WikiWise: Intelligent Search Solutions for Wikipedia
*Group*0
CENG 596 Final Report

Burak Eren Dere
e209892@metu.edu.tr

Özlem Demirtaş Altıntaş
ozlem.demirtas@metu.edu.tr

Abstract

In this project, we aimed to develop a specialized search engine tailored for retrieving and ranking Wikipedia pages based on their relevance to user queries. We have preprocessed the 96GB recent wikipedia dump and created an inverted index. After creating the inverted index, we have implemented a Query Runner that processes the queries and ranks them. During the development process, we have faced difficulties with the size of the dump, so we have made some adjustments on the predetermined roadmap.

Implementation: The code and other supplementary material (i.e. data, documentation etc.) are available at < WikiWise, Intelligent Search Solutions for Wikipedia >.

1 Introduction

With the expansion of digital content on the internet, people needed to implement effective tools in order to retrieve information in a fast and a scalable way. In today's world, Wikipedia is one of the largest and most comprehensive repositories of structured knowledge, which is a perfect database for testing information retrieval tools. In this project, we have implemented a search engine that gets queries from the user and gives the relevant data directly from the recent data dump of Wikipedia. The features that we have implemented are:

- Text Processing: The engine utilizes language processing techniques and analyzes and understands Wikipedia articles. We do tokenization, stemming and reduce words to their base or root form.
- Feature Extraction: By using different fields in the file data coming from Wikipedia, we get parts of the document like title, body and references and treat them differently.
- Indexing: We create separated indexes in different files at first, and then merge them together to create the full inverted index.
- Ranking and Scoring: The scoring system is developed using term frequencies and inverse document frequencies and also taking weights from the field map weights which is explained in later sections.
- Evaluation: We have compared the effectivenes of this system with Google's search results.

In this project report, we will introduce the system architecture and give information about the structure of the project and how different modules are connected to eachother. After that, we will give information about the effectiveness of the system and how what are the results of our evaluation. Lastly, we will conclude the project report and explain what we have learned and what difficulties that we have faced.

2 Related Work

The development of search engines tailored to specific datasets like Wikipedia involves integrating various information retrieval (IR) techniques, including text processing, link analysis, feature extraction, and sophisticated ranking algorithms. Here we outline notable works and technologies that underpin our approach and highlight advancements relevant to our project.

Significant advancements in text processing, particularly through natural language processing (NLP), play a critical role in content analysis and understanding. A fundamental technique in IR, as discussed by Manning, Raghavan, and Schütze (2008) in their book "Introduction to Information Retrieval", involves tokenization, stemming, and lemmatization, which are essential for reducing words to their base or root forms to improve search relevance and performance.

The application of link analysis algorithms such as PageRank, initially developed by Page et al. (1999) for Google's search algorithm, helps in assessing the importance and relevance of Wikipedia pages. This methodology is particularly pertinent to Wikipedia due to its rich internal hyperlinking structure, which can be leveraged to determine the importance of articles.

Feature extraction plays a vital role in enhancing the indexing process. Work by Baeza-Yates and Ribeiro-Neto (2011) in their text "Modern Information Retrieval" details various strategies for metadata and link extraction that can enrich information indices. Utilizing tools like Lucene for creating efficient indexes, as described by McCandless, Hatcher, and Gospodnetic (2010) in "Lucene in Action", aligns with our approach to optimize the retrieval process.

Our ranking mechanism, which incorporates both textual similarity and link analysis, is inspired by the vector space model and enhancements such as the BM25 algorithm, which is widely recognized for its effectiveness in information retrieval tasks. Recent innovations in neural information

retrieval, as outlined by Mitra and Craswell (2019) in their review on neural models for information retrieval, provide a foundation for integrating machine learning techniques like word embeddings and BERT to improve ranking accuracy.

Evaluating our search engine against established benchmarks like Google's results for Wikipedia-specific queries offers a direct measure of effectiveness and accuracy. Research by Voorhees and Harman (2005) on the TREC experiment provides methodologies for structured evaluation using precision-recall metrics, which we adopt for assessing our system.

