Mini Project 2 Report

Toes 3

Obaloluwa Odelana (odelana)

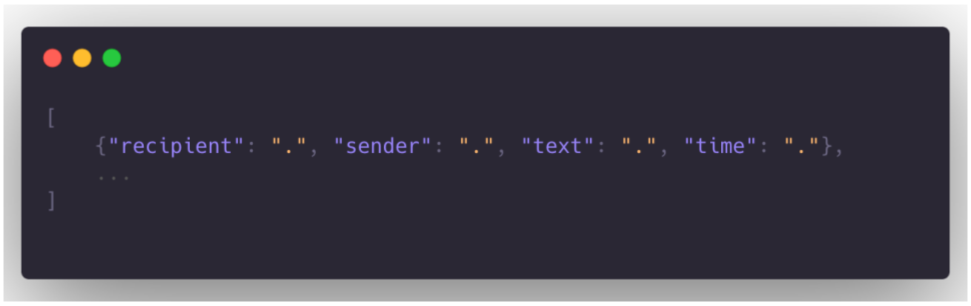
Tomiwa Orimoloye (oorimolo)

Excel Ojeifo (eojeifo)

Sean Carson (swcarson)

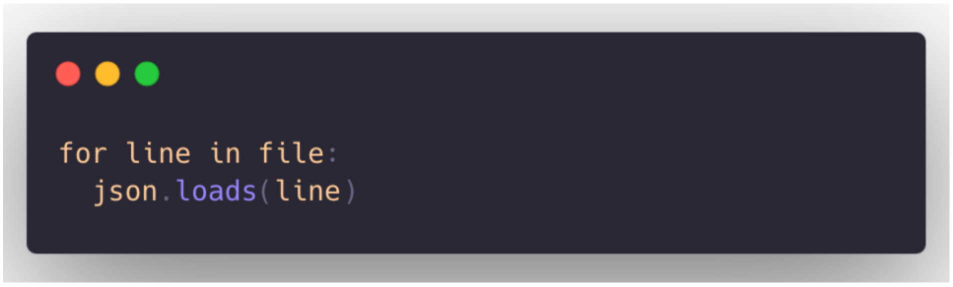
Strategy for Handling Large JSON File

Given the structure of the `messages.json`:



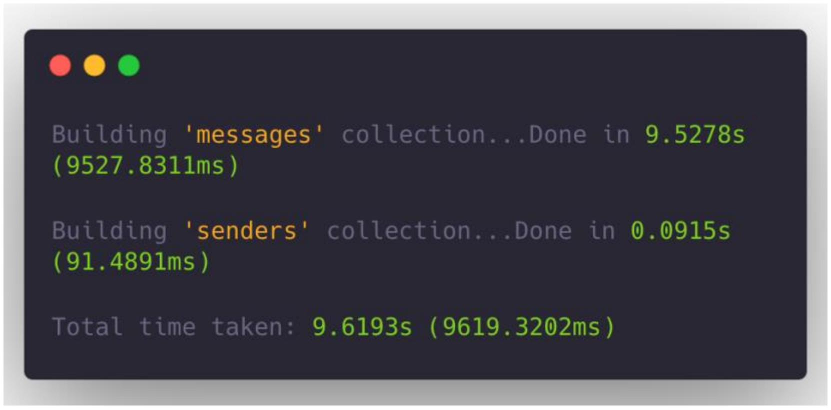
```
[
    {"recipient": ".", "sender": ".", "text": ".", "time": "."},
    ...
]
```

We see that the file is a list of documents, hence, our approach to parsing the file was to treat each line as a JSON file. That is, parsing each line in isolation. To make this work, we removed the trailing commas on each line and skipped the lines containing the opening and closing brackets. After this, our code is straightforward:



