TTDS Group Project - Books and Book Quotes Search Engine

s1603986, s1735833, s1757868, s1763241

Abstract

The Books and Book Quotes Search Engine is a system that finds books, book quotes and book details for a particular set of books. The book dataset consists of 24,985 books, from which there are 40,999,867 quotes. The search engines uses two inverted index structures to provide quote search, strict phrase search and book search functionalities. For a quote query, the system returns the most relevant quotes, their contexts and their book details; the relevant quotes will be ranked with respect to their BM-25 score. For a book query, the system returns the most relevant books using terms appearing in the book title; the relevant books will be ranked with respect to their TF-IDF score. For a phrase query, the system returns the quotes that contain the exact search phrase. We implemented the search engine in the form of a web application, with the frontend providing the user interface and the backend responsible for querying the database and ranking the results. The web application can be found at book-search-ttds14.surge.sh and the repository containing the codebase can be found at https://github.com/yussefsoudan/ttds-14.

1 Introduction

"Books are a uniquely portable magic."

- Stephen King

The importance of reading has long been highlighted, with books often taking the readers on a journey to view the world as someone other than themselves that experiences it, thereby allowing them to gain greater perspective on the world and on others. Many books also contain words and lessons that may touch readers even when the books have come to an end. These explain our decision to create a books and book quotes search engine, aiming to facilitate moments when a reader wants to revisit a book or a particularly captivating quote, but cannot remember the exact book title or the quote that they are looking for.

Like its name suggests, our search engine allows a user to not only search books, but also the quotes of books. The user needs to first indicate what search they would like to be performed; book quotes search is the default option, but book search can be toggled with a button on the website. For both books and quotes search, the user should type parts of the book title or quote which they remember into the search bar. The full book title with the book's information (such as a brief description of the content) will be returned if they were looking for a book. The quote, together with its corresponding book, as well as the paragraph the quote belongs to, are returned if quote search were performed. Note that there are two possible ways of searching for quotes. One takes the user's search terms and return quotes which contain these terms, with complete disregard for where they are located in the quote. The other, slightly more complicated, method would take the user's search terms as an exact phrase and find the quotes which contain this phrase. We use the TF-IDF algorithm to rank the results of our book search, and a BM-25 scoring system for the simpler version of our quote search. The justification for those choices of the ranking algorithms are given in subsequent sections.

Beyond the basic features of our search engine, we have also included a variety of filtering options. This allows a user to search with a particular book title, author, genre, a date range (with respect to the publication date) and ratings. Finally, we have a spellchecker feature which corrects the user's query in the search bar if the user wishes to accept the recommended correction; this is to aid in querying the correct terms.

We are using a public books dataset to provide the books and quotes for our search engine. We filtered this dataset down to 24,985 books and 40,999,867 quotes. We have built two MongoDB collections - `books`, `quotes`, and two inverted indexes: `invertedIndex` and `bookInvertedIndex`. The first index is indexed on the terms that appear in quote documents, and the second index is indexed on the terms that appear in book titles. Those two indexes assist in the querying of books, quotes and phrased.

2 Overall Design

Our system can be divided into two parts: the frontend and the backend. The frontend consists of the user interface, where a user can type a search query and indicate their preferences for the search, while the backend is where interactions with the database take place and relevant searches are performed. The frontend is compiled into a static file, which is hosted on Surge, and the backend lives as a server on a DigitalOcean droplet. A diagram depicting an overview of the system can be seen in Figure 1.

