
TTDS Group Project - Movies & TV Shows Quotes Search Engine

S1965737, S1837964, S1958674, S1885912, S1964892 & S1452787

Abstract

The Movies and TV Shows Quotes Search Engine is a website, that finds quotes and quote details for popular movies and TV shows. Given a query, it returns the most relevant quotes, along with the corresponding movie details. The collection of documents contains 77,584,425 quotes from 218,000 movies and TV shows. It uses a multi-level inverted index occupying 21.7GB on disk. The ranking algorithm uses BM25 and logarithmic popularity weighting for phrase search. Instead of quotes, the website can also perform a second search for full movies using weighted TFIDF algorithm. It also contains advanced search features, such as filtering by year, actors and keywords, and it includes genre filtering and query completion. The website can be reached from this link: <http://167.71.139.222>.

1. Introduction

Motion picture, such as movies and TV shows, is one of the most influential media type in the current and last century. It is not only consumed a lot, with many people watching movies and shows for several hours a day, but is also discussed in other forms of media and private conversations. Movie quotes are now commonly referred to in other media, such as YouTube or Newspapers, or in other movies. With the rise of meme-culture, the use of movie quotes as a means of communication has become ubiquitous. This inspired us to create a search engine for movie quotes.

Our website is targeted on users, who remember parts of a quote and want to find out the details about the whole quote and the movie. The search engine returns the full quote, the movie title, the character who said the quote, the time stamp withing the movie, and other movie details. Advanced search features, such as the specification of the movie title, an actor, a range of years and movie keywords help refine the search.

The engine can additionally be used as means to find movies that dis-proportionally use certain words, or find all movies that reference a common saying, such as 'Carpe Diem'. For this, we build the movie search, which returns results for movies, instead of quotes, and phrase search for quotes.

To be able to retrieve the most relevant results, a large number of quotes is necessary. However, it is not trivial to collect this number of quotes using web crawling. There are some quotes for popular movies on some websites such as IMDB, but they are not enough to generate a search engine. Thus, we decided to collect the subtitles for the movies/TV shows and extract the quotes from them. We obtained around 120,000 subtitle files from the OpenSubtitles.org website. Over 75 million sentences were extracted from the subtitle files and stored in a Database (DB) collection. iMDB database was also used for showing detailed information about the movies related to search query. In our design, each sentence (quote)

is a document in the inverted index, since our goal is to find the most relevant quotes to the search query. For the movie search support, we created a multi-level positional inverted index. The first level maps the distinct terms to the movies while the second level maps the movies to the quotes. Since we do not have enough memory to load the whole inverted index at once, we decided to use [MongoDB](#) which is based on B-Tree data structure to store the inverted index.

For the quote search, we use the BM25 ranking algorithm coupled with popularity weighing, while for the movie search we use TFIDF. The BM25 is a close relative of TFIDF, but accounts for documents lengths and gives more accurate results for short documents ([Lixin Xu et al., 2014](#)). For the phrase search, the results are sorted by the popularity.

After we made sure that all the basic features of the search engine are functions very well, we started researching and implementing some advanced features which will be discussed in a later section. We decided to implement the Query Completion feature, which uses a language model to predict the next word given the previous one typed by the user. The model was trained the model using a Recurrent Neural Network (RNN) and used a dataset of the 1000 most popular movies of all time.

This report is organised as follows. In Section 2 we describe the system architecture and the technologies used in the project. In Section 3 we explain the methodology used for collecting and storing the subtitle files and the movies dataset. In Sections 4 and 5, we explain the pre-processing methods and the indexing algorithm of the quotes that we used in detail. We describe the BM25 algorithm in Section 6 and explain the advanced search functionalities implemented in the project in Section 7. The query completion model is explained in Section 8. The design of the API and the Graphical User Interface (GUI) are described in Sections 9 and 10. The project is evaluated and future work is discussed in section 11. The individual contribution of each team member is shown in Section 12.

2. System Design

We started to work by designing the whole system architecture and deciding which modules we need to develop. Then we modified our system design to improve efficiency of the system and get faster results. The final architecture design is shown in figure 1. The system can be split in two parts, *server side* and *client side*. Server side modules process the search requests that can be received from multiple clients and return the results. The client side consists of the GUI and its parts for connecting with the server side.

First of all, we need a data collector module providing all of the data use in the project such as subtitle files and iMDB data. This module collects all of the data and stores them in our system storage space in a customized directory format specified in section 3. After collecting the files we need, they

should be processed and the inverted index should be generated. Hierarchical Indexer module processes the collected data and produces the hierarchical positional inverted index as described in sections 4 and 5. The GUI takes the user inputs and runs the search queries. Query completion module gives some suggestions to users while they are writing a search input. When the users finish writing and run the search algorithm, the GUI parses the query and sends it to the server. Query handler module of the API (described in detail in 9) handles the request and triggers the result finder. The documents retrieved from the inverted index are sorted by ranking results in the ranker module and stored in a session cache. Then, page loader in the GUI requests manages the results to be shown in the GUI. When the user wants to go to next page, it requests the results for the next page. In this way the computing load stays in the server and users have a fluent experience. If the user wants to see the details about any of the result quote, GUI requests and shows the details to the user.

To develop the system described above, we preferred to use MongoDB as the database and Python language for the back-end application and scripts. Our API is developed in Flask framework and the frontend is developed in ReactJS. For the deployment environment, we decided to use a cloud server (Droplet) from Digital Oceans to host the DB, backend/frontend applications and the API. For the development environment, we used github for the version control and Trello for the project and requirement management. We also performed an agile development methodology and reviewed the codes of each other during the development process.

3. Data Collection and Storage

To develop the search engine, we have collected a large dataset of movie information and subtitles files. Movie information was taken from [iMDB](#), and subtitles files were downloaded from [OpenSubtitles](#). iMDB provides some [public datasets](#) containing various information associated with every title – a movie, a TV series or its episode. However, for our use-case, we required more specific information that is only available in the HTML pages of iMDB website.

3.1. iMDB Title Information

From the iMDB public datasets, we used *title.ratings.tsv.gz*, which contains the average rating and number of votes for each title on iMDB. We sorted the titles first by number of ratings, then by average rating, both in descending order. Then from the resulting subset of titles, we took iMDB IDs of all titles that have been rated by at least 100 people, of which there were around 218,000 in total. This seemed to include most if not all titles we have ever heard of.

Then, we used an open-source framework Scrapy [Scrapy](#) to extract all title information from the iMDB website. A “spider” was developed to load a file of iMDB IDs and for each ID, visit 4 pages:

- [imdb.com/title/\[ID\]](#) – contains basic information such as title name, description, year, genres and a thumbnail picture.
- [imdb.com/title/\[ID\]/fullcredits](#) – contains cast information: actor names and corresponding character names.
- [imdb.com/title/\[ID\]/keywords](#) – contains user-added key-

words for the movie. For example, movie Shawshank Redemption has a keyword “prison”. We had plans to use these keywords to enhance movie search, but the idea was discarded later, because for the most part, the same keywords already appear in the sentences occurring in the movies. For example, term “prison” appears in 21 spoken sentences throughout the movie Shawshank Redemption and in fact, movie search query “prison” returns Shawshank Redemption as the Top 1 result.

- [imdb.com/title/\[ID\]/quotes](#) – contains user-added noteworthy quotes spoken in the movie. Each quote consists of a character name and one or more sentences. These were matched to sentences processed from subtitles to include character names for sentences wherever possible. Even though the quotes from iMDB were quite limited, around 1.2 M out of 77+ M sentences have had the character name added.