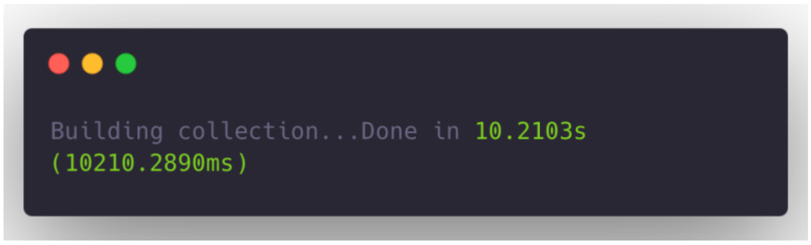
```
for line in file:
  json.loads(line)
```

Comparing Runtimes for Building Document Stores



```
Building 'messages' collection...Done in 9.5278s
(9527.8311ms)

Building 'senders' collection...Done in 0.0915s
(91.4891ms)

Total time taken: 9.6193s (9619.3202ms)
```

Task 1

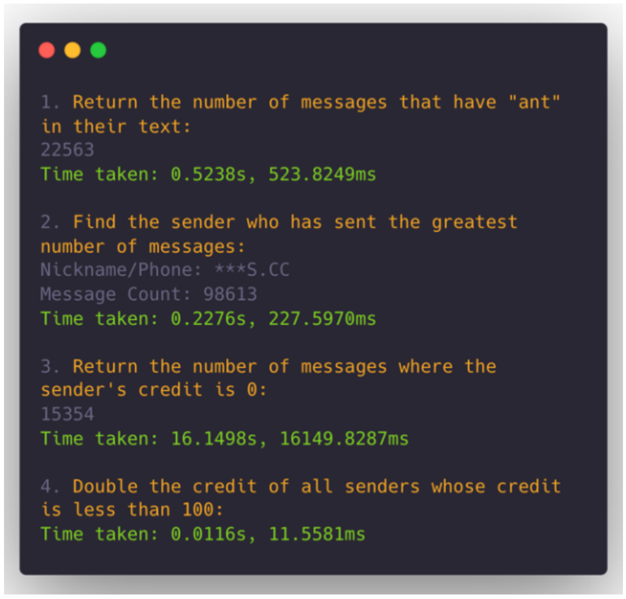


```
Building collection...Done in 10.2103s
(10210.2890ms)
```

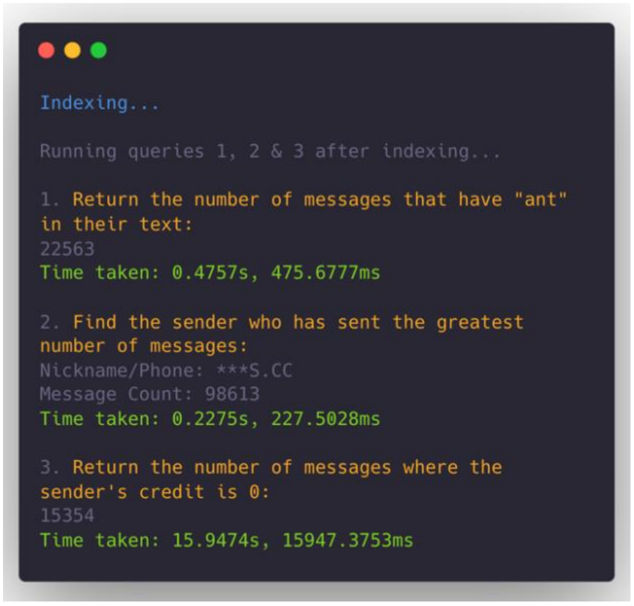
Task 2

Task 1 takes less time overall. This is possibly due to parsing and creating each collection individually. Both tasks involve nearly identical steps, the difference lies when creating the \`messages\` collection in Task 2. For task 2, an additional memory access operation is done in each iteration to retrieve data from the \`senders\` dictionary.