3 System Architecture

In this section, we will discuss the architecture of our system. In the overall picture;

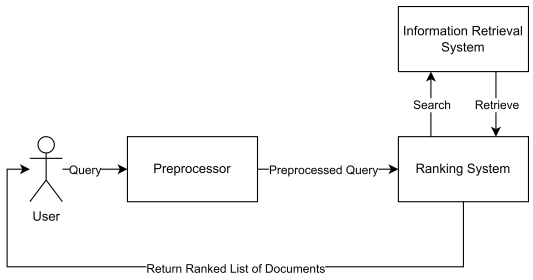

Figure 1. Overall System Architecture

- User first enters a query. This query can be a query can have two different formats. The first format is bag of words approach. In the second format, we give query in such way that for different parts of the wikipedia data, we search for different set of words. This approach is explained later.
- The preprocessor takes this query and modifies it by stemming, tokenization, removing non ascii characters and html tags.
- After that part, preprocessed query is sent to the ranking system. Ranking system retrieves documents from the inverted index and ranks them according to the preprocessed query.
- Lastly, the resulting list of relevant documents are sent back to the user.

In later parts of this section;

- How we preprocess the data
- Information Retrieval System
- Ranking System

Components and their subcomponents are being explained.

3.1 Data Preprocessing

We use data preprocessing in both running the query and building the inverted index. The steps are:

- We use PyStemmer[2] for stemming the words. This is useful for grouping highly related words together.
- We get english stopwords from natural language toolkit [1] and use them inside the preprocessor.
- We have also utilized Re module in python which gives us an ability to invoke regular expressions. This way, we also get a list of html tags that needs to be removed from the data if necessary.

For the preprocessing, we have implemented a separate class that takes a **Stemmer**, **Html Tag List**, **List of Stop Words** inside its constructor. For preprocessing tokens and wikipedia dump, we have utilized methods that are written inside this class.

3.1.1 Preprocessing the Wikipedia Dump File.

The wikipedia dump xml file that we have used is approximately 96GB in size. It has all the articles in wikipedia written until year 2024. This corpus has over 6 million articles, it has over 4 billion words. In average, there are 668 words per article. We have used the same Preprocessing class for preprocessing the fields in wikipedia dump file.

3.2 Information Retrieval System

In the progress report, we were planning to use Apache Lucene as our core information retrieval toolkit and integrate BM25 scoring algorithm. However, processing the data consumed too much of our time and we could not integrate the wikipedia dump file we have at hand to these systems. As a result, we have implemented our own information retrieval system and we have written an Indexer component for it. In the following part, we explain how we have read the wikipedia dump file, how we create and store indexes.

The dump file has the information of article ids, titles, categories, text bodies as an xml file. In order to parse this xml file, we have used xml sax library [4] in python.

3.2.1 Handling the XML.

The following figure shows the relationship between xml.sax, XMLParser, PageProcessor and IndexCreator classes.

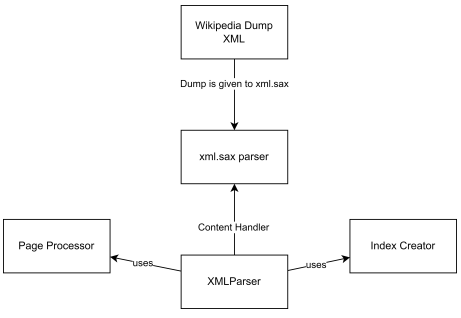

Figure 2. XML Handling

The xml.sax class requires us to write an XML parser class for parsing the related fields in the XML file. This is called a **content handler**. In our case, we have named our **content handler** as **XML Parser**. XML Parser class used

two additional classes for processing pages and creating indexes. These are **Page Processor** and **Index Creator** and they are being explained in further subsections.

