INDEXER (Assignment-01)

Session: 2021 – 2025

Submitted by:

Muhammad Yaqoob 2021-CS-118

Supervised by:

Dr Syed Khaldoon Khurshid

Department of Computer Science

University of Engineering and Technology UET Lahore