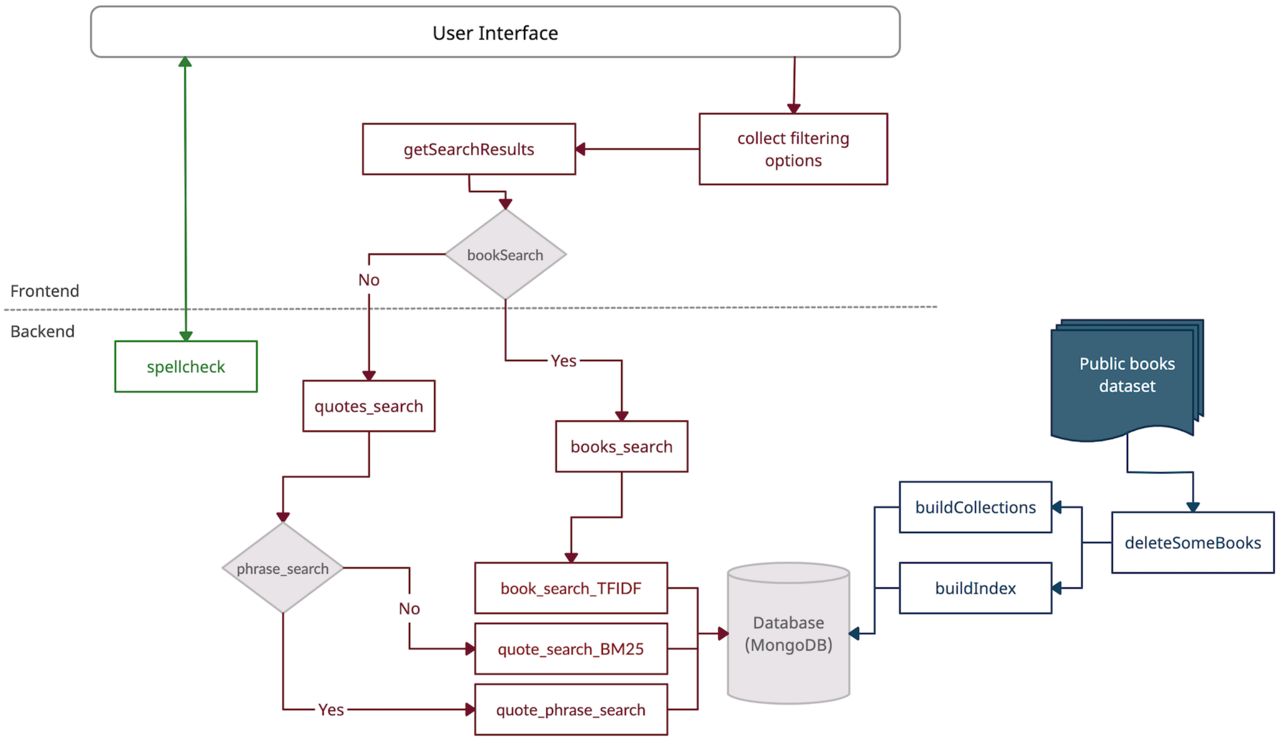

Figure 1: Overview of the Books and Book Quotes Search Engine

The blue part of the figure should first be mentioned, and it depicts how we collected a dataset of books and processed it to build the `quotes`, `books`, and the inverted index collections, which are all stored in our MongoDB database. Details of data collection and processing can be found in Section 3. The main component of the system is coloured brown in the figure, representing the search part of our system. The green-coloured component in the diagram is the spell-checker feature located in the backend that we implemented to help correct misspellings in search queries. While a user types their search query, the spell-checker dynamically evaluates it for corrections, and recommends the most likely correction if a spelling mistake is found.

The user is also able to add filters using the options provided on the user interface. When the user submits their search, the query is sent from the user interface, through an API call, to the backend, whereby documents are retrieved from the database, relevant searches are performed, the results ranked by a certain order, and are then returned back to the frontend where they are rendered on the user interface. Details about the retrieval techniques are in Section 4, a description of the backend can be found in Section 5, and that of the frontend is in Section 6.

3 Collection and Index Building

We were quite fortunate to have came across a public 196,640 .txt books dataset curated by shawwn and the-eye.eu. Note that the dataset does not contain many of the popular books or popular book series like the 'Harry Potter' book series or the 'Lord of the Rings' book series. The books were downloaded on the DigitalOcean droplet, taking 108 GB of disk space when unzipped. We decided to choose MongoDB to implement our collections and inverted index structures. In particular, MongoDB indexes use B-tree structures, which keep keys in sorted order for sequential traversing. This is particularly helpful to us with use cases such as that of range-queries.