3.2.2 Global Variables. Before explaining how we parse the xml and retrieve the data in more depth, we need to explain the global variables we use in the system. These variables are all used by different components of the system while the program is running. We keep these inside a class called **Settings**. In Settings, we have:

- **Output Path:** Path where the index files are going to be created.
- **Id Title Map:** This is a dictionary where the keys are the ids of pages and values are the lowercased titles.
- **Index Map:** This is also a dictionary where we keep track of the frequencies of words. In this dictionary, keys are unique words that we encounter during parsing and values represent frequencies for each document and their respective parts. This will be explained further while explaining the **Index Creator** class.
- **File Number:** Keeping track of how many files we have.
- **PageNumber:** Keeping track of how many pages we have.

3.2.3 Page Processor. As explained above, our **XML Parser** class uses utilities from the **Page Processor** class. This class is responsible of extracting:

- Title
- Body
- Category
- Infobox
- Link
- Reference

Fields while reading the xml file. It has a utility function named **Page Process**. This function reads the title and the text acquired from the xml parser and extracts the parts explained above. After that, extracted parts are given to the **Index Creator**.

3.2.4 Index Creator. Class has a utility function named **Index**, this function takes the information extracted from **Page Processor** as explained above and updates the global **Index Map** dictionary. The keys are unique words and values are postings. The format of the postings are like the following:

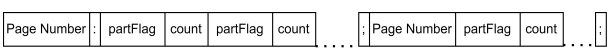

Figure 3. Index Map Structure

For example, let's think that we are processing the Wikipedia XML dump and we are at the 30th page. So, our indexer is actually populating the **Index Map** dictionary with the words they encounter. Let's consider the word **birth** and it has the following posting:

- 2:b1c1i3;13:c1i1;26:b1c1i3;28:b1c1i3

This posting explains that, page 2 has the word **birth**:

- 1 times in the body
- 1 times in the category
- 3 times in the infobox

After the symbol **;** we go to the next page which is the 13th page and it has the word **birth**

- 1 times in the category
- 1 times in the infobox

The logic is the same for pages 26 and 28 as well.

3.2.5 Writing Data inside Index Creator. The data we are processing is huge, so in order to use the memory efficiently, we have to make intermediate index files and save up the memory space for the upcoming pages. For that, we have implemented a class named **DataWriter**, this class is responsible of writing intermediate index files and the id title map. So, in each 40.000th page we do the following:

- Call DataWriter's **IntermediateIndex** function
- Call DataWriter's **IdTitleMap** function
- Clear global **indexMap** variable
- Clear global **idTitleMap** variable

Now we explain the two methods of **DataWriter** class in detail:

- **IdTitleMap():** This function gets the global **Id Title Map** variable and appends it to a file on the output path named **id_title_map.txt**.
- **IntermediateIndex():** This function gets the dictionary **IndexMap** and sorts all the elements according to the key values which are the words. So, we get a list that has tuples of words and its postings and the list sorted according to words' alphabetical order. We append the word with its postings by putting a - symbol between them and write them as an intermediate index file. For example:
  - **Word:** birth
  - **Posting:** 2:b1c1i3;13:c1i1;26:b1c1i3;28:b1c1i3
  - **File Entry:** birth-2:b1c1i3;13:c1i1;26:b1c1i3;28:b1c1i3

After writing the intermediate index files, we need to merge them together. For that, we have created **FileMerger** class.

3.2.6 File Merger class. This class has a method named **Merge** and its job is to merge the index files that we have created before. It basically reads the intermediate index files and updates the postings. The file merger class also creates a separate file for each 30.000th posting. For writing the merged data, it uses **WriteFiles** function from the **DataWriter** class. We also create a secondary index for calculating total term frequencies and and which lines and parts those terms appear. For example for the word **zim**:

- **Postings:** 18140:b1i6;23560:b1;
- **Secondary Index:** zim - 5 - 2 - - 4 - -1- - -

The format of the secondary index first starts with the word. The second term gives the total frequency of this word inside the Wikipedia Data dump. The third term gives which one of the final index file this word exists. The other numbers between the - symbols represent in which line of the corresponding parts' index file the word exists. This way, we can easily find information about the words while ranking the queries.

At the end, we also write the number of files and other specs such as token count and page count to their corresponding txt files. Because we need these informations as well while ranking the documents.