Using XPath selectors, all title information was extracted from HTML pages and written to a file with the JSON lines (JSONL) format. The web-scraping script had to be run over several days, so additional methods were programmed, given an iMDB ID, to check what type of information we are missing. This way, the script could be stopped at any point and it would pick up from where it left off on the next run.

Finally, data for all movies was inserted into the *movies* collection of our MongoDB instance, with one exception. Quotes information was not written to MongoDB but to text files instead, following the folder structure of subtitle files described in subsection 3.2. For example, all quotes of title with ID “tt1234567” would be stored in file [quotes/1/2/3/tt1234567.txt](#). A quotes file consists of one or more lines containing a character name followed by “: ” and the quote spoken.

3.2. Subtitles Files

The main purpose of this project was to enable users to search in words spoken in movies. For that, we required to acquire subtitles files for as many titles as possible. At the time of writing, [OpenSubtitles.org](#) has over five million subtitles files in many languages. We decided to base our search engine only on English subtitles, as the majority of well-known movies are English.

OpenSubtitles offers an API for searching subtitles files and downloading them. However, the API comes with limits: 200 downloads per day for a regular user and 1000 for VIP (which requires a yearly subscription). A script was written to load the file of iMDB IDs (with at least 100 ratings, as described in subsection 3.1) and for each ID that we do not yet have subtitles for, send an OpenSubtitles API request to find all English subtitles matching the iMDB ID, then from the results, choose the subtitles file that has the highest downloads count (we assume it to be the best quality measure), and download it. A logging mechanism was set up to permanently keep track of iMDB IDs for which there are no English subtitles available, as well as iMDB IDs for which we have downloaded subtitles already, in order to not repeat any requests.

Due to the API limits, it would only have been possible to download at most 60,000 subtitles files (VIP plan) with the time constraints we had. To help with acquiring the subtitles, we reached out to OpenSubtitles team explaining our situation and asking to increase our limits. In response, they provided us a link to download a large export of subtitles to be used

only for the purpose of this project (thank you, OpenSubtitles team). The export was created on 18th Oct 2019. It contained a nested folder structure filled with g-zipped subtitles, named as OpenSubtitles IDs used by them internally to organise the subtitles. The export also included a text file containing metadata for each OpenSubtitles ID, including corresponding iMDB ID, language and downloads count. These were used to find the file names of most downloaded English subtitles for movies we have. The files were then unzipped and reorganised into our own nested folder structure. For example, subtitles for movie with ID “tt1234567” would be stored at [subtitles/1/2/3/tt1234567.srt](#).

Around 110,000 subtitles files were acquired from the export, and around 10,000 files were downloaded using the custom API script (over several days), for a total of 121,958 subtitles files.

The subtitles were then processed (as will be explained in section 4) and every sentence was inserted into the *sentences* collection of our MongoDB instance. Each sentence document contains a unique ‘_id’ (a 32-bit integer ranging from 0 to 77 M+), a ‘movie_id’ (iMDB ID), a ‘sentence’ string and some sentences also contain a ‘character’ name of the person who spoke the sentence in the movie.

3.3. Database

We have decided to use MongoDB for data storage due to its natural compatibility with JSON format used by React (the front-end framework we used), as well as support for documents with nested structure (used for our inverted index, explained in section 5 and shown in figure 2).

The *sentences*, *inverted_index* and *movies* collections can be accessed via a MongoDB module from any other function. Within this module, one can retrieve: the inverted index entries for a term on quote or movie level, the movies fitting to an advanced search, the full quotes for a list of quote IDs in the order requested, and the movie details for a list of movie IDs. The module uses *MongoClient* to connect to the database, and there are database indexes created for fields that we query by in order to increase the speed of queries. Collections *movies* and *sentences* are indexed by movie or sentence IDs, respectively. *sentences* are further indexed by *movie_id* to allow efficient access of all sentences for a given movie (for ad-hoc operations). *inverted_index* is indexed by term for efficiently retrieving all inverted index entries by a query term, and also indexed by the first movie _id in the movies list shown in listing 5 to support linear merge for quote phrase search described in section 6.2.

4. Preprocessing

The subtitle files are not the documents we want to search in this project. The goal is to find a phrase said by a character in a movie/TV show. As a first step, we parsed the subtitle files and extracted all the sentences out of them. Each sentence is considered a document and will be retrieved as the quote that the user wants. More than 75 million sentences were extracted and stored in the database. The database structure of a document is shown in Listing 1. Then, the sentences were tokenized, punctuation symbols and stop words were removed and the tokens were stemmed before they were indexed. To provide more functionalities to the website, we stored not only the quote sentence, but also the name of the character who

said it. The subtitle files do not contain character information but the iMDB data has this information for some popular quotes, as described in section 3.1. We enriched some of the parsed sentences with the character information using that. Even though the character is not available for all of the quotes, it is available for the most popular ones. Occasionally of the user search results will have a character name, since many users will mostly search for popular quotes, which can be also handled by our system.

```
{
  "_id": 8777416,
  "sentence": "I am your father.",
  "time_ms": 6403840,
  "movie_id": "tt0080684",
  "character": "Darth Vader"
}
```

Listing 1. Example quote record in the sentence collection of DB.

4.1. Parsing Subtitles

There are two widely used subtitle formats which are SubRip (.srt) and MicroDVD (.sub), and our subtitle data collection contains both of them. Thus, we developed two different parsers. (.srt) files contain formatted lines of text groups separated by an empty line. Each group represents a subtitle to be shown for a particular time range. For each group, the first line contains the identifier number, the second line contains the time range, while the rest of the lines contain the subtitle content. There are two challenges to parse this format for our purposes. First one, a sentence might be split into multiple groups and a group might have multiple sentences said by different characters. The other problem is that the (.srt) format supports HTML tags to modify the text appearance on the screen. Thus, we developed a (.srt) parser which merges the subtitle groups unless it detects an end of sentence while removing HTML tags from the subtitle texts. An example for a (.srt) format is shown in listing 2. Two groups contain one single expression and an HTML tag is also used. Our script parses it and generates a database record as shown in listing 3. (_id) is generated incrementally by the database, (time_ms) is the start time of the expression in milliseconds and (movie_id) is the iMDB identifier of the movie, which is the file name in our case. (character) is the name of the character who said the quote. How we retrieve the character names are explained in section 4.2.

```
46
00:04:09,912 --> 00:04:11,872
Since I am innocent of this crime...

47
00:04:12,073 --> 00:04:16,512
...I find it decidedly inconvenient
that the gun was <b>never</b> found.
```

Listing 2. Example subtitle text in SubRip format.

MicroDVD is the other popular subtitle format. In this format, each line represents a subtitle. Expressions may be separated by "|" character to be shown in multiple lines on the screen. The lines have some control codes before the expressions, written between "{" and "}" characters. Only first two control codes, the start frame and end frame numbers, are important for our purpose. This format uses frame numbers to represent when the text to be shown, which is different from (.srt)

that uses standard time format. The (.sub) parser extracts the start frame and the expression for each line and calculates the time in milliseconds, by multiplying the start frame by $\frac{1000}{24}$ (assuming that the subtitles are encoded in 24fps). An expression may be split into multiple lines in (.sub) format as well. Thus, the same algorithm for merging the expressions is also used in this parser. The example (.srt) formatted subtitle in listing 2 is shown in (.sub) format in listing 4. Both of the parser scripts are developed in Python using standard regex library. Over 75 million sentences were produced and written into the database.

```
{
  "_id": 1,
  "sentence": "Since I am innocent of this
              crime I find it decidedly inconvenient
              that the gun was never found.",
  "time_ms": 249912,
  "movie_id": "tt0111161",
  "character": "Andy Dufresne"
}
```

Listing 3. The database record of the subtitle in SubRip format shown in listing 2, generated by the parser script.

```
{5997}{6044}Since I am innocent of this cr-
ime
{6049}{6156}{Y:i}I find it decidedly incon-
venient|that the gun was never found.
```

Listing 4. Example subtitle text in MicroDVD format.

