Project Report: Building an Inverted Index Search System

Malay Agarwal MT2022059
Umair Ahmad Beig MT2022126
Sami Ullah Naikoo MT2022099

May 15, 2023

Abstract

In this project report, we present the development of an Inverted Index Search System. The system is designed to efficiently index and search a large collection of documents, providing users with quick and relevant search results. We discuss the key components of the system, including data preprocessing, index construction, and search algorithms. Finally, we discuss potential improvements and future work to enhance the system's functionality and performance. Overall, this project report serves as a guide to understanding and implementing an Inverted Index Search System.

1 Introduction

1.1 Problem Definition

The purpose of this project is to develop a search system based on an inverted index for efficient word retrieval in multiple datasets. The system should allow users to enter a word and select a dataset, and it should return the documents in which the word appears.

In addition to the search functionality, the project aims to implement the following features:

- Dataset preprocessing to extract relevant information and generate the inverted index.
- Integration with MongoDB for efficient storage and retrieval of search results.
- Development of a user-friendly frontend using React.
- Implementation of a backend API using Spring Boot to handle search requests and interact with the database.

1.2 Approach

To achieve the project objectives, the following approach was followed:

1. Dataset Preprocessing: The three datasets were processed to extract the necessary information and generate the inverted index. This involved tokenization and removing stop words.
2. MapReduce Job: A MapReduce job was implemented using Hadoop to calculate the inverted index of words in each dataset and generate three output files. This job distributed the processing across multiple nodes for scalability and efficiency.
3. MongoDB Integration: The output files generated from the MapReduce job were stored in a MongoDB database. The database schema was designed to store the inverted index and related information efficiently for quick retrieval.
4. Frontend Development: A user-friendly frontend was developed using React. The frontend

provided an interface for users to enter search queries, select the dataset, and display the search results.

5. Backend Development: A backend API was implemented using Spring Boot. The API handled incoming search requests, interacted with the MongoDB database, and returned the search results to the frontend.

2 Implementation Details

In this section, we provide detailed information about the implementation of each component.

2.1 Dataset Preprocessing

The dataset preprocessing step involves tokenizing the documents, stemming the words, and removing stop words. Tokenization splits the documents into individual words. Stop words, such as common articles and prepositions, are removed to reduce noise in the inverted index. Here's an overview of what happens:

- Tokenization: - The map() method receives a document as a Text object. - The document is converted to a string using value.toString(). - The string is tokenized into individual words using a StringTokenizer on the text variable.

- Stop Word Removal: - The stopWords set contains the stop words that are loaded from a file in the setup() method. - During tokenization, each token is checked against the stopWords set using stopWords.contains(token). - If a token is not a stop word, it is further processed and emitted as a key-value pair.

3 MapReduce Job

The MapReduce job was implemented using Hadoop to calculate the inverted index of words in each dataset. The job involved two main components: the mapper and the reducer.

3.0.1 Mapper

The mapper class, `InvertedIndexMapper`, extends the `Mapper` class provided by Hadoop. It processes the input key-value pairs and emits intermediate key-value pairs.

The `map` method is overridden to perform the mapping operation. It takes an input key-value pair and extracts the document ID and text. Then, it tokenizes the text and processes each token.

Within the `map` method: - The document ID is extracted from the input key. - The text is tokenized using a `StringTokenizer`. - For each token: - The token is assigned to the `word` variable of type `Text`. - The document ID and token count are stored in the `docMap` variable of type `MapWritable`. - The `word` and `docMap` are emitted as the intermediate key-value pair using the `context.write()` method.

The output of the mapper is the intermediate key-value pairs, where the key is a word and the value is a `MapWritable` containing the document ID and token count.

3.0.2 Reducer

The reducer class, `InvertedIndexReducer`, extends the `Reducer` class provided by Hadoop. It processes the intermediate key-value pairs and emits the final key-value pairs.

The `reduce` method is overridden to perform the reducing operation. It receives a word and an iterable of `MapWritable` values. The values are combined to calculate the total token count per document.