3.1 Data Filtering

After running the `deleteSomeBooks.py` script, books meeting the following criteria were deleted:

- Non-English books
- Books containing no ISBN
- Empty books

This deletes 21,833 books, leaving us with 174,807 books. When running the `buildCollections.js` script (discussed in the following subsection) the following books are also deleted:

- Books with an ISBN not matching any of the Google Books API ISBNs.
- Books **NOT** in the following genres:

```
["fiction","biography & autobiography","juvenile fiction",
 "poetry","young adult fiction","philosophy",
 "young adult nonfiction","true crime","indic fiction (english)"]
```

This leaves us with only 24,985 books.

3.2 Collection Building

There are two collections (apart from the index; see the next subsection) that were built directly from the filtered dataset of the 24,985 books using the `buildCollections.js` script. The following two collections collectively occupy roughly 8 GB of disk space:

Books

The `books` collection has 24,985 documents of the following form:

```
{
    "_id" : 11,
    "title" : "The Malazan Book of the Fallen - Collection 1",
    "authors" : [ "Steven Erikson" ],
    "isbn-10" : "9781409092414",
    "isbn-13" : "1409092410",
    "categories" : [ "Fiction" ],
    "thumbnail" : "http://books.google.com/books/content?id=...",
    "publishedDate" : "2009-11-01",
    "previewLink" :"http://books.google.co.uk/books?id=...",
    "pageCount" : 1728,
    "averageRating" : "",
    "ratingsCount" : "",
    "terms_count" : 208721
}
```

Each `books` document contains metadata about the book obtained from the Google Books API.

Quotes

The `quotes` collection has 40,999,867 documents of the following form:

```
{
    "_id" : 12250,
    "book_id" : 11,
    "quote" : "She wondered just how far she could drive with the warning light on before the vehicle
        quit. She was miles away from the ranch, even farther from Trinidad, and she'd left the
        reservation behind nearly two hours ago."
}
```

The `buildCollections.js` script builds the `quotes` collection from the `.txt` book files as follows:

- Each quote is obtained by splitting on `\n\n`, making each quote a paragraph. This is to provide context for the reader, instead of just returning short sentences.
- A quote is only added to the quotes collection if it meets the following criteria:
  - It contains at least 10 alphanumeric characters.
  - It does NOT include any of the following words or characters:

```
        \itemsep0
["#","<",">","*","_",":","\n","@","copy right","copyright",
".com","www","copyediting","of fiction","e-book",
"all rights reserved","published by", "publisher","manuscript"
,"editor","coincidental","reproduce","special thank"]
```

  - It does NOT include the word "thank" more than 2 times (to avoid including thank-you notes that might have missed any of the filtering done above).

3.3 Index Building

In the process of building the inverted index structures, we would preprocess the terms of a given string as follows:

- Split terms on `[\s.,;'\"\(\)\[\]]`
  - We split in this way (instead of simply on `\w+`, for example) to account for punctuation between terms when doing phrase search.
  - For example, the sentence "I went home, school was boring", should not be a match for the phrase search of "home school".
- For each term, we then:
  - case-fold,
  - eliminate if it's a stop word,
  - and finally stem using Porter Stemmer.

The first inverted index structure we have is the `bookInvertedIndex` collection, which contains 11,596 documents and occupies 1.8 MB of disk space. It represents an inverted heirarchical index that maps each term to the book titles it appears in. Each document takes the following form:

```
{
  "_id": ObjectId("605de3fc024e8e9500825c55"),
  "term": "special",
  "term_freq": 23,
  "books": [
    {
      "_id": 1703,
      "title_len": 3,
      "term_freq_in_book_title": 1,
      "pos": [
        1
      ]
    },
    .
    .
}
```

Here, `term_freq` corresponds to the frequency of the term in all the book titles, `title_len` corresponds to the number of *all terms* in the title, `term_freq_in_book_title` corresponds to the frequency of the term in this particular book title and `pos` contains its positions with respect to all the terms of the title.