4.2. Character-Quote Matching

The sentences/quote records in the database generated by parsing the subtitle files did not have character information. The content of the iMDB dataset was explained in section 3.1. We exported the quotes from the iMDB dataset in the same directory hierarchy as the subtitle files. Each line of these files has a character name and a sentence or a couple of sentences. Our goal here was to find these sentences in our quote records in the database and assign the character name for the found quote records. However there were some challenges when we tried doing that. The sentences in the subtitle files can have a missing punctuation or short form of the words. In addition, the quotes in the iMDB dataset are added by their users so there can be some missing words or differences in the iMDB quotes. Therefore, we could not just match the iMDB quotes with ours by looking at the text directly. Thus, we decided to use the "Cosine Similarity" to find the most similar iMDB quote for each of our quote records.

The Cosine Similarity is based on Euclidean dot product. To calculate it, sentences should be vectorized first. Then the cosine similarity score could be calculated by using the following formula for given S_1, S_2 sentence vectors.

$$\text{cosine similarity} = \frac{S_1 \cdot S_2}{\|S_1\| \|S_2\|} = \frac{\sum_{i=1}^n S_{1i} S_{2i}}{\sqrt{\sum_{i=1}^n S_{1i}^2} \sqrt{\sum_{i=1}^n S_{2i}^2}} \quad (1)$$

Cosine similarity function was implemented using formula 1 and it is used in the algorithm for finding the correct character names for each quote. The algorithm is shown in algorithm

1. The function iterates over all of the quote records in the database and for each quote it finds the iMDB quote file by the movie name. It finds the quote and character name having the largest cosine similarity score. 80% was defined the threshold, to prevent matching quotes with incorrect sentences. The character names are available for around 1.2 million quote records in total. Even though it is a small proportion of whole collection, they are the most popular quotes from the most popular movies. Thus the probability of seeing the character names in the search results is much more than their proportion in the collection.

Algorithm 1 Quote-character name matching algorithm.

```
foreach quote in DB do
  find iMDB quote file by movie name
  similaritymax ← 0.8
  foreach iMDB character-quote pair do
    similarity ← cosine similarity between the quotes
    if similarity > similaritymax then
      similaritymax ← similarity
      name = iMDB character
    end
  end
  assign name to the quote in DB
end
```

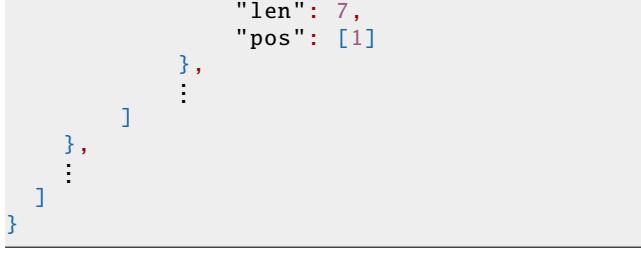
4.3. Tokenization, Text Normalization, Stemming and Stopwords Removal

The standard preprocessing algorithms were implemented to prepare the text and convert it the standard format after the parsing is done. First, case folding was applied, which converted all letters to lower case. After that, non-standard characters and non-alpha-numeric characters were removed. Then, all the text was tokenized. Since the stop words are very common in most of the quotes and they do not contribute much to the meaning of the quote, we decided to remove them. We used the standard list of stop words provided by [NLTK Library](#). Finally, the stemming was done using the PorterStemmer class in the NLTK library.

5. Indexing

Users are able to search a quote by typing the query, or by specifying a movie, a genre, or a time range. The details of the search possibilities are explained in section 7. To support these advanced search features, we decided to use a hierarchical positional inverted index. The first level of the index maps the terms to the movies while the second level maps the movies to the quotes. The structure of the inverted index shown in listing 5. we added document frequency information (doc_count) in both levels in addition to the standard inverted index structure to calculate ranking functions faster. We also added document lengths since it is needed in BM25 ranking algorithm.

```
{
  "_id": 1,
  "term": "father",
  "doc_count": 348007,
  "movies": [
    {
      "_id": "tt0002423",
      "doc_count": 1,
      "sentences": [
        {
          "_id": 292,
```



Listing 5. Structure of the hierarchical positional inverted index.

Since we have a huge number of documents, a single inverted index would not be feasible for our system since it has 2GB of memory. Thus, we decided to use B-Tree data structure for indexing. Instead of implementing a B-Tree algorithm ourselves or using a Python implementation of it, we preferred to store the index in [MongoDB](#) which already uses B-Tree for indexing. Maximum document size is limited to 16MB in MongoDB and popular terms in the index exceeds this limit (term "i" exists in around 20% of all sentences in movies, and the inverted index for this term takes more than 800MB of space). As a result, we divided the first level of the inverted index into smaller parts. In other words, there are multiple (up to 52) records for a term in the inverted index instead of one term having all of the index data.

6. Ranked Retrieval

The ranked retrieval can be split into three main functionalities: quote search, quote phrase search and movie search. In both quote search cases, the user inputs what he remembers of a quote, for example 'May the Force', and the most relevant quotes are then retrieved and displayed. In movie search, the user may input what he remembers of words spoken throughout the movie, for example 'Luke father', and the movies where these words appear most open, are retrieved and displayed (based on the TFIDF measure).

6.1. Quote Search

The quote search uses the BM25 algorithm and logarithmic popularity weighting. The algorithm uses bag-of-words, and ranks the quotes based on the query terms appearing in them. For the simple quote search, no weight is put on the proximity of the terms towards each other. The formula for the BM25 score is as follows:

$$BM25_{D,Q} = \sum_{i=1}^n IDF(q_i) \frac{f(q_i, D)(k+1)}{f(q_i, D) + k(1 - b + b \frac{|D|}{avgdl})} \quad (2)$$

with $S(D, Q)$ being the score per document and query, n the number of query terms, $IDF(g_i)$ the inverted document frequency of the query term, $f(q_i, D)$ the term frequency of the query term in the document, $|D|$ the document length, and $avgdl$ the average document length in the collection. k and b are free parameters. We set k to 1.2, b to 0.75, and $|D|$ is 4.82.

To increase the relevance of the query results, we decided to add popularity weighing to the BM25 score. Each score entry was multiplied by the logarithm to base 10 of the number of ratings the corresponding movie has on iMDB.

$$S(D, Q) = BM25_{D,Q} * \log_{10}(\#ratings) \quad (3)$$

The score is then entered into the Score Tracker, which keeps the top 100,000 results in memory.

The algorithm is implemented using multiple for-loops. Note that our inverted index has the structure `index_id` \rightarrow `movie_id` \rightarrow `sentence_id`. Due to memory constraints, the results for each term are split into 52 entries, and `index_id` is thus not unique. The entries are retrieved in batches of five. The quote search algorithm then loops through the entries in the batch, and withing each entry, through the movies. Since the DF is saved in the inverted index, it can be used directly, and does not need to be calculated. For each movie, the algorithm checks whether this movie confirms to the advanced search constraints (if any), and then calculates the IDF for each sentence in the movie. In the same loop, the resulting score is added to the Score Tracker, which keeps the maximum number of saved results at 100,000 (as explained in section 6.4.2).

By saving the score directly into the Score Tracker, the algorithm avoids using any additional memory. By looping only through relevant movies for the IDF, and not requiring further calculations for the DF, it is also time efficient. When advanced search is enabled, the algorithm only loops through movies that conform to the filter.