Within the `reduce` method: - A `Map` called `docCount` is created to store the document IDs and their respective token counts. - For each `MapWritable` value in the input values: - For each entry in the `MapWritable`: - The document ID and token count are extracted. - If the document ID exists in `docCount`, the token count is added to the existing count. Otherwise, a new entry is created. - The `docCount` is used to populate the `result` variable of type `MapWritable`. - The final key-value pair, with the word as the key and the `result` as the value, is emitted using the `context.write()` method.

The output of the reducer is the final key-value

pairs, where the key is a word and the value is a `MapWritable` containing the document IDs and their respective token counts.

4 Frontend Development

The frontend was developed using React to provide a user-friendly interface for search queries and displaying the search results. The frontend consisted of components such as a search form, dropdown menu for dataset selection, and a table to display the search results.

The following code snippet shows an example of the search form component in React:

```
import React, { useState } from 'react';

const SearchForm = () => {
const [word, setWord] = useState('');
const [selectedDataset, setSelectedDataset]
= useState('');

const handleWordChange = (event) => {
setWord(event.target.value);
};

const handleDatasetChange = (event) => {
setSelectedDataset(event.target.value);
};

const handleSubmit = (event) => {
event.preventDefault();

arduino
Copy code
// Perform search query using backend API
// Display the search results
};

return (
<form onSubmit={handleSubmit}>
<input type="text" value={word} onChange=
{handleWordChange} placeholder=
"Enter word" />
<select value={selectedDataset} onChange={
handleDatasetChange}>
<option value="">Select Dataset</option>
<option value="dataset1">Dataset 1</option>
<option value="dataset2">Dataset 2</option>
<option value="dataset3">Dataset 3</option>
</select>
<button type="submit">Search</button>
</form>
);
};
```

5 Backend Development

The backend API was implemented using Spring Boot to handle search requests, interact with the MongoDB database, and return the search results to the frontend.

5.0.1 SearchController

The `SearchController` class is responsible for handling search requests and interacting with the inverted index service. It defines an API endpoint `/api/search/word/dataset` that receives the search word and dataset as path variables. The simplified code snippet is as follows:

```
\begin{minted}[breaklines]{java}
@RestController
@RequestMapping("/api")
public class SearchController {

less
Copy code
@Autowired
private InvertedIndexService invertedIndexService;

@GetMapping("/search/{word}/{dataset}")
public List<Document> search(@PathVariable
String word, @PathVariable String dataset) {
    // Perform search query using the
    //inverted index service
    // Retrieve the documents where the word
    //appears in the specified dataset
    // Return the search results
}
```

```
}
\end{minted}
```

Within the `search` method, you would integrate the inverted index service to execute the search query, retrieve the relevant documents, and return the search results to the frontend.

5.0.2 FileController

The `FileController` class is responsible for handling requests related to accessing files stored in HDFS. It defines an API endpoint `/api/open/filePath/selectedValue` that receives the file path and selected value as path variables. The code snippet below demonstrates the key functionality:

```
\begin{minted}[breaklines]{java}
@RequestMapping("/api")
public class FileController {

less
Copy code
@Autowired
private Configuration hadoopConfiguration;

@CrossOrigin(origins="http://localhost:3000")
@GetMapping("/open/{filePath}/{seldataset}")
public ResponseEntity<Resource>
displayTextFile(
@PathVariable String filePath, @PathVariable
String selectedValue) throws IOException {
    // Access HDFS using the Configuration
    & retrieve the file at the specified path
    // Return the file as a response entity,
    setting appropriate headers
}
}
\end{minted}
```

In the `displayTextFile` method, you would use the `hadoopConfiguration` to access HDFS, retrieve the file at the specified `filePath`, and return it as a response entity, setting the appropriate headers.