The second inverted structure is the `invertedIndex` collection, which contains 8,247,755 documents and occupies 19.3 GB of disk space. It represents an inverted heirarchical index that maps each term to the books it appears in to the quotes that appear in each book to the positions of the term in each such quote. Each document takes the following form:

```
{
        "_id": ObjectId("6040b3b44600b73f8ca43cde"),
        "term":"conquer",
        "term_freq": 22,
        "books":[
                {
                  "_id":11,
                  "term_freq_in_book":13,
                  "quotes": [
                    {
                        "_id":14576,
                        "len":45,
                        "pos":[16]
                    },
                    .
                    .
                    ]
                 },
                 .
                 .
                ]
}
```

Here, `term_freq` corresponds to the frequency of the term in all quotes nested in this document (not all quotes in the filtered dataset), `term_freq_in_book` corresponds to the frequency of the term in this particular `quotes` array, `len` corresponds to the number of *all terms* in this referenced quote and `pos` corresponds to the positions of the term with respect to all the terms of the quote.

The droplet used to build the index has 160 GB of disk space and 8 GB of memory. The following are the challenges presented to us by the limitations of our resources:

1. Creating a document for each term was not possible due to the MongoDB limit of 16MB per document.
2. Iterating through the quotes, creating the `invertedIndex` documents and writing them to MongoDB (or updating already existing MongoDB documents) with them was not possible. This is because performing an insert or update operation multiple times for each of the 40,999,867 quotes would have roughly taken 28 days in the best case.

We have catered for both of those challenges in the process of building the `invertedIndex` collection. The process is two-fold:

Stage 1: Make the Documents and Write them to Disk

The first stage is executed by the `db/index_building/py/buildIndex.py` script. As we cannot build the index whilst scanning through batches of the quotes, the following approach was adopted:

1. Load in memory 1000 `quotes` documents at a time.
2. Use the those quotes to build a portion of the index (in a `currIndexPortion = dict()` variable).
    - Preprocess each term by case-folding, removing if a stop word or if it is not an alphanumeric and then stem.
    - Update the `currIndexPortion`.
3. Every 205 iterations (205,000 quotes), we write the `currIndexPortion` documents to disk.
4. Garbage collect and clear variables, then return to step 1.

This divides the whole index into 200 disk files or batches. This means that each term can have a maximum of 200 documents. As quotes here are paragraphs, splitting the index into 200 files on disk was the best we can do given the memory limitations. Any lesser split and the script would crash because `currIndexPortion` was getting too large.

Stage 2: Load Index Disk Batches; Write them to Mongo

The second stage is executed by the `db/index_building/py/writeIndexToMongo.py` script.

1. Load each of the 200 index batch files.
2. Use the `insert_many` MongoDB operation to write the documents of each file.
3. Garbage collect and clear variables, then return to step 1.

Notes

Notice that you would find the equivalent of the previous two scripts written in Javascript inside `db/index_building/js/`. These scripts failed to create the index due to always running out of memory due to the lack of any control given to the programmer on the process of garbage collection. The Python scripts would similarily run out of memory if we fail to include `.clear()` or `gc.collect()` for clearing used memory.

In retrospect, we think if we had started developing the API and retrieval skeletons first and built the index to fit those needs second, it would have been a much more efficient arrangement. However, given the uncertainty of the time it would take to build the `MongoDB` collections we built the indexes in a predictive fashion instead of building it to fulfil certain needs.

4 Retrieval Techniques

Our search engine provides three types of searches: searching for a book, searching for a quote that considers each term separately, and searching for a quote as a phrase.