Impact of Indexes



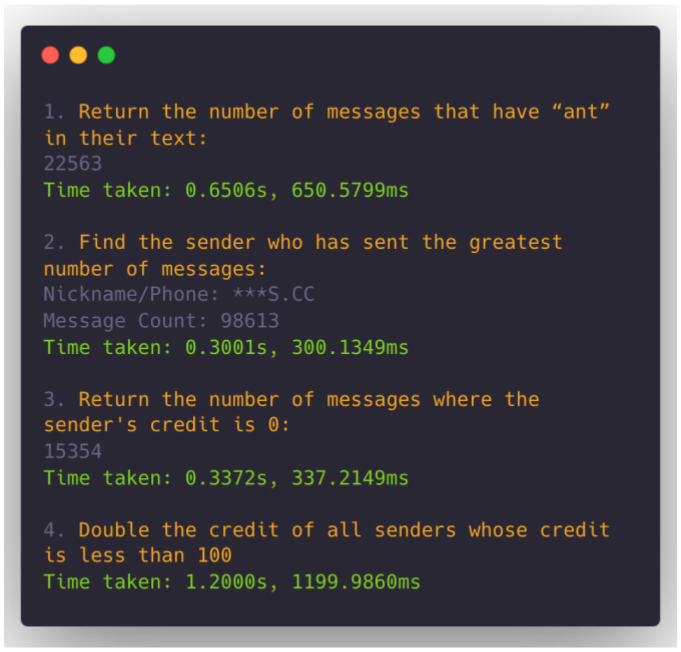
```
1. Return the number of messages that have "ant"
in their text:
22563
Time taken: 0.5238s, 523.8249ms

2. Find the sender who has sent the greatest
number of messages:
Nickname/Phone: ***S.CC
Message Count: 98613
Time taken: 0.2276s, 227.5970ms

3. Return the number of messages where the
sender's credit is 0:
15354
Time taken: 16.1498s, 16149.8287ms

4. Double the credit of all senders whose credit
is less than 100:
Time taken: 0.0116s, 11.5581ms
```



```
Indexing...

Running queries 1, 2 & 3 after indexing...

1. Return the number of messages that have "ant"
in their text:
22563
Time taken: 0.4757s, 475.6777ms

2. Find the sender who has sent the greatest
number of messages:
Nickname/Phone: ***S.CC
Message Count: 98613
Time taken: 0.2275s, 227.5028ms

3. Return the number of messages where the
sender's credit is 0:
15354
Time taken: 15.9474s, 15947.3753ms
```

We see that there is a decrease in the runtimes of each query after indexing. Indexing limits the number of documents that must be scanned. Rather than linearly searching through the whole collection, an index helps to efficiently identify relevant data. In our case, a text index boosts performance by mapping terms to documents, hence, rather than searching through all documents, we only look at each term, thereby speeding up retrieval. Furthermore, MongoDB's use of the B-tree data structure allows for sub-linear searching of keys, leading to an increase in performance.

Embedded Document Store



```
1. Return the number of messages that have “ant”
in their text:
22563
Time taken: 0.6506s, 650.5799ms

2. Find the sender who has sent the greatest
number of messages:
Nickname/Phone: ***S.CC
Message Count: 98613
Time taken: 0.3001s, 300.1349ms

3. Return the number of messages where the
sender's credit is 0:
15354
Time taken: 0.3372s, 337.2149ms

4. Double the credit of all senders whose credit
is less than 100
Time taken: 1.2000s, 1199.9860ms
```

When compared to the normalized document store,

1. For query 1, the embedded document store performs worse than its counterpart. However, when running both implementations multiple times, they have similar running times. A possible cause for the inconsistency might be due to their difference in size (the embedded database is larger).
2. For query 2, the query code is the same for both implementations, however, the difference in size causes the embedded model to take a bit more time.

For queries 1 and 2, the normalized document store is a better model, for two reasons: it avoids redundancy and duplication of data, and it has a smaller file size.

3. For query 3, the embedded model is almost 50x faster than the normalized model. This is largely because the former avoids joins. The embedded model only performs a linear scan of a single collection to answer the query. The normalized model, on the other hand, must linearly scan through the `messages` collection for every sender in `senders`.

   In this case, the embedded model is a far better choice.
4. For query 4, the normalized model does better due to its smaller file size. In this case, the normalized model is a better choice.