5.1 Results and Discussion

The implemented inverted index search system successfully achieved the objectives of efficient word retrieval in multiple datasets. Users can enter a word and select a dataset, and the system returns the documents in which the word appears. The frontend provides a user-friendly interface for search queries, and the backend handles the search requests and interacts with the MongoDB database effectively.

The system demonstrated efficient search functionality by utilizing the inverted index and leveraging the MapReduce framework for distributed processing. The dataset preprocessing step, including tokenization, stemming, and stop word removal, improved search accuracy. MongoDB integration allowed for efficient storage and retrieval of search results.

The frontend, developed using React, provided an intuitive user interface with a search form and dataset selection dropdown. The backend, implemented using Spring Boot, handled search requests, interacted with the MongoDB database, and returned search results to the frontend.

Overall, the project achieved its goals of developing an inverted index search system and integrating it with frontend and backend components. The system can be further enhanced with additional features such as advanced search options, relevancy ranking, and scalability improvements.

6 Conclusion

In this project, we have implemented an inverted index using Hadoop MapReduce. The inverted index is a fundamental data structure for efficient full-text search and information retrieval. By processing a collection of documents, we have generated an index that maps each unique word to the documents that contain it, along with the frequency of occurrence.

The MapReduce framework provides a scalable and distributed approach to process large datasets in parallel, making it suitable for handling big data scenarios. By leveraging the parallel processing capabilities of MapReduce, we were able to efficiently generate

the inverted index for a large collection of documents.

6.1 Improvements and Further Work

While our implementation successfully builds the inverted index, there are several areas that can be improved and further expanded:

- **Advanced Text Processing Techniques:** Currently, our implementation performs basic tokenization and stop word removal. To enhance the quality of the index, additional text processing techniques can be incorporated, such as stemming, lemmatization, or part-of-speech tagging. These techniques can help improve search accuracy by reducing words to their base form or considering the grammatical context.

- **Handling Multilingual Documents:** The current implementation assumes English documents. To handle multilingual documents, language identification techniques can be integrated to determine the language of each document and apply language-specific preprocessing accordingly. This would enable the creation of separate inverted indexes for different languages.

- **Index Compression and Optimization:** As the size of the document collection increases, the inverted index can become quite large. Techniques such as compression algorithms and index partitioning can be employed to reduce the storage requirements and enhance query performance.

- **Integration with Search Engine:** The inverted index generated through MapReduce can serve as the backbone for building a search engine. By integrating the inverted index with a search engine framework, users can perform efficient keyword searches across the document collection.

- **Enhanced Query Capabilities:** In addition to simple keyword searches, more advanced query capabilities can be implemented, such as phrase search, proximity search, or relevance ranking based on term frequency-inverse document frequency (TF-IDF) or other scoring algorithms.

By addressing these improvements and further extending the functionality, our inverted index implementation can be transformed into a more robust and versatile information retrieval system.

In conclusion, the MapReduce-based inverted index provides a scalable and efficient solution for indexing and searching large document collections. By building upon this foundation and incorporating advanced techniques, we can enhance its capabilities and create a powerful information retrieval system for a wide range of applications.

7 References

MapReduce: Simplified Data Processing on Large Clusters. Jeffrey Dean and Sanjay Ghemawat. Research paper. Available at: https://research.google/pubs/pub62/

Hadoop Official Documentation: Apache Hadoop i:Official documentation can be found at: https://hadoop.apache.org/documentation/

Spring Boot Official Documentation: : https://docs.spring.io/spring-boot/docs/current/reference/

React Official Documentation https://reactjs.org/docs/

"Information Retrieval: Implementing and Evaluating Search Engines" by Stefan Büttcher, Charles L. A. Clarke, and Gordon V. Cormack. Available at: https://www.oreilly.com/library/view/information-retrieval-implementing/9780262026512/

7.1 datasets

https://www.kaggle.com/datasets/hgultekin/bbcnewsarchive BBC news

https://github.com/soskek/bookcorpus Books

https://en.wikipedia.org/wiki/Wikipedia:Database$_{d}ownloadWiki$