4.1 Book Search

This feature allows users to retrieve books with book titles that match their given terms to an extent. The terms are preprocessed, and used to rank book titles based on the TF-IDF score. Note that the returned results do not necessarily contain all the given search terms, but a greedy approach explained below is used to decide what is to be returned. To execute a book search, the user must toggle the book search switch. Furthermore, a user is able to customise their search, with options in the frontend leading to the filtering of certain authors, genres etc. when returning the books found.

As mentioned, the returned book titles are ranked based on the TF-IDF score. This score measures how relevant a word is to a document in a collection of documents by multiplying two metrics: how many times a word appears in a document (term frequency), and the inverse document frequency of the word across a set of documents.

The equation we used to calculate the TF-IDF score of a term is:

$$TF - IDF(tf, N_t, df, N_b) = \frac{tf}{N_t} \cdot \ln(\frac{N_b}{df}) \tag{1}$$

where tf is the term's number of appearances in the book's title (term frequency), N_t is the total number of terms in the book's title, df is the number of book titles that contain the term, and N_b is the total number of book titles in our collection (in our case, it is 24,985).

When a user searches for a book title using a sequence of terms, those terms will be individually used to obtain the book titles containing each term (if the user has included any options in their search, these will be accounted for and a filtered set of book titles will be used instead). To retrieve the book titles per term, the book's inverted index is queried which returns the book objects that their title contains the requested term. The score each book title, the necessary information such as term frequency and document frequency are computed, and finally the TF-IDF score is calculated for each of the retrieved document. For each term, a dictionary is kept that maintains the score of the retrieved book title documents. A time limit is also used to ensure the best user experience. When the time limit is exceed, only the intermediate documents gathered will be considered.

Since it is possible that not all requested terms appear in any book title we have, we follow a greedy approach to decide which book title to consider. The first step is to identify if there are book titles that contain all the given terms. If that holds, then those documents are returned. If that is not the case, then we disregard one of the given terms and repeat the above process. The worst case scenario, is when only one term is eventually used to retrieve book titles.

The way we disregard a term is the following: We firstly identify, for each term, what is the score of their top retrieved document. Each term is associated with this value and the term with the lowest score is disregarded in the next iteration. Once a set of book title documents are common among the term list, their scores are summed and used for sorting. In the end, the top 100 books, ranked by their TF-IDF scores, are returned to the user.

4.2 Quote Search

This feature is a simple quote search, where the user's text input is split into terms to be processed individually, and quotes that contain all the *preprocessed* (excluding stop words) terms of the search query will be returned. Quotes returned are ranked by the BM25 scoring algorithm, which is a bag-of-words retrieval algorithm that ranks a set of documents based on the query terms' appearance in each document.

We decided to use BM25 for this particular search feature since it goes beyond TF-IDF to account for the document's length and term frequency saturation[1], making it more suitable for searching quotes or paragraphs instead of book titles, for instance. The equation for calculating the BM25 score is:

$$BM25(D,Q) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{f(q_i,D) \cdot (k_1+1)}{f(q_i,D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{avgdl})} \tag{2}$$

where $f(q_i, D)$ is q_i's term frequency in the document D, $|D|$ is the length of the document D in words, and *avgdl* is the average document length in the text collection from which documents are drawn. k_1 and b are free parameters, usually chosen, in absence of an advanced optimization, as $k_1 \in [1.2, 2.0]$ and $b = 0.75$. IDF(q_i) is the IDF (inverse document frequency) weight of the query term q_i, and can be calculated as following:

$$IDF(q_i) = \ln\left(\frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} + 1\right) \tag{3}$$

where N is the total number of documents in the collection, and $n(q_i)$ is the number of documents containing q_i.

The routine to calculate the score of each quote document is slightly different from the one used in book search. In this routine, we had to take into consideration that each term could have up to 200 `invertedIndex` entries, due to memory constraints and the large number of quotes that a frequent term could appear in. For that reason, the inverted index documents are retrieved in batches to allow for efficient iterations of both levels.

