Assignment 1

Abhinav Verma- 2020A7PS1093H
Ritvik - 2020A7PS1723H
Abhinav Tyagi - 2020A7PS2043H

Constructing Inverted Index Table

For constructing the data structure inverted index table, the input data is provided in the form of a json file. For each document to be added, a json file stores the title and the different sections of the document. For each section we have the section name, and the paragraphs that are present in that section.

Before adding words to the inverted index table, preprocessing is done by converting words to lowercase and all the numbers are removed from the words. In order to tokenize, **'nltk.tokenize'** package is used. In order to remove the stop words, **'nltk.corpus'** package is used.

In order to deal with wildcard queries, at this stage we store all the prefixes, substrings and suffixes. For prefixes and substrings, separate files are created for words starting from the same first character. For suffixes, in one file suffixes that end with the same character are stored. After this, **Porter Stemmer** is used to reduce words to their base forms, before being added to the inverted index table.

In the data structure inverted index table, for each word the following are stored:

- Paragraph Frequency - Stores the number of paragraphs in which word occurs
- List of Paragraph IDs - Stores the IDs of the paragraphs in which the word occurs.
  <u>For each paragraph:</u>
- Positions - Stores the list of positions where the words occurs in the paragraph
- Sentence IDs - Stores the list of IDs of sentences in which the words occurs in the paragraph
- Frequency - Stores the frequency of the word in the paragraph

Also at this stage the **title mappings** are done to the paragraph IDs, and for each document, **sections mappings** are done to paragraph IDs. This would be useful later on during paragraph retrieval.

The working of the method that adds a document to the Inverted Index Table is described below:

The approach repeats over the paragraphs in each section of the document and tokenizes each paragraph into a phrase. For each sentence, it tokenizes the words and utilizes NLTK to conduct part-of-speech tagging. After that, it repeats the process, processing each word in the sentence as follows -

1. Remove any non-ASCII characters and change the word's case to lowercase.
2. Check if the word is a stop word or contains a digit. If so, skip it.
3. Use the Porter stemming algorithm to the word.
4. The index data dictionary, a dictionary of dictionaries that holds the inverted index, is accessed by using the first character of the stemmed word as an index.
5. Add the character and initialize it to an empty dictionary if it isn't already a key.
6. If the stemmed word isn't already a key in the sub-dictionary, add it and initialize it to a dictionary with a single key-value pair, which should include the current paragraph ID, a dictionary with the word's position in the paragraph, the ID of the sentence that contains it, and a frequency count of 1.
7. Update the value of the stemmed word if it is already a key in the sub-dictionary with the current paragraph ID and the word's location, and if the word has already appeared in the current paragraph, increase the frequency count.

Other data structures are also created and updated by the procedure, such as dictionaries for prefixes, suffixes, and substrings, as well as a dictionary that links the text of each paragraph to its associated paragraph ID.

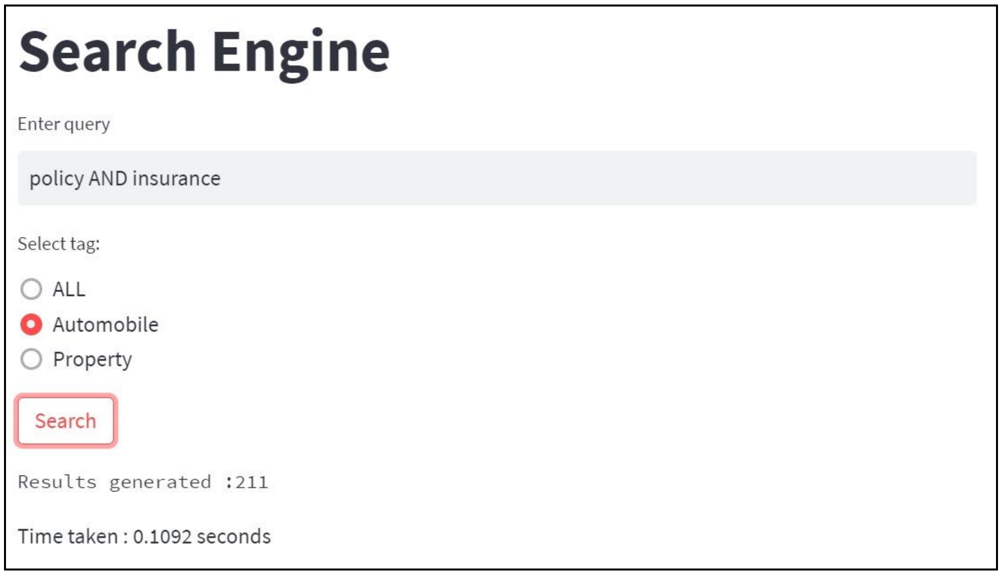
Finally, the method writes the updated index data, paragraph mapping, and prefix/suffix/substring dictionaries.