The figure below shows the overall picture of the creation of index files as explained above in this subsection:

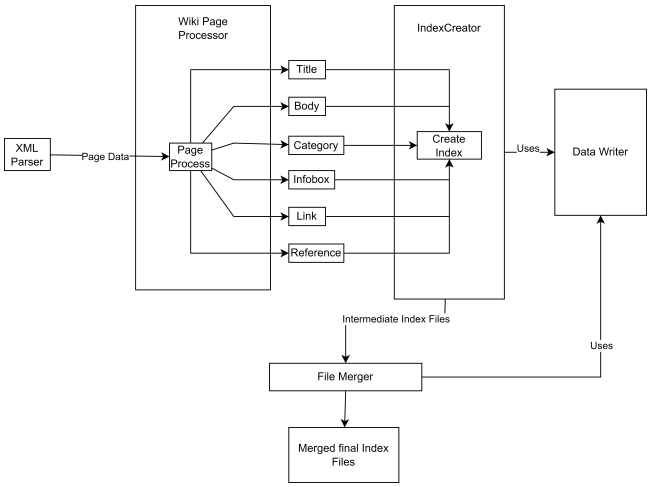

Figure 4. Overall Inverted Index creation

3.3 Query Processing and Ranking System

After creating the indexes from our Wikipedia Data Dump xml file, we can use the information that we have gathered to evaluate some queries. The program we have implemented works as follows:

- We first get the english stopwords
- We get the html tags
- Finally, we get the stemmer

Later, we read how many pages we have from its corresponding txt file and record the value. Just like in the indexing part of the system, we have several new classes that we use in the QueryRunner class which runs the queries and gives us the results:

- **File Traverser:** This class has several methods that we use, these are:
  - **Binary Searcher:** In this method, we read token info from corresponding files. For example, if we are processing a token in the query named **Harry**, after preprocessing the word we got **Harri** and we begin searching for the information about this word. We do it by invoking **Binary Searcher** method. It yields which documents have this word.
  - **Get Token Info:** This method uses **Binary Searcher** explained above by giving corresponding file path and the word to that function and yields the information given by the **Binary Searcher**.
  - **Title Search:** We use this to get the title of the page. We read it from the file that we keep idTitle map which was a global variable in the indexer.
  - **Search Field File:** This method gets a field, file number and line number and gives us the posting of that word for that field. For example, for:
    - field: 'title'
    - file num: '2'
    - line num: 3710

    We get all the postings for titles in the second file and it exists in line number 3710. We easily get this line and extract the posting.
- **Query Result Getter:** This class particularly uses the **File Traverser** class that has been explained above. It has two main methods for two different query types we support. These query types are:
  - **Simple Query:** We give a bag of words as the query. For example 'Alexander the Great'. For each token, we read the indexes for each field like title, body, category etc. and get the page frequencies for each token in the query and return this result.
  - **Field Query:** With this query type, we specifically denote in which fields we want to search for that token. We put the first letter of that field and a **:** symbol before each token. For example;
    - Query: 't:Star b:Solar', we search 'Star' in titles and 'Solar' in the bodies.
- **Run Query:** This class is responsible of running the queries. It is initialized by giving objects of **File Traverser**, **Ranker**, **Query Result Getter** classes. After that, we run the query using the **User Input** function. This function preprocesses the query and gives it to **Query Results** method. This method gets the page frequencies and postings using **Query Result Getter**'s methods and then calculates the ranks for the documents. At last, we get a list of ranked documents.
- **Ranker:** After acquiring the page frequencies and page postings, the information is given to the **Ranker**'s **Rank** function. Inside the **Rank** function, we hold weights for each field. In our case we have:
  - **Title:** 1.0
  - **Body:** 0.85
  - **Category:** 0.3
  - **Infobox:** 0.65
  - **Link:** 0.15

– **Reference:** 0.15

For each document, we calculate the rank for a specific field with the following formula:

$$rank_f = \sum_{t \in T} \sum_{p \in P} \frac{w_f * (1 + log(p)) * log(N - pF(t))}{pF(t)} \quad (1)$$

In this formula:
– T stands for all the tokens in the query
– P stands for all the posts for that token
– d stands for the document we work on
– $pF(t)$ is the page frequency of that token
– f is the field
– w_f is the weight for that field

So, the final rank of the document is calculated by:

$$r = \sum_{f \in F} rank_f \quad (2)$$

So, in this formula, f is the field name.

