

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

1. Q1: Number of Messages Containing "ant" in Their Text.
   The execution time using the embedded model is slightly longer than that using the normalized model. This is because each document in the messages collection of the embedded model is slightly bigger than its counterpart in the normalized model. This means fewer documents in one page, and more pages need to be fetched to execute the query.

2. Q2: Sender With the Greatest Number of Messages.
   The execution time using the embedded model is slightly longer than that using the normalized model. This is because each document in the messages collection of the embedded model is slightly bigger than its counterpart in the normalized model. This means fewer documents in one page, and more pages need to be fetched to execute the query.

3. Q3: Number of Messages Where Sender's Credit is 0.
   The execution time for Q3 using the embedded model (0.62 s) is much shorter than that using the normalized model without indexes (44 s). This is because, with the embedded model, the sender's credit is embedded in each message. We just need to scan the messages collection once to count the ones with a sender's credit of 0. Using the normalized model without indexes, however, we had to scan the entire messages collection for each sender with 0 credits, thus taking much longer. In contrast, using the normalized model with indexes, the execution time was 0.05 s, which is faster than using the embedded model. Using the normalized model with indexes, we use the index to find all messages for each sender with 0 credits. The execution time is linearithmic (number of senders with 0 credits times the logarithmic time to traverse the index), but the number of senders with 0 credits (93) is much less than the number of messages (~1.2 million).

4. Q4: Double the credit of all senders whose credit is less than 100.
   With the normalized model, the number of senders whose credit is less than 100 is 1063, of which 970 were updated (93 senders have a credit of 0) in about 23 ms. With the embedded model, all the messages with a particular sender have a copy of the sender's information. If that sender's credit needs to be updated, we must update all the copies to keep the data consistent. This is why we found 135138 messages with a sender that had less than 100 credits and 119784 of which were updated in 5 s. In short, the embedded model has redundant data, causing more updates, and thus taking more time.

2) Which model is a better choice for the query and why?
   1. Q1: Number of Messages Containing "ant" in Their Text.
      The normalized model is better because each document is smaller without the embedded sender's information. Therefore, we can fit more documents into one page and we can scan the entire collection with less page access than the embedded model.

   2. Q2: Sender With the Greatest Number of Messages.
      The normalized model is better because each document is smaller without the embedded sender's information. Therefore, we can fit more documents into one page and we can scan the entire collection with less page access than the embedded model.

   3. Q3: Number of Messages Where Sender's Credit is 0.
      Suppose we can create an index on the 'sender' field of the messages collection. In that case, the normalized model is better because the query executes an order of magnitude faster than

the embedded model. If we cannot create an index on the ‘sender’ field of the messages collection, then the embedded model is better as it avoids the expensive lookup operation.

4. Double the credit of all senders whose credit is less than 100.
   The normalized model is better because we store only one copy of the information of each sender in the senders collection. That means when we update any sender’s information, we only need to update one copy, and hence not have to worry about inconsistent data.