Whenever new documents are to be added to the Inverted Index Table data structure, the parsed json files are first added to the **'parsed_json'** folder, and after the indexing process is over, the file is shifted to the **'done_processing'** folder. To add a new document to the inverted index table, there is no need to create mapping and affixes again; these files would simply be updated.

Query Processing

In query processing, for the binary operators, the precedence order is as follows:

1. AND
2. \\n
3. OR



Search Engine

Enter query

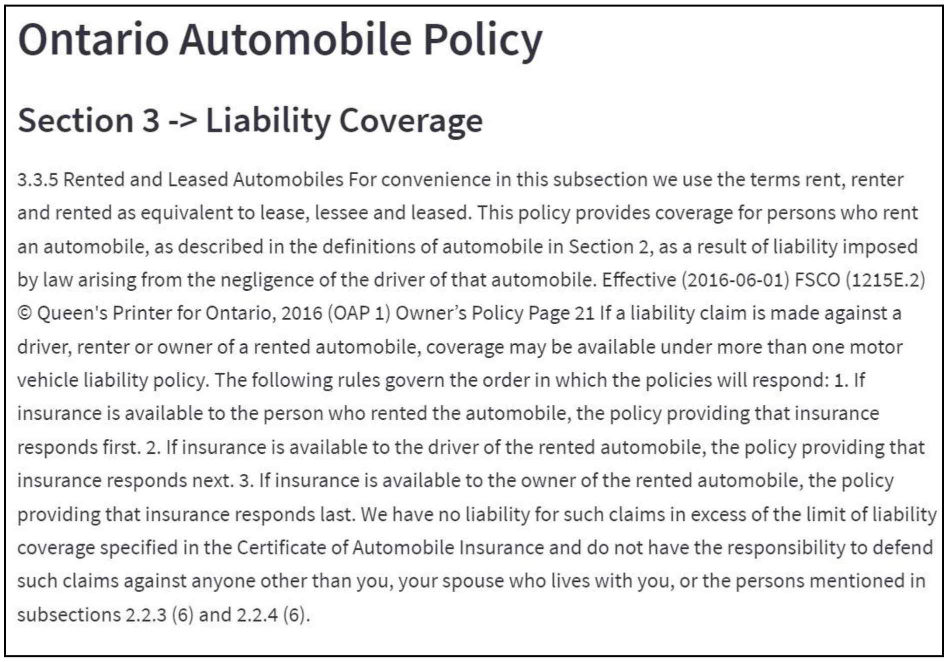
policy AND insurance

Select tag:

ALL [✓ Automobile] Property

```
Results generated :211
```

Time taken : 0.1092 seconds



Ontario Automobile Policy

Section 3 -> Liability Coverage

3.3.5 Rented and Leased Automobiles For convenience in this subsection we use the terms rent, renter and rented as equivalent to lease, lessee and leased. This policy provides coverage for persons who rent an automobile, as described in the definitions of automobile in Section 2, as a result of liability imposed by law arising from the negligence of the driver of that automobile. Effective (2016-06-01) FSCO (1215E.2) © Queen's Printer for Ontario, 2016 (OAP 1) Owner’s Policy Page 21 If a liability claim is made against a driver, renter or owner of a rented automobile, coverage may be available under more than one motor vehicle liability policy. The following rules govern the order in which the policies will respond: 1. If insurance is available to the person who rented the automobile, the policy providing that insurance responds first. 2. If insurance is available to the driver of the rented automobile, the policy providing that insurance responds next. 3. If insurance is available to the owner of the rented automobile, the policy providing that insurance responds last. We have no liability for such claims in excess of the limit of liability coverage specified in the Certificate of Automobile Insurance and do not have the responsibility to defend such claims against anyone other than you, your spouse who lives with you, or the persons mentioned in subsections 2.2.3 (6) and 2.2.4 (6).