This way, we get the ranked list of documents. The figure below shows the big picture how we get the ranked list of results back to the user.

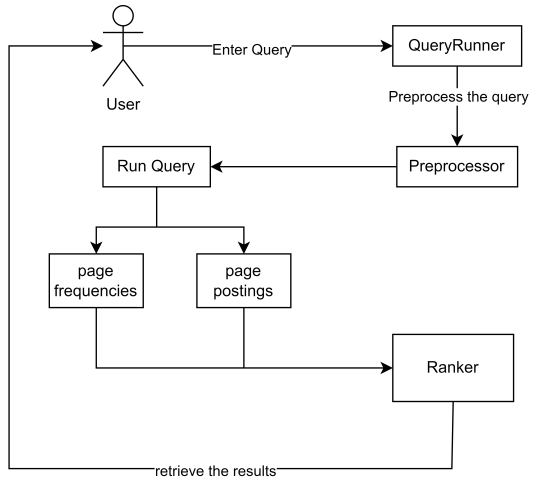

Figure 5. Query Processing and Ranking, detailed big picture

3.4 User Interface

Despite we were short in time because of the size of our data, we could not specificaly build a user interface. However, we provide a **jupyter notebook** for evaluating the results as a user interface. We support simple queries as bag of words and field queries as explained in above sections. The query runner can be started by running the first code cell and the 3th code cell.We can also run the Indexer in the 2nd code cell if we do not have indexes. For demonstration, time we have also indexed a small xml file that is nearly 97MB's.

For firing a query, we run the Query runner and give how many documents we want to see.

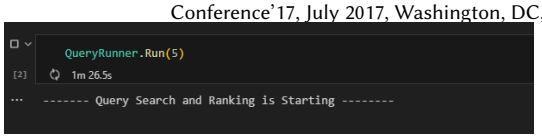

Figure 6. Running the Query from the cell

After running this cell, the program wants inputs from us. For example, lets write 'Alexander the Great':

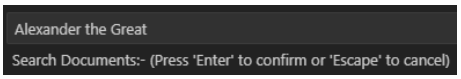

Figure 7. Running Alexander the Great

We get the following result (highest ranked 5 document IDs and their titles):

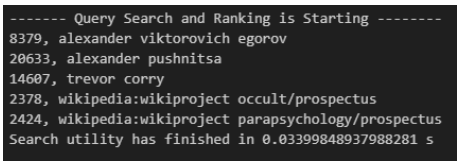

Figure 8. Running Alexander the Great result

Here, as we can see, we could not get what we have expected at first. It is because we are using the tiny dataset we have at hand so the ranks of these documents are not very high. With the actual 96GB wikipedia dump, we get more accurate results. The results with the actual data is shared in the **Evaluation** section.

4 Evaluation

First of all, as said above, we have developed the system using 96GB wikipedia dump xml file. But debugging with this file was tedious. For this, we have also introduced an approximately 97MB tiny xml file and despite we were not sure that we were getting actual ranks and highly related documents, it helped us to evaluate the Ranking system before testing the actual data. In the above section, we can see the result coming from the tiny data. Now, let's fire the query 'Alexander the Great' again but this time use the indexes created from the actual data..

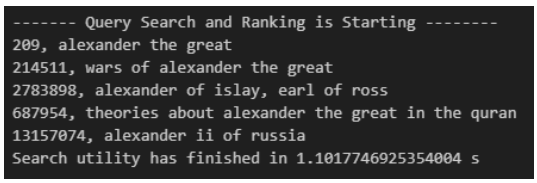

Figure 9. Running Alexander the Great result from actual wikipedia dump

As we can see, this time, we get more accurate results for the Query. If we look at the time we spend searching for the query terms, we see that there is not much difference thanks to the secondary indexes and the binary search algorithm of the **File Traverser**.