6.2. Quote Phrase Search

The quote phrase search is activated if the user's entered query is wrapped in double quotation marks (") and if the count of terms (after removing the stop words from the query) is at least 2. The algorithm performs a linear merge to find sentences containing the phrase (matching the query exactly), and returns the results ordered by the popularity of the movie (in descending order).

Linear merge requires that the documents in the inverted index are all sorted. We have created an inverted index with the following hierarchy: `index_id` \rightarrow `movie_id` \rightarrow `sentence_id` \rightarrow `position`. We were able to ensure a sorted order from the movie level downwards, but unfortunately, since the index entries had to be split and inserted as separate MongoDB documents of less than 16MB space, MongoDB was not able to preserve the natural ordering at the index level, meaning that a DB request for inverted index entries for a term may return entries in which individually the movies, sentences and positions are sorted but the entries themselves are unsorted. Because of this, a database index by the first movie `_id` had to be created for the `inverted_index` collection as described in section 3.3. When retrieving the inverted index entries for linear merge, the database query must sort the entries by the first movie `_id`, which unfortunately is slow for popular terms and because of this, the quote phrase search takes quite a bit longer to complete than the regular BM25 quote search. Still, the algorithm used is noteworthy and the details will be explained in the following paragraphs.

The algorithm initialises a list of multi-level iterators for each term. A multi-level iterator consists of: a [MongoDB Cursor](#) object for a sorted query to the inverted index collection, the current inverted index entry (which is set to `None` when the cursor for the term is exhausted) and indexes for the current movie, sentence and position (starting from 0). Once initialised, the algorithm runs a loop until at least one cursor is exhausted or the time limit is reached.

One iteration of the loop consists of the following steps:

- For every adjacent pair of iterators $iter_i, iter_{i+1}$, run method *catchup*($iter_{i+1}, iter_i$) that advances the iterator from the first argument until the (movie, sentence and position) entity pointed to by the iterator is “in front of” (greater than) the iterator from the second argument.
- Iterate over the list of iterators to check if a precise movie, sentence and positional sequence match has been found. If so, add the sentence to the results.
- Advance the first iterator $iter_1$ to approach the last iterator $iter_N$ but not “overtake” it.
- If time limit has been reached, return the results, ordered by movie popularity.

Further, the catch-up mechanism was made to make iterator comparisons on each level, meaning that, for example, one advancement of an iterator can jump to the next movie without jumping sentences or positions if the movie ID of the iterator is behind from the other iterator. Finally, after any advancement of the iterator, the algorithm checks if the advancement caused an exhaustion in the cursor. If that is the case, the results are immediately returned since it would not be impossible to match any other phrases precisely even if a single term has no more documents.

The algorithm is meant to run in $O(NM)$ time where N is the number of terms and M is the maximum number of documents for a term. However, due to the multi-level catch-up mechanism, many irrelevant documents are “jumped over” in practice, reducing the run-time significantly. If sorting the inverted index entries was not needed on every query, the run-time of quote phrase search is expected to be $\frac{2}{3}$ of the run-time of the BM25 algorithm, based on local experiments. At the moment, however, quote phrase search takes approximately twice as long as the BM25 quote search.

6.3. Movie Search

Sometimes, a user may not remember any particular quote of a movie but they remember the words that were repeated often throughout the whole movie. Also, a user may be interested to find a particular movie by its keywords. These are the problems movie search aims to solve. Clarification: a movie in this context could also be a TV show or an episode of a TV show.

While quote search processes the inverted index at a sentence level, movie search only uses term occurrences on a movie level in the inverted index. This allows for a much faster processing time. Furthermore, we take advantage of MongoDB multi-level projection feature to request inverted index entries only up to the movie level, as shown in listing 6. This greatly reduces the amount of data that needs to be loaded from disk and processed into a Python nested dictionary structure.

```
{
  "_id": 1,
  "term": "father",
  "doc_count": 348007,
  "movies": [
    {
      "_id": "tt0002423",
      "doc_count": 1
    },
    {
      "_id": "tt0003930",
      "doc_count": 4
    }
  ]
}
```

```
},
:
]
```

Listing 6. Inverted index projected up to the movie level

Movie search can be activated by selecting the “Search for movies” checkbox on the GUI, and it uses a popularity-weighted TFIDF ranking algorithm to find the most relevant movies to the query. For each term, the inverted index entries are loaded and for each movie containing the term, the TFIDF score for the document-term pair is assigned (added) to a dictionary of movie ID, total score pairs. Since there are only around 120,000 movies in our collection, the full scores dictionary can easily fit into main memory.

$$TFIDF(tf, dl, df, p) = p * \frac{tf}{dl} * \ln \frac{N}{df} \quad (4)$$

The TFIDF score formula is shown in equation 4 and it uses the following parameters: tf is the term frequency indicating how many times the term appears in the movie (*doc_count* property of the movie). dl is the document length indicating the total number of terms (not necessarily unique) that the document contains. df is the document frequency indicating how many documents contain the term at least once (total sum of lengths of *movies* lists in the inverted index entries matching the term). p is the popularity of the movie defined by the total count of ratings on iMDB. N is the total number of movies that contain at least one term (number of movies we have subtitles for). In our case, N is a constant equal to 121,958, but for future work, a mechanism would be required to update N as new subtitles are added.

Since our inverted index was based on the sentence information, the movie-level *doc_count* means the number of sentences in the movie that contain the term at least once. We wanted to utilise the same hierarchical index for both the quote search and movie search, therefore, the tf and dl parameters have a slightly different meaning in our implementation of TFIDF ranking for movies. tf is the number of sentences in the movie containing the term, while dl is computed in the following way: for each sentence in the movie, add the count of unique terms appearing in the sentence. This method may be slightly less accurate than the actual TFIDF algorithm, but only marginally so in our case because movie sentences tend to be short and rarely contain the same term more than once (if the term is not a stop word).

A dictionary of dl and p values for each movie are pre-computed via ad-hoc database scripts and permanently stored in two pickle files (less than 6MB in total). On system load-up, the dictionaries are read into main memory so retrieval of the values is almost instantaneous. For future work, a mechanism would be required to update these dictionaries as new subtitles are added.

6.4. Additional Features

6.4.1. TIME LIMITS

If the user inputs a large number of words, or a query with a high proportion of common non-stopwords, the query runs the risk of taking very long. We thus implemented stopping mechanisms in our ranking functions. First of all, there is a total time limit per query. When the limit is reached, the

ranking algorithm stops and returns the intermediate results as they are saved in the Score Tracker. To prevent the algorithm from taking too long for specific words, the quote search additionally contains a stopping mechanism per term. When the search for one term takes too long, an exception is raised and the search continues with the next term, with partial results being saved in the Score Tracker.

6.4.2. SCORE TRACKER

For tracking the scores during the operation of the ranking algorithm, we promptly realised that, given our computational main memory constraints (2GB on Digital Ocean droplet), it would be infeasible to maintain a Python dictionary of float scores for up to 80 million sentences in main memory. Because of this, some sort of mechanism was required to keep a memory bound on how many scores can be maintained in-memory and discarding the lowest scores once the memory bound is reached.