For the unary operators and wildcard operators, there is no such priority order. Keeping these points in consideration, a parse tree is generated for the queries.

For multiple ‘AND’ operators, it takes more time if the ‘AND’ operations are performed in a nested order. One optimization that has been performed is **flattening of the parse tree**. The ‘AND’ operations are performed in a linear fashion, resulting in faster performance. This is due to the fact that as the parse tree becomes flattened, it is faster to traverse it.

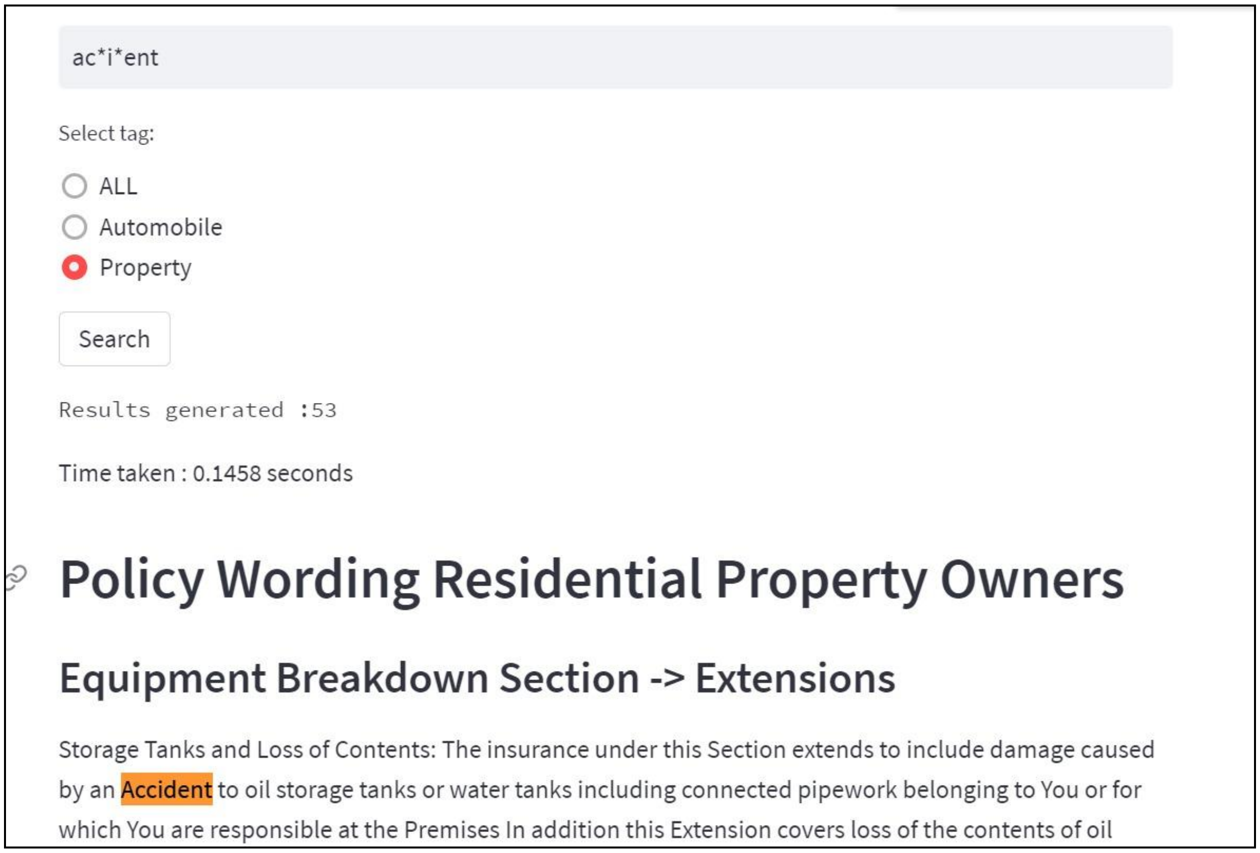
The queries provided by the user can be categorized into 2 types:

1. Basic Queries
2. Wildcard Queries

In order to process wildcard queries, if there is only one word in the query, the first results for the first word found is returned as the answer. If there are multiple words int the query, first all the possible combinations are generated. Then the relevance of these combinations generated are checked with the other words of the query using **‘nltk.util.ngrams’**.

Starting from the most relevant generation, it is checked whether the combination is present in the Inverted Index Table or not. If it is present, the results are returned accordingly. Otherwise, the next most relevant combination is checked.

