Web Information Extraction and Retrieval Programming Assignment 3: Document indexing and querying

Marko Prelevikj
63130345
mp2638@student.uni-lj.si

Gojko Hajduković
63180431
gh8590@student.uni-lj.si

Stefan Ivanišević
63170405
si0539@student.uni-lj.si

May 2019

1 Introduction

With the increasing development of information technologies, in the modern, digital era, most of the documents and information are stored in a digital format in order to be easy accessible to everyone within the globe. Many of the problems arose when that increasing number of data needed to be stored efficiently in order for users to have quick access to the information needed. In many information system where some type of searching is required, first natural approach in finding a related words, queries within the documents was the naive approach of sequentially looking into all of the documents for the specific words, which was very inefficient regarding to time and space. Nowadays, the most widely used and efficient way of storing the data in order to be searched from it quickly is the concept of Inverted Index.

`Inverted index` represents an efficient technique for storing mappings of words, content to its locations in document or in a set of documents. In this paper we introduce our implementation of first preprocessing the documents, then building an inverted index from the preprocessed content in order to allow users quick search of the content in need. We also implement the naive approach, `sequential file reading` and compare its efficiency with the approach based on inverted index. Explanation and implementation specifics are provided in following sections.

2 Data Pre-processing

For a more efficient indexing step, we needed to preprocess the corpus. The corpus contains 1416 crawled web pages. Each web page is processed in the same manner as described in continuation.

First, we extract the text data using the package inscriptis. Beside that, we also normalizing the text into lowercase. Since we retrieved some of the `HTML` tags alongside with the textual data, we used regex to remove these tags.

The next step during preprocessing was to tokenize the text. We have performed tokenization (splitting up a textual data into words) using the nltk.tokenize package. During this step, we also removed all the stopwords, special characters, and the all the duplicates among the tokens.

We have also tokenized the whole textual data from the source file without removing the stopwords and special characters. We need this content in order to easily build up the snippets of the search results which are a part of the real content, and not the tokenized text.

The pre-processing output is a dictionary which contains 1416 keys denoting the source file names. The structure of our pre-processed corpus looks like this:

```
{
     "<inputFileName1>": {
         "tokens": ["<token1>", "<token2>", ... , "<tokenN>"],
         "content": ["<word1>", "<word2>", ... , "<wordN>"]
    },
   "<inputFileName2>": {
         "tokens": ["<token1>", "<token2>", ... , "<tokenN>"],
         "content": ["<word1>", "<word2>", ... , "<wordN>"]
    },
    ...,
     "<inputFileNameN>": {
         "tokens": ["<token1>", "<token2>", ... , "<tokenN>"],
         "content": ["<word1>", "<word2>", ... , "<wordN>"]
    }
}
```

3 Index building

To be able to benchmark the performance of the indices, we built two different indices: *Inverted Index* and *Sequential Index*. In the following subsections we describe how we are doing that.

3.1 Inverted Index

In our implementation of Inverted index, we used a database in order to simulate the inverted index structure. The database structure is consisted of three tables:

`IndexWord` consists of all the words indexed from the documents in the corpus, i.e. our dictionary.

`Posting` consists of a word from *IndexWord*, a document name in which the specific it appears, the frequency of appearance in the document, and the indices where the word appears in the source document.

`Existing` consists of a column `doesExist` that we have added in order to store a single boolean value indicating whether the inverted index has been built. We introduced this to know when we need to perform a re-initialization of the index because that is a costly operation.

In order to construct the *Inverted Index* out of a given pre-processed corpus of words for each file, we have constructed dictionary data structure. It allows us while iterating through the pre-processed corpus of words to store a word out of a given set of words as a key whose values are number of occurrences of a word in each iterated file along with its frequency and indexes of occurrences.

Next step in our implementation is storing the constructed Inverted index in the database. First, we store the list of the keys from the *Inverted Index* dictionary which represent the set of all unique words from a corpus in the table IndexWord, then we construct a list which holds all postings in the format
[(`word, documentName, frequency, indexes`)].

3.2 Sequential index

The *Sequential Index* does not require a special procedure of preparing in order to perform the search. We simply use the pre-processed data to perform search on.

4 Data retrieval

The data retrieval process (search) is performed on a user provided query which is first pre-processed, to get it in the same form as the rest of the corpus, i.e. it is tokenized. Afterward we are performing the search based on which index has been chosen by the user (either sequential or inverted). Both methods return the results in the same format: a list of tuples containing the cumulative *frequency* per document, the *document* name, and the aggregated *indices* of all results.

4.1 Inverted Index

Search the *Inverted Index* is performed with a single query on the database. The query is shown in Listing **??**, and it is an excerpt of the *Python* code which is performing the query.

```
SELECT documentName, sum(frequency) as freq, group_concat(indexes)
FROM Posting
-- the following part is filled by Python based on
-- the length of the tokenized query
WHERE word IN ({','.join(['?']*len(query))})
GROUP BY documentName
ORDER BY freq DESC
```

The query groups together the postings which match the words in the tokenized query, sums up the frequencies from the corresponding files, and aggregates the indices from the source document. The end result is a list of tuples, as previously described.

4.2 Sequential Index

The search for the *Sequential Index* is really simple: We get the pre-processed corpus and we iterate throughout the tokenized file content to get all the matches per file, and keep the track of the matches.

5 Implementation details

Some further implementation details worth noting:

Output printing Each row from the obtained result is printed in a table as provided in the instructions. The printing is provided by texttable.

Snippets The snippets mark the queried word with * symbols, and include up to 3 words to the left and to the right of the result.

REPL mode It is possible to enter in an interactive mode where the user is able to: *query* both types of indices, *change* the type of index being queried, force a *recreation* of the index and change the *number of results* printed in the table.

6 Analysis

7 Conclusion

Throughout the paper we introduced our implementation of building inverted index and query against it. Key differences between naive approach of sequential file reading and inverted index are noted supporting with experiments and results which prove how much more efficient the inverted index approach is, which is the reason why that concept is widely used nowadays even though Inverted Index takes much time in constructing the structure it allows very efficient querying.