A class `ScoreTracker` was developed that can track scores potentially for billions of documents. The class has an initialiser accepting a `max_size` parameter, and it has two public methods: `add_score(id, score)` for adding a score to a document id (for each document this would be called as many times as there are terms in the query) and `get_top(n, skip)` for retrieving the current top n document IDs and corresponding scores (this would be used at the end of the ranking algorithm to retrieve the best results). Internally, the class maintains a `heap` - a list of score, ID pairs which represents a [binary min-heap](#) data structure. The heap is maintained using $O(\log(N))$ insert and delete operations (via Python-native `heapq` library's methods) to guarantee that the first element of the list always has the minimum score. However, a heap is not enough to be able to efficiently add scores to existing documents in the `ScoreTracker`, so the class also maintains a dictionary `entry_finder` which stores document IDs as keys and pointers to heap entries as values.

Adding a score for a document `id` follows a procedure of 5 steps:

1. If `id` exists in the `entry_finder`, add the existing score of the document to `score`.
2. Check `score` against the minimum (first entry of the heap). If lower, abort.
3. If `ID` exists in the `entry_finder`, set the heap entry's document ID to the `REMOVED` flag.
4. Push the `(score, id)` pair to the heap and add the entry to the `entry_finder`.
5. Cleanup: Repeatedly pop the minimum score entry from the heap (and the `entry_finder`) while the entry's document ID matches the `REMOVED` flag or the size of the heap is higher than the `max_size` set during the initialisation.

Retrieving the best results via `get_top(n, skip)` is only meant to be done once per query. The method sorts the heap entries (filtered to not include `REMOVED` entries) by score in descending order, and returns `(id, score)` tuples based on positional `n` and `skip` parameters.

The `ScoreTracker` class provides a fast and memory-efficient solution to accumulate document scores and it has

enabled us to implement quote search using the BM25 scores. However, the speed and memory-efficiency brings two drawbacks that should be mentioned. First, the score tracker only adds a new entry if its score exceeds the minimum score kept so far. Because of this, a situation may arise in which a relevant document may not be able to enter the score tracker due to its individual term scores not being higher than the score of the minimum entry (which may be a sum of several scores). This can be mitigated by increasing the `max_size` value (which is currently set to 100,000) to a higher percentage of total number of documents, but the greatest fix for this issue was using movie popularity as a weight to the scores – the mechanism is biased to push sentences from the more popular movies into the Top 100,000. Second, the heap algorithm only allows popping the minimum score entry efficiently. To update an existing entry's score, we have to set its ID to the `REMOVED` flag (to be cleaned up later) and push a new entry with the updated score. This leaves some `REMOVED` entries maintained within the heap, taking memory space. Tests with `doc_count >> max_size` and random scores from the Gaussian distribution showed that the count of removed entries kept converges to but does not exceed $\frac{max_size}{2}$.

6.4.3. CACHE OF RESULTS

The GUI allows to easily enable or disable advanced search or movie search. Any time a user changes any of the search options, a new request (either `/query_search` or `/movie_search`) is issued to the API, querying for the results for the following set of query parameters: `query`, `movie_title`, `categories`, `actor`, `keywords`, `year`. In order to avoid repeating the ranking algorithms for the same queries in case the user wants to come back to a previous query or if the query is popular among other users, a caching mechanism was implemented using the Least Recently Used (LRU) replacement policy (implementation of LRU Cache provided by package `cacheout` ([Gilland](#))). Two caches of maximum 512 query results each are maintained in a custom Singleton class `ResultsCache`.

The `ResultsCache` class has three public methods: `get(query_params, which_cache)` for retrieving the output from a cache or `False` if the output does not exist for such query parameters, `store(query_params, output, which_cache)` for storing the output associated with the query parameters in a cache, `clear_all()` for emptying all caches, the last one used for unit testing. The main requirement for a cache is that the key (`query_params` dictionary) must be hashable. Since all the query parameters are encoded in strings, a unique (with extremely high probability) hash can be obtained using the following built-in Python methods: `hash(frozenset(query_params.items()))`

Since `categories` and `keywords` are strings containing a comma-delimited list of categories or keywords, the caching mechanism further ensures that different query parameters with same semantics would produce the same hash by splitting each comma-delimited string, sorting the words (converted to all lowercase) alphabetically and re-joining them back into one comma-delimited string.

After receiving a request, the API first uses the cache's `get` method to try to obtain the output for the request. If the output exists, then the `query_time` value in the output is updated to reflect how long it took from receiving the query to obtaining

it from the cache, and the output is sent to the GUI. If the cache does not contain the output for the query parameters, the results are generated in a usual way using the appropriate ranking function, and the output is stored in the cache just before sending the output to the GUI.

To save space, the outputs for queries contain only minimal information required to display the results in a list as can be seen on our GUI. Pressing on an individual result invokes another query for movie details by ID that returns complete information for the movie. This way, a full cache of 512 quote results and 512 movie results should take around 250MB of main memory based on our estimates (where each result contains a list of 500 documents).

7. Advanced Search

Advanced search facilitates the user to search the quotes on a more refined level. Users can input any movie title or year range or actor name or categories to filter out their desired quotes. For example, an input for year '2000-2005' will only retrieve the quotes within the year range. Advanced search is invoked before carrying out ranking so that unnecessary quotes are not ranked, thereby enhancing the performance.

This has been implemented by applying MongoDB projection where the movies are filtered based on the user inputs and only these filtered movies corresponding with the inverted index entries retrieved are provided as the inputs to the ranking algorithm. Moreover, GUI has elegant features to accommodate the advanced search parameters.

The user can also enter keywords in the advanced search. Keywords are terms defined on iMDB, that further describe the movie, such as 'murder mystery' or 'detective'. Since the user might not know the exact keywords for a movie, we chose to only sort the results instead of filtering them. The results that contain at least one keyword are moved to the top, without changing the order within those results. This sorting is implemented in the API, after the quotes and movie details are retrieved.

8. Query Completion

After we made sure that all the basic features of the search engine are functions very well, we started researching and implementing some advanced features such as spelling check, query expansion, and query completion. For the spelling check feature, we have explored many options but we decided not to implement it at the end since it might cause a delay in the retrieval time and it is better to be implemented in the front-end. Query expansion was one of the features we researched and discussed with each other about. After careful evaluation for this method, we decided that we are not going to use it since it will most likely cause a delay in our website while not adding much value to the system (it might even make the results worse since we care about the exact matches and not only the aboutness).

Query completion was the valuable addition which needed so much researching and coding to make it work properly. We implemented a language model (a bi-gram model) which predicts the next word given the previous one. We chose the bi-gram model because it is scalable and can give a fast prediction for the users as soon as they type the first word. We trained the model using a Recurrent Neural Network (RNN),

specifically a Long Short-Term Memory (LSTM) model (Gers et al., 1999). The problem with this model is that it needs so much time and computing power to train the model on the whole dataset, which is not feasible with the current conditions. To avoid this problem, we decided to train the model using the 1000 most popular movies of all time, since these would most likely have high quality dialogues and will contain most of the famous quotes. The results were satisfactory and the suggestions have been added as a drop-down list which will suggest the next word, two words, and three words after each word typed by the user. Given more time, we would train the model using the whole dataset and using more epochs to get more accurate results.

9. API

The API is the middleware between the GUI, the ranking function and the database. It is built using [Flask](#), a popular Python web framework, which allows the creation of end-points that the GUI points to to retrieve data. Depending on the search input, there are two distinct search functions that can be executed: the *query search function* and the *movie search function*.

The *query search function* receives the query input from the GUI, along with indicators for advanced search and movie search. It then preprocesses the query in the same way as described in 4 and calls the *ranked retrieval* quote search function described in section 6. The ranking function outputs a sorted list of quote ids, which are then used to retrieve the full sentences, movie ids and corresponding movie details using the database functions described in section 3.3. Then, a list of the retrieved categories is made and returned along with the query results. If keywords are specified in the advanced search, the results are resorted, moving the results with at least one keyword to the top. The order within movies with keywords, and movies without stays the same. The results, containing the full quotes, and all corresponding quote and movie details are then returned as a JSON to be used by the GUI. The results of the query search is displayed in figure 3.

The *movie search function* works similarly. It has the same input as the *query search function* and preprocesses the query in the same way. But instead of calling the *ranked retrieval* function in ranking, it calls the *ranked movie search* function. This function return movie ids, which are then used to retrieve movie details and create a genre list. If keyword filtering is enabled, it is performed in the same as in the query search. The results are returned to the GUI in the same format as the query search, without the full quotes.

10. GUI

For the GUI we used the [React JS](#) library, since it is ideal for building responsive user interfaces, allowing for efficient handling of huge amount of data without making the application slower. We take advantage of its component-based architecture, which renders components into the DOM only when needed. The main parts of the GUI include an input for the quote query, an advanced search which allows the user to enhance the search by providing more attributes (*movie title, actors, year range and keywords*) and a second search which only returns the movies/tv-series where the query was found.

The quote results are rendered in cards, where information for each quote and movie is visible, such as quote, character,

time the quote was mentioned in the movie and thumbnail, as seen in Figure 3. The second search is run by clicking on the 'Search for movies' checkbox and it returns only the movie names, as seen in Figure 5. A card with movie details is visible on the right-hand side, which appears by clicking on any quote/movie and provides a movie description, a list of people in the cast and a link to the IMDb page.

The GUI is also enhanced with the following features that support the back-end functionality. The advanced search, visible when toggling the corresponding button, allows the user to perform an advanced filtering of the results before the search is triggered, using the fields *movie title*, *year range*, *actor/actress* and *keywords*. The query completion is triggered when the user types in a word, and possible words after it are being suggested by the model. The suggestions are provided in a dropdown underneath the query input. We also use pagination, to prevent simultaneous rendering of all the results and enable a good user experience. A front-end filtering is also provided, which allows the user to filter the movies by genre.

11. Evaluation and Future Work

All the basic features in the search engine are working fine and can be used many types of users. Indeed, individual users can benefit from using the search engine but also some specific types of users might see our website as a great help. For example, YouTube content creators usually use clips containing some famous useful quotes from movies to illustrate their points. Our search engine can show the exact time when these quotes were said, which would help the content creators find a quote more easily.

We are also aware that our system has some limitations. For example, we are using static data, which we downloaded all at once and created the whole project based on that. A periodic update would be needed to add the newly released movies to our database. Apart from that, we currently do not have a way of evaluating the performance of our system. Since there is no annotated data, we can not use Precision, Recall, or F-1 score. The best way to evaluate a search engine is to use users' feedback and users' behaviour, but the problem here is that we currently do not have actual users so we evaluated the performance of the system using the team members' experiences in movies and TV shows.

There are many future developments which can be performed to improve the performance of the search engine. First, we would train the query completion model on the whole dataset to make it able to give better predictions for the next words. Second, we would implement a spelling checker in the front-end instead of the back-end, which will make it faster and more efficient. We would also have a back-end pagination, which would allow for the retrieval of more results without loading the page at once. To get useful feedback from users, we would include a form so they can inform us whether the results were relevant to the query or not. Finally, we would implement an optional filter which blocks the explicit words, which can be used to prevent the results related to these words from appearing in the search results.

12. Individual Contributions

At the beginning of the project we decided to work on sub-groups to have all the basic features ready as early as possible. After that, each team member started working on some in-

dividual advanced tasks which improves the performance of the system. That said, there will be several overlapping tasks between the team members, especially in the basic/necessary tasks.

S1965737 - Oguz

I developed the parsing, preprocessing and inverted index parts of the projects and worked on designing the system. I designed a detailed system architecture and work plan according to group members' previous experiences.

For parsing the subtitle files, I analysed the subtitle formats we have and developed a Python script that extracts the quotes from the subtitles and writes them into the database. The challenge in this part was that some of the subtitles were not generated in the exact SubRip or MicroDVD formats. I had to find the differences or possible format mistakes then modify the parser script for parsing that files too. I used regular regex library of Python for this task. I developed the tokenization, stemming and removing the stop words with Maysara by using nltk library in Python. We also built the MongoDB on the droplet.

I developed the script for generating the inverted index for 77.5 millions of quotes. It was a challenge by itself because collecting all of the documents for each term in this large data collection needs too much computational power, memory and storage. On the other hand, MongoDB does not allow to store any single record larger than 16MB. The document numbers for even a single term can be larger than 16MB in our index. Thus, I created an inverted index by dividing the quote collection into parts so that the terms could be appear more than once (up to 52) in the index. Each of the records has a unique index in the database and the term field is also set as index in DB collection (selecting index in DB creates a B-Tree on that field). It is not a problem having the same terms in different records in this structure. The index is also designed as hierarchical and it occupies 21.7GB in DB.

S1837964 - Maysara

At the beginning, I started by working on the preprocessing task, I worked on building the scripts used for parsing the different types of files we have. As mentioned earlier, the problem here was that many files do not exactly follow the standard way of creating a subtitle file, so we needed to discover these problems and take them into account while writing the scripts. After that, I worked on the other preprocessing tasks, such as tokenization, text normalization, stemming and stop words removal. These were critical tasks because they will be used in several other parts of the project such as the api and the advanced features. We needed to make sure that these functions are perfectly suitable for our purpose and that they are compatible with the different parts of the project.

After that, I moved to the indexing and database building. I worked with Oguz on building the inverted index for the all the files we have and then save the inverted index in an efficient way, because the index size was huge. After creating the hierarchical positional inverted index, as mentioned earlier, we built a data base using MongoDB to store the index in an efficient and organised way.

Finally, after the basic features were done, I worked on researching and implementing some advanced features such as spelling check, query expansion, and query completion. I had

to research each of these topics theoretically and practically. I started by reading about each of these features then implementing the best available solution. After implementing, I needed to check the effect of adding each of these features on our system. I decided not to use query expansion because of computing constraints. The spell check code was working in the backend, but after discussing with the team, we decided that this feature should be implemented in the front end. Query completion was a big task, which needed lots of research on how to implement it on such a huge dataset. At the end I decided to build a language model and train it using an LSTM. Even though this method yields very nice results, it needs so much time and computational power to train it on the full dataset, so I decided to optimize the task as much as possible, as mentioned earlier in the report.

S1958674 - Marina

I initially set up the project structure on Github and installed React and Flask so that the connection of front-end and back-end is established before we started working on the project. I completed almost all features in the GUI, from building a robust architecture to rendering the results correctly into each component. For each event triggered in the GUI, I make a REST HTTP call (GET or POST) to existing endpoints in the API and store the results in the state of parent components.

The most important GUI features are swapping between quote and movie search, the advanced search feature and the user-friendly transition between quotes and movie information. In addition, I used [Scss](#) to style the components following the principles of [Material Design](#).

Finally, I created a React production build, which minifies the JavaScript and Scss files and is used to reduce the load time and the bandwidth usage of the website. I further configured it so the application is being served from within the Flask environment.

S1885912 - Leonie

I worked on API, ranking and the database functions. Specifically, I implemented the quote search function in the API, which receives the query parameters and returns the query results. I also wrote API functions such as the function to create a list of retrieved categories and the sorting after keywords. In ranking, I implemented the first basic TFIDF, and adjusted it according to inverted index structural changes. I also refined the BM25 by implemented database retrieval for the inverted index. I also implemented the timing function, popularity weighing in quote search and enabled the BM25 to save its results using the ScoreTracker. I implemented the use of PyMongo in the MongoDB module, and wrote functions to retrieve sentences by quote id, movie details by movie id and inverted index entries for quotes per term. I also connected the API, ranking and databases to use each other as modules.

Since we all worked together as a team, I also tested and reviewed other parts of the code.

S1964892 - Maheshkumar

I have contributed to ranking, GUI and the database functionalities. I implemented the BM25 algorithm for ranking and worked on its further refinement. Advanced Search feature has been implemented by me which involves the querying movies from MongoDB efficiently with the use of MongoDB

projection feature to only return movie IDs such that this feature doesn't slow down the ranking algorithm. The initial test collection of movie details (1,000 documents) have been inserted by me to the MongoDB following the structure so that access from the collection can be made efficiently. I helped Marina on some of the GUI features such as front-end pagination, adding quote examples and some other minor functionalities.

S1452787 - Kasparas

I completed the whole data collection part of the project: a module to web-scrape iMDB data and download subtitles. I contacted OpenSubtitles team for help and processed and organised the subtitles dump received from them. I wrote numerous scripts for performing ad-hoc operations on the database such as uploading movies data from JSONL, computing the average sentence terms length (for BM25), finding term counts for each movie (for TFIDF movie search), fixing sentence IDs to sequential integer format (for saving space) and several others. Most of the scripts were optimised to perform database operations in batches, processing up to 100,000 documents at a time, and were made to have a stop/resume functionality. I made a documentation page for our API which can be accessed at <http://167.71.139.222/api/docs/>. The documentation was meant to be the single source of truth and I was updating it regularly to display all the request and response schemas to aid front-end development. Further, I made unit tests (16 by the end) for various functions in the project to help catch any bugs before merging new changes to the master branch.

For information retrieval, I implemented the ScoreTracker class (based on the heap algorithm) and helped Leonie and Mahesh to optimise the BM25 algorithm for quote search. Further, I implemented the quote phrase search performing a linear merge using positions to match only the exact phrases within sentences, and return the matched sentences ordered by corresponding movie popularity. Finally, I implemented the movie search using TFIDF scoring weighted by the popularity of the movies. For API, I developed endpoints `/movie_search`, `/movie/:movie_id` and `/query_suggest` (using Maysara's model to predict up to 3 next words in the query), and implemented the LRU caching mechanism for matching query parameters to results. Finally, I configured the production environment including machine learning libraries used and set up the Flask production build to run permanently on our DigitalOcean droplet, serving the API via nginx and uWSGI. For the report, I wrote sections 3, 6.2, 6.3, 6.4.2 and 6.4.3.

References

- Gers, Felix A, Schmidhuber, Jürgen, and Cummins, Fred. Learning to forget: Continual prediction with lstm. 1999.
- Gilland, Derrick. Cacheout, howpublished = "<https://pypi.org/project/cacheout/>", year = , note = "[online; accessed 25-february-2020]".
- Lixin Xu, Guang Chen, and Lei Yang. Incremental clustering in short text streams based on bm25. In *2014 IEEE 3rd International Conference on Cloud Computing and Intelligence Systems*, pp. 8–12, Nov 2014. doi: 10.1109/CCIS.2014.7175694.

Appendix

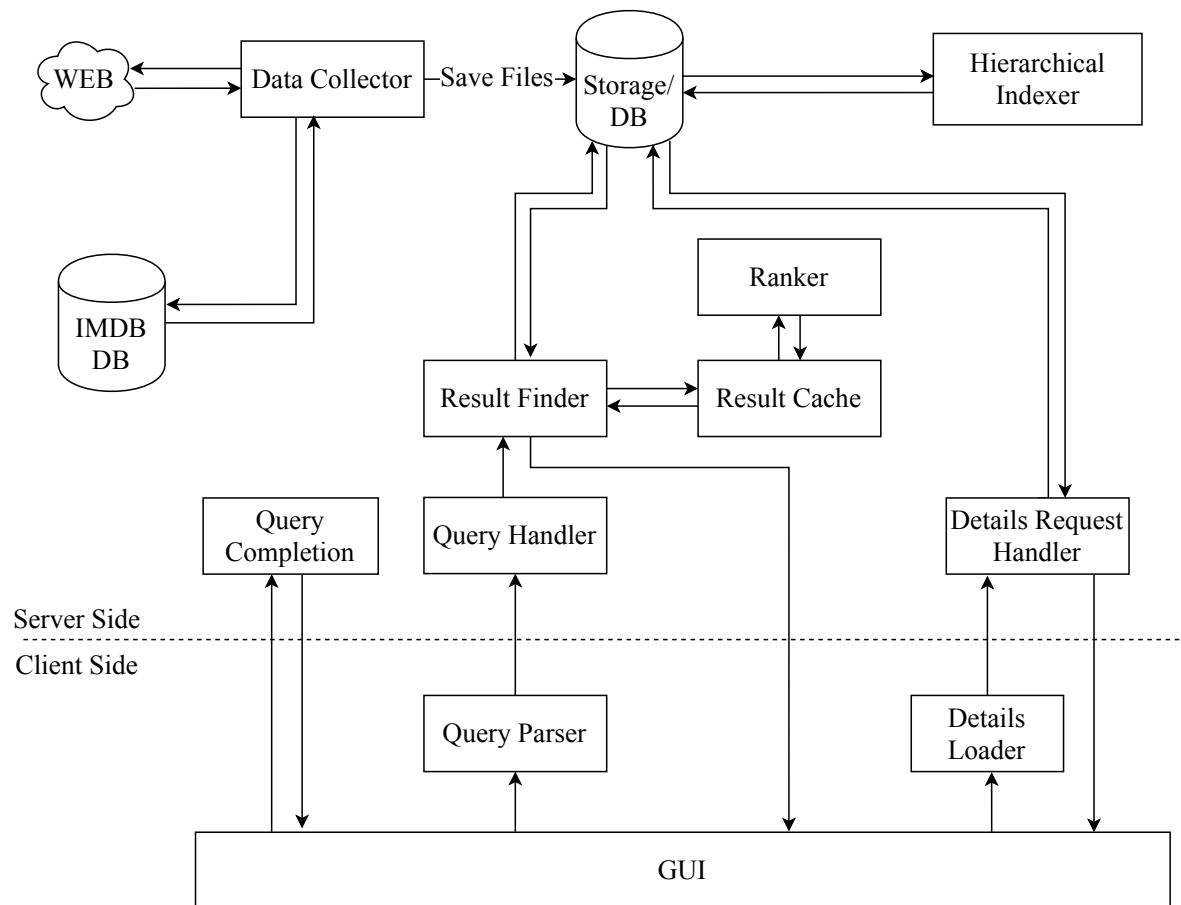


Figure 1. The system architecture of the search engine.


```

{
  "_id" : ObjectId("5e4bc789bce8840fc530c041"),  This can be safely ignored (it is required for MongoDB)
  "term" : "father",
  "doc_count" : 348007,  This is the total sentence count among all inverted_index entries matching the term
  "movies" : [
    {
      "_id" : "tt0002423",  _id in movies collection (IMDB ID)
      "doc_count" : 1,  Number of sentences in the movie containing the term
      "sentences" : [
        {
          "_id" : 292,  _id in sentences collection (all sentences are numbered from 0 to 75M+)
          "len" : 7,  Length of the sentence (used for BM25)
          "pos" : [  Positions inside the sentence where the term appears
            1
          ]
        }
      ]
    },
    {
      "_id" : "tt0003419",
      "doc_count" : 1,
      "sentences" : [
        {
          "_id" : 692,
          "len" : 206,
          "pos" : [
            6,
            15
          ]
        }
      ]
    }
  ]
  ...
}

```

Figure 2. Inverted Index Nested Multi-level Structure

Movie Quotes Search Engine

Search for a movie quote...
May the force be with you


SEARCH ☐ Search for movies

☐ Advanced Search


[DRAMA](#) [ACTION](#) [ADVENTURE](#) [COMEDY](#) [CRIME](#) [THRILLER](#) [MYSTERY](#) [ROMANCE](#) [FANTASY](#) [HORROR](#) [ANIMATION](#) [BIOGRAPHY](#)
[WAR](#) [FAMILY](#) [HISTORY](#) [SPORT](#) [MUSIC](#) [MUSICAL](#) [DOCUMENTARY](#) [WESTERN](#) [SHORT](#)

Query results: 500 movies (8.920 seconds)


< 1 2 3 4 5 ... 49 50 >



"May the Force be with you." - Leia
 Star Wars: Episode VII - The Force Awakens
 Quote was said at 02:03:17
 Category: Action, Adventure



"May the Force be with your Schwartz." - Wade
 Friday the 13th
 Quote was said at 00:10:36
 Category: Horror, Mystery, Thriller



"Japanese forces may attack the Philippines. . ."
 Tora! Tora! Tora!
 Quote was said at 00:53:03
 Category: Action, Drama, History

Star Wars: Episode VII - The Force Awakens
 Three decades after the Empire's defeat, a new threat arises in the militant First Order. Defected stormtrooper Finn and the scavenger Rey are caught up in the Resistance's search for the missing Luke Skywalker.
 Year: 2015
 Rating: 7.9

Cast
 Harrison Ford as *Han Solo*
 Mark Hamill as *Luke Skywalker*
 Carrie Fisher as *Princess Leia*
 Adam Driver as *Kylo Ren*
 Daisy Ridley as *Rey*
 John Boyega as *Finn*
 Oscar Isaac as *Poe Dameron*
 Lupita Nyong'o as *Maz Kanata*
 Andy Serkis as *Supreme Leader Snoke*

Figure 3. Query search performed using the quote "May the force be with you". The search returns a list of quotes along with details about the character who said the quote and information about the movie.

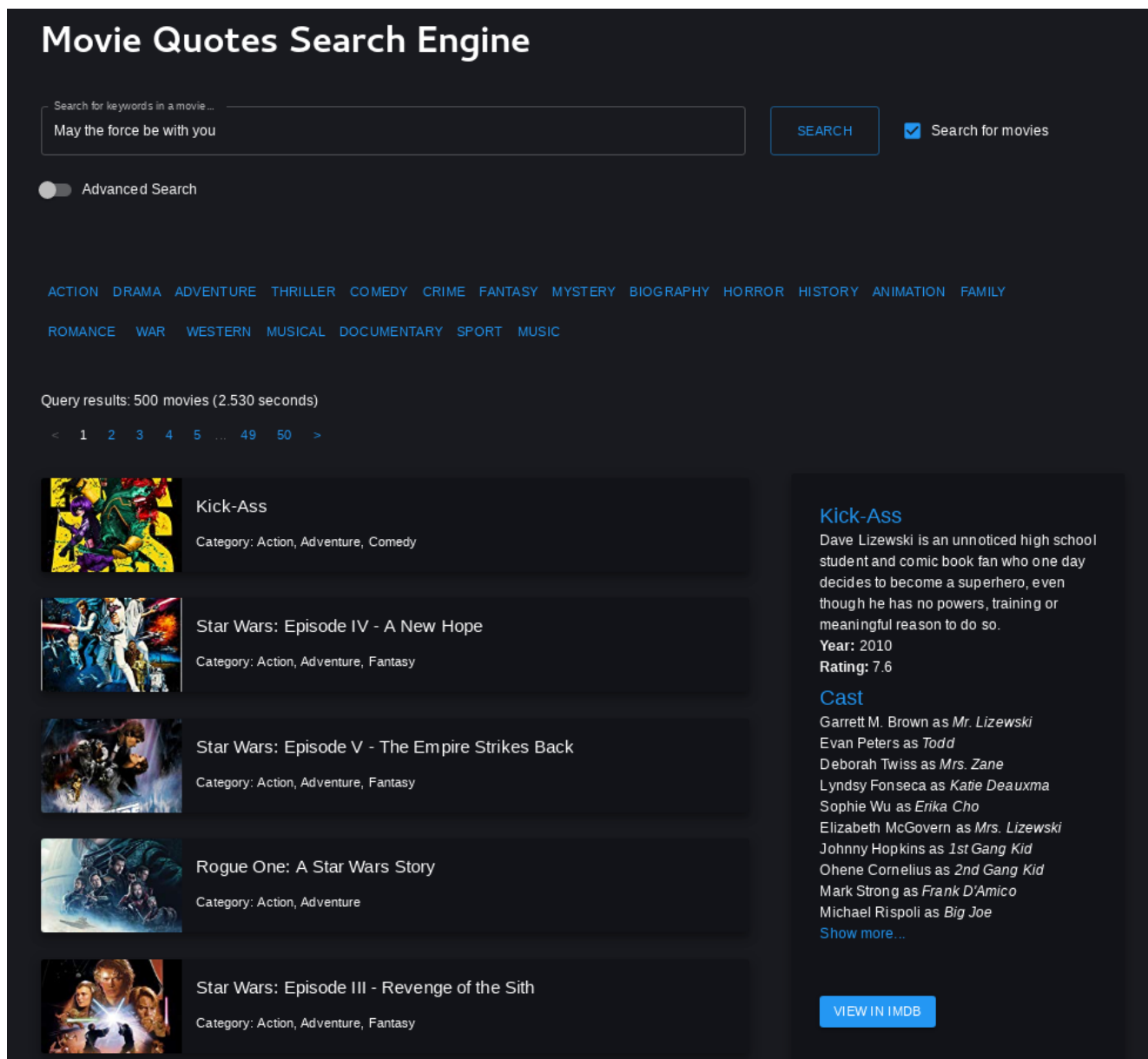


Figure 4. Movie search performed using the same quote "May the force be with you". The search now returns a list of movies where the quote was found.

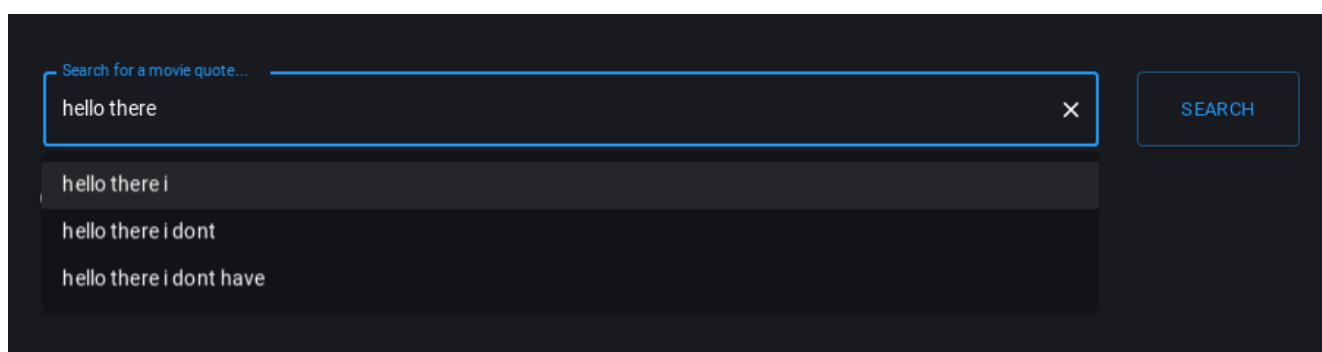


Figure 5. Query suggestion example