Following an iterative approach, for each `books` document retrieved, we iterate over its associated `quotes` documents to calculate their scores. Using the `len` and `pos` attributes of the `quotes` document we calculate the document length and term frequency which are used in the BM25 score. We use directly the `term_freq` attribute of the inverted index entry to calculate the number of documents containing the particular term.

Similar to book search, for each term, a dictionary is kept that maintains the score of the retrieved `quotes` documents. After all terms have been iterated, that dictionary is used to gather only the common `quotes` documents between all the search terms given as opposed to book search where a greedy approach is followed. The scores of the final quote documents are summed and used for sorting. Note that, just like booksearch, a time limit exists to bound the execution time of a quote search. In the end, the top 100 quotes, ranked by their BM25 score, are returned. In case no quote contains all the search terms, an empty result list is returned.

4.3 Phrase Search

This particular feature allows the user's quote to be processed as a phrase. Users are required to wrap their search quote in quotation marks (e.g. "this is my phrase") for the quote phrase search to be triggered as long as 'Book search' is not toggled. Books that contain the exact phrase being searched will be returned with the phrase's context. Although there is no default order of the results being returned, users are able to toggle a ranking based on the number of ratings, the average rating, the number of pages, and the title of the book containing the search phrase.

The phrase search retrieval takes longer time as compared to the other two searches due to the nested nature of the `invertedIndex`. A brute force approach would have been to take the search phrase, retrieve the quote

[1]http://www.kmwllc.com/index.php/2020/03/20/understanding-tf-idf-and-bm25/

objects nested in the inverted index for each non-stop term in the phrase, compute the distance between each pair of such terms, eliminate quotes that don't have such an arrangement, and finally determine if the stop words in the phrase are in their correct respective positions. Clearly, this is very inefficient as it would have to retrieve all quotes for each non-stop term and iterate over them yet again, which would result in extremely long retrieval times.

Thus, we came up with the following algorithm as a superior alternative:

1. Pick the first instance of a non-stop term in the search phrase and call it `root`.
2. Compute the distances between `root` and any other term instance in the phrase. Store those in a dictionary, `distancesDict`.
   - For example, a phrase search of "I love you", would result in `root = "love"`, and
   - it would result in `distancesDict = {-1 : "i", +1: "you"}`
3. Use the inverted index to retrieve the documents of `root`, `rootDocs`, and use it to make a set of the quote IDs of the quotes containing `root`, `quoteIDsSet`, and book IDs of those quotes, `bookIDsSet`.
4. For each other non-stop term in the phrase search, obtain a list for its quote IDs and book IDs from the inverted index and use them for the following elimination.
   - Eliminate quote IDs in the `quoteIDsSet` that do not intersect with the quote IDs for the current term.
   - Eliminate book IDs in the `bookIDsSet` that do not intersect with the book IDs for the current term.
5. Obtain the quote objects corresponding to the quote IDs in `quoteIDsSet`, *limiting the number of returned objects to 25,000.*
6. Make a dictionary, `quoteIDToObject`, that maps each quote ID to its quote object. This is for faster retrieval in the next step.
7. Go through each document in `rootDocs` (obtained in step 3).
   (a) For each nested quote object in `rootDocs`, verify using `quoteIDToObject` that it contains all non-stop terms of the search phrase.
   (b) If it does, obtain the positions of `root` in the current quote.
   (c) Using the `distancesDict`, compute each offset in the current quote to see if all terms are where we expect. If the offsets match, we add this quote's ID to a result set.

The concession we make in the italicised part in step 5 is there for two reasons: memory and retrieval time. On the one hand, an extremely long phrase search can lead to a `quoteIDToObject` that is too large for memory, therefore it needs to be bounded. Likewise, we also reduce the number of documents retrieved per term to 60 to avoid exhausting memory. On the other hand, looking for a really common phrase can retrieve a large number documents. Even if the memory is able to handle that many results, it will take a significant amount of time for the DB to retrieve a large batch of objects. Therefore, although our compromise might result in some phrases not being retrieved, it will result in fast retrievals for others.