An example of wildcard query has been shown on the next page:


ac*i*ent

Select tag:

ALL
Automobile
[✓ Property]

Search

```
Results generated :53
```

Time taken : 0.1458 seconds

Policy Wording Residential Property Owners

Equipment Breakdown Section -> Extensions

Storage Tanks and Loss of Contents: The insurance under this Section extends to include damage caused by an Accident to oil storage tanks or water tanks including connected pipework belonging to You or for which You are responsible at the Premises In addition this Extension covers loss of the contents of oil

For basic queries (queries not containing wildcards), if a word searched is not found in the Inverted Index Table, we follow the following procedure. The word is searched using **‘nltk.corpus’** and if it is found in the English language, the synonyms of the word are searched.

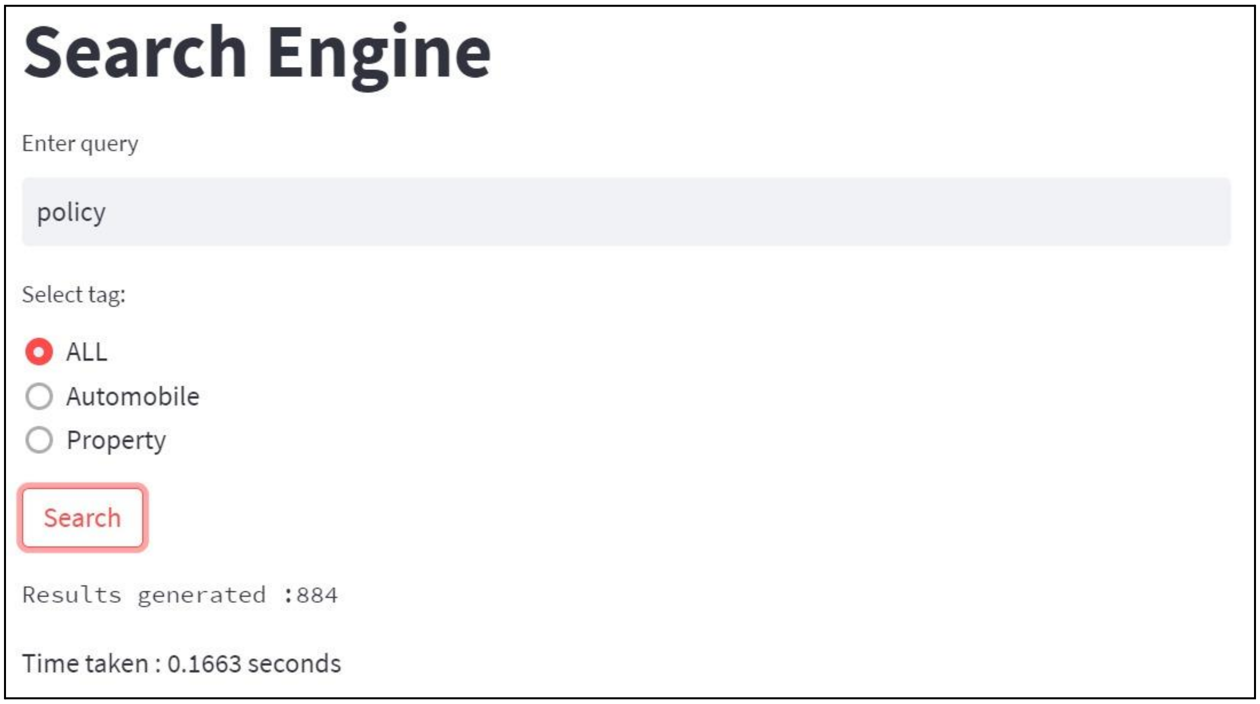
Of the top 10 **synonyms** that are returned, starting from the top it is checked if the synonym is present in the table or not. In case it is present, the sections containing it are returned. Otherwise the next synonym is checked. If we get no result for the top 10 synonyms of the word, it is concluded that there is no result matching the query given by the user.

If the word is not found in the English language, it is assumed that the word may have been **misspelled**. In this case, using **Jaccard Distance**, the top 10 words that the user may intended to search are calculated. Again the same procedure is applied here.

Starting from the top it is checked if the word is present in the table or not. In case it is present, the sections containing it are returned. Otherwise the next closest word is checked. If we get no result for the top 10 words, it is concluded that there is no result matching the query given by the user.