4.1 Specs of the Computer and Data

In this section, we give the specs of the computer that we have used to test the implementation and some formal information about the Data we have used for index creation. The computer has:

- 64 GB RAM
- Intel(R) Core(TM) i9-14900K 3.20 GHz processor
- 64 bit Operating System (Windows)

The data we have used are:

- **Large Data:** This is the recent (2024) data dump of english wikipedia:
  - Aricles: Over 6.500.00
  - Words: Over 4.000.000.000
  - Words Per Article: 668
  - Size: 96 GB
- **Small Data:** This is a tiny version of english wikipedia:
  - Articles: 28.694
  - Words: 150.412
  - Words Per Article: Similar to the Large data
  - Size: 97 MB

4.2 Time complexity of Searching

The table below gives the average time spent searching for large and small datasets using different sized queries (1, 10, 20):

Table 1. Running Times of Searching

Database	1 Token	10 Tokens	20 Tokens
Tiny Dump	**0.003 s**	**0.006 s**	**0.007 s**
Large Dump	**2.45 s**	**8.20 s**	**8.94 s**

From the table above, we can see that tiny dump caps at 0.007 seconds and large dumps is around 9 seconds as we increase the words in the query. This performance can further be increased utilizing parallel processing.

On the other hand, index creation for small dump lasts **33 seconds**, but successfuly creating an index for the large dump lasted more than **2 days**.

4.3 Usage of field queries

As explained above, we can use field queries for getting more relevant documents. For example, when we fire the simple query 'rings saruman' we get:

```
6108290, saruman
652225, the lord of the rings: the third age
7362174, the lord of the rings: aragorn's quest
1377024, the lord of the rings: the two towers (video game)
1071210, the lord of the rings: the fellowship of the ring (video game)
Search utility has finished in 0.07399964332580566 s
```

Figure 10. Running 'rings saruman'

And if we write the same query but as a field query, specifying that we want to see 'rings' in the infobox field and 'saruman' in the body field. We get this ranking:

```
6108290, saruman
652225, the lord of the rings: the third age
7362174, the lord of the rings: aragorn's quest
139636, isengard
1377024, the lord of the rings: the two towers (video game)
Search utility has finished in 0.005998849868774414 s
```

Figure 11. Running 'i:rings b:saruman'

We start getting articles more affiliated with the name saruman because in that articles, we see more 'saruman' words and that is what we are looking for. In addition, the results may very according to the weights we give to each field. The weights were explained in above sections.

4.4 Evaluating the Relevance

We have evaluated the relevance of 20 simple queries with 5 results by visually comparing the results with the first pages of the search utility [3] of wikipedia. From those queries, we got 100 resulting documents. Among those documents, if we evaulate each query with their result, we got all of them in the first pages of documents and the highest rank documents were always in the first page.

5 Conclusion

In this system, we have implemented an indexer that processes wikipedia dump files and builds an inverted index for each field in the xml file. And then, our system can get simple or field queries and return the relevant documents. While writing the progress report, we were sure that the project was going to be on time. But since we have lost so many time from trying to create an index from, we could not implement BM25 algorithm or use Lucene as we said. We can further improve our system by implementing BM25 algorithm, relevance feedback and wildcard support and if we had more time, we would definitely implement them as well. To sum up, this project was very helpful for us to understand the course material more and we will try to tackle the problems we could not handle as a future study.

References

[1] [n.d.]. Nltk. https://www.nltk.org. Accessed: 2024-06-03.
[2] [n.d.]. PyStemmer. https://pypi.org/project/PyStemmer. Accessed: 2024-06-03.
[3] [n.d.]. Wikipedia Search. https://en.wikipedia.org/w/index.php?fulltext=1&search. Accessed: 2024-06-03.
[4] [n.d.]. XmlSax. https://docs.python.org/3/library/xml.sax.html. Accessed: 2024-06-03.