5 Backend

We built our server with Python and our API using the Flask framework. An alternative was to use Node.js for the server since it could directly integrate with the frontend; however, we eventually decided to use Python, since it was a language our team felt more comfortable coding the retrieval techniques in.

Our API receives the queries from the frontend, and is in charge of pre-processing the terms in the queries before forwarding them to the server, as well as cleaning and re-structuring results before returning them to the frontend. Our spell-checker feature is also located in the API, and this is where the spelling of a user's search while they are typing is checked; detection of a mistake would lead to the correct spelling being suggested.

The server of our project is responsible for fetching documents from MongoDB, performing the searches and ranking (described in Section 4), and filtering the results to return as per the user's options indicated on the frontend. Furthermore, on rendering of the frontend, the server will also return a list of authors and book titles, and this supports the frontend's feature of allowing a search that is filtered based on authors or books. We chose this additional feature to provide suggestions for the user when it comes to authors and books instead of allowing them to type a specific name (like the other filtering options), because we found that names could be represented differently but refer to the same book, or person, and so having choices provided more accuracy for our search.

6 Frontend

The GUI of the project was built using React JS. We chose to use React as it was extremely proficient and modular in handling the view layer of web applications. Furthermore, one member of our team was already familiar with the framework which solidified the decision. Our website is built as a one-page view and can be logically divided into two main components, the search input area and the result area.

The search input area includes the main search bar, additional search features and action buttons. A user can enter a quote or a phrase on the main search bar and submit their request to retrieve quotes that match their request. Note that a phrase search will be triggered only if the user's input is within double quotation marks. To further the control the user has over their search, a user can provide any other additional search attributes like book title, author name, book genre, publication year range and minimum rating. Those additional attributes will be used to filter the returned results. Lastly, a user can also choose to perform a book search. By toggling the book search switch, the returned results will not contain the matched quote but rather the highest ranked books based on the terms given.

On a successful search, the results are displayed as cards below the search input area. In case of a quote or phrase search, each of the cards contains the book title and the quote that matched the given search, highlighting within the quote the user search terms or phrase. The card also contains additional information about the associated book like book category, number of pages, ISBN, etc., which can be viewed by expanding the card. The cards also provide the user with the option to be redirected to the Google Books' page of that particular book. In the case of book search, the cards do not include any quote in their body but rather include all the book information in their main body.

To further enhance user experience, we included some additional functionality on our webpage. Firstly, the search results are displayed in a paginated manner, with 10 cards displayed per page and a pagination mechanism to allow the user to navigate to subsequent pages. Furthermore, the user is given the option to sort the results by their rating count, average rating, alphabetically by the book title or even by the page count. Additionally, we use a spell checking mechanism on the main search bar that will automatically prompt the user with a correct version of their input in case a spelling error is detected. Lastly, we enhanced the two advanced search inputs, namely "author" and "book title" to include all the possible authors or book titles of our collection as drop down options. That allows the user to see the possible options for those inputs as the dropdown list is dynamically updated with the matching options based on the user input, just like an auto suggestion mechanism.

7 Evaluation and Future Improvements

To evaluate our system and ensure that we provide the best user experience, we devoted a good amount of effort on optimising and enhancing the ranking searches. Table 1 demonstrates some example queries and their run times for every available feature in our system. Note that in order to diversify our evaluation, the evaluated queries included a variety of terms, from common to rare, to accommodate for all edge cases.

Evaluation		
Feature	**Query**	**Retrieval Time** (seconds)
Quote search	fall in love	1.39
Quote search	summer heat wave	1.04
Quote search	death is around	2.01
Book search	star trek	0.87
Book search	life like love	0.6
Book search	things to know about	0.64
Phrase search	"she hate me"	1.96
Phrase search	"he said"	26.18
Phrase search	"the love of my life"	5.95

Table 1: Query examples and retrieval time

The project has left us hungry to go back and do a lot of improvements; these are some of the improvements we would have liked to make.