Also another optimization that has been done is ‘**caching’**. On run-time, whenever a word is searched its results are stored. So the next time if the same word has to be searched, there is no need to search the inverted index table. This results in faster retrieval time.

Below is an example where when the term ‘policy’ was searched the second time, the result was significantly faster.

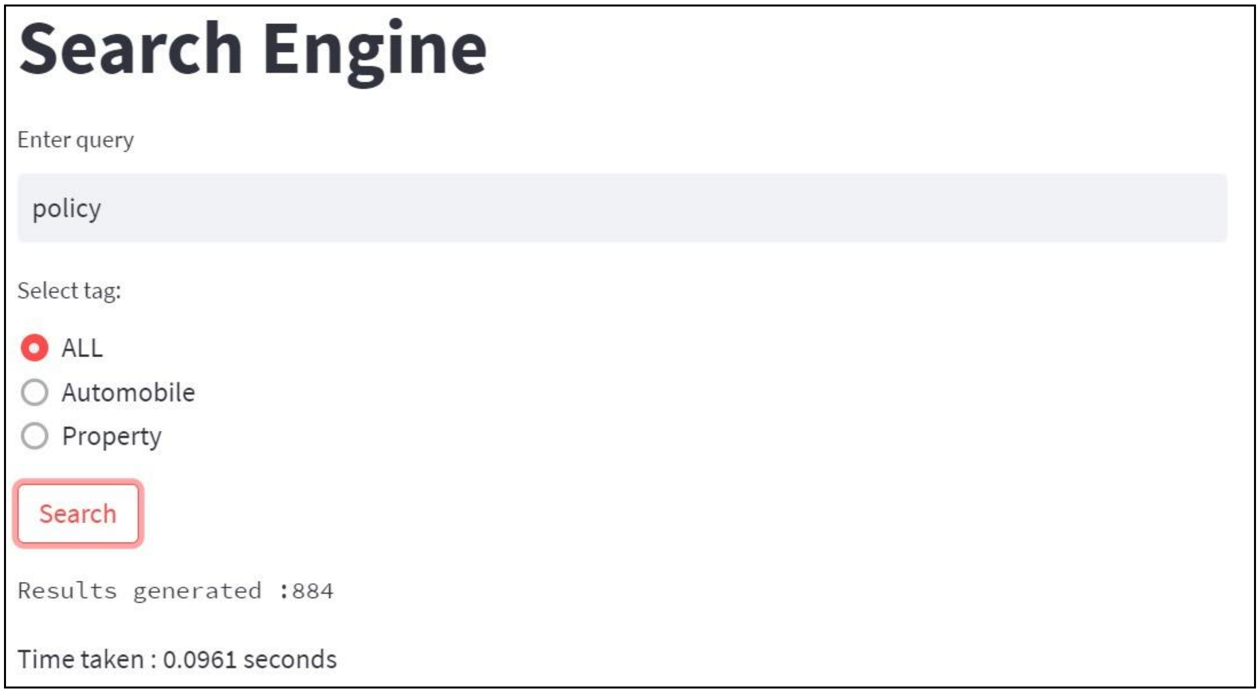


policy

Select tag: [✓ ALL] Automobile Property

```
Results generated :884
```

Time taken : 0.1663 seconds



policy

Select tag:

[✓ ALL]
Automobile
Property

```
Results generated :884
```

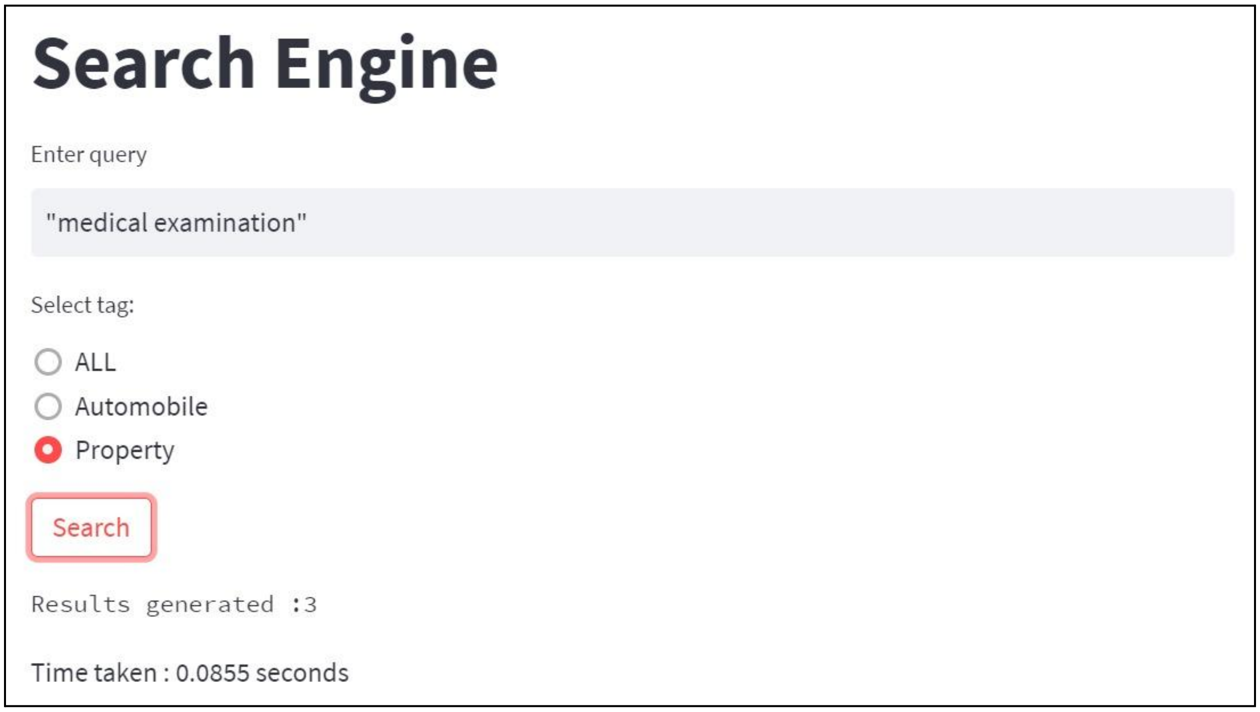
Time taken : 0.0961 seconds

The ‘AND’ operator provides the result which contains both the words on either side of the operator. The ‘OR’ operator provides the result which contains either of the words on either side of the operator. Also it should be noted that the ‘NOT’ operator before a word should be used by the user only if there is an ‘AND’ operator along with this word. Otherwise it actually does not make sense to use the ‘NOT’ operator alone.

The meaning of some operators are given below:

1. \\s : unary operator to be used when the user wishes for the words to be in the **same sentence**
2. \”w\” : unary operator to be used when the user wishes to give a **phase query**
3. <suffix> : wildcard operator to be used when the user wishes to specify the suffix of a word to be searched
4. <prefix> : wildcard operator to be used when the user wishes to specify the prefix of a word to be searched
5. <substring> : wildcard operator to be used when the user wishes to specify the substring of a word to be searched

An example of phrase query has been shown below:


Search Engine

Enter query

"medical examination"

Select tag:

ALL

Automobile

[✓ Property]

```
Results generated :3
```

Time taken : 0.0855 seconds

Paragraph Retrieval

The constructor of the paragraph retrieval class takes as input the output of the Inverted Index Table, which contains information about the frequency of words in each paragraph of each document. This information is used to rank the relevance of the paragraphs based on the query.

The **'rank_retrievals'** function computes the relevance score for each paragraph based on its frequency and the total number of paragraphs in the collection. The relevance score is computed by using the logarithm of the frequency of the terms, ensuring that the most relevant paragraphs are returned.

The **'retrieve_paragraphs'** function retrieves the top 10 paragraphs based on the relevance score computed in the previous step, using the paragraph mapping information stored in the **'paragraph_mapping'** folder. This information maps each paragraph to the corresponding document and section in which it appears. The function returns a list of dictionaries containing the document name, section name, and the relevant paragraph.

The paragraph retrieval process implements a kind of **‘distributed index table’** that is used for generating **‘filtered results’**. It is not needed to carry out a linear search on all documents to find which document has the required paragraph ID to be retrieved. For this purpose, the paragraph IDs have been mapped to the documents, and the document containing the paragraphs to be retrieved can be found by following this mapping.

Furthermore, for each document, the paragraph IDs have been mapped to the sections, making it even faster to retrieve the section satisfying the query. These mappings combined result in a **tree-like structure** where the nodes of the tree contain the actual data.