- Extending book search to allow a user to search for a book with a brief description of its contents. This can be achieved by building an index on the book descriptions.
- Query auto-completion, which we believe can be achieved with a machine learning model.
- A recommendation system, which will recommend similar books or quotes based on the user's search.
- Splitting quotes as sentences, instead of paragraphs. This might improve our retrieval performance in terms of relevance.
- Reshape the `invertedIndex` to querying and performance needs instead of building it in a predictive fashion.
- Finally, as we mentioned before, our dataset does not have many popular book series (Harry Potter, Lord of the Rings, etc.), and we would like to include these books to not only provide a wider variety of books, but also cater more to public demand.

Lastly, we learnt a harsh lesson towards the end of the project by accidentally leaving the MongoDB port open on the droplet to any IPs, which lead to our database being wiped out and the attacker asking for BitCoin in return of our database dump. This happened as we were experimenting with Heroku as an option to host the frontend (and potentially the backend, leaving the droplet as a remote DB server). We temporarily allowed the firewall on the droplet to accept requests from any IP address, since a Heroku app does not have an IP range or a static IP address to whitelist it on the droplet. It took only 24 hours of forgetting to bring the firewall back up to see the attacker's message as a lone entry in our DB.

This resulted in the team losing time as we had to re-download our dataset, filter it, and rebuild our collection and indexes from scratch. We have, however, learnt a great deal about firewalls from such an event and used this event moving forward to make the firewall of our droplet and access to our database extremely restricted.

8 Individual Contributions

s1603986 - Yussef

Throughout the semester, I have contributed to the project in the following ways:

- Took part in the hunt for a dataset, helped set up the droplet and helped analyse the dataset.
- Wrote the scripts that filter the dataset, helped write the scripts that build the DB collections, and wrote the scripts that build the indexes.
- Wrote the algorithm for phrase search.
- Helped in coming up with the implementation ideas for the other search features.
- Designed and implemented the result cards, along with their features (e.g. quote highlighting and show-more/show-less features).
- Helped implement the feature that allows users to sort their results.
- Implemented the spell-checker feature.
- Took charge of the deployment process; including setting up the droplet as backend server, frontend hosting and the communication between the two.

s1735833 - Erodotos

My contributions in the project are the following:

- Analysed and designed a parsing mechanism for our book collection. Main challenge was identifying patterns and common structures among books to utilise for efficient parsing.
- Helped write the scripts that parsed and filtered the dataset
- Designed, created and updated the web application besides the card components.
- Implemented advance functionalities like auto completion on filter options, sorting and pagination of results.
- Handled the HTTP POST request calls to backend endpoints.
- Along with Muyao, I set up the Flask server, wrote the MongoDB script to interact with the database, designed and implemented the quote search.
- Implemented the book search routine. Main challenge was optimising the ranking routines in terms of retrieval time.

s1757868 - Muyao

I have worked on the following parts of the project:

- Researched parsing methods and came up with experimental functions to parse the books.
- Set up the API in Flask, and wrote code for the interaction with our MongoDB database, alongside Erodotos.
- Wrote an earlier version of phrase search (which was not used in the end).
- Helped with optimising quote search to return results in a shorter period of time.
- Wrote the introductory and overview sections of the report, and created a system overview diagram.

s1763241 - Maria

List of the things I have worked on during the semester:

- Manually analyzed the dataset, looked for strange patterns, phrases, or anything that would cause problems with preprocessing
- Calculated average sizes of book parts before and after the main content of books
- Helped to set up our digital ocean droplet to be the server and made sure everyone is able to connect to the db

- Helped with the advanced filtering in the backend so that we can filter the books based on options chosen from the advanced search
- Tested all searches, spent time on finding bugs and strange behaviours of our website, worked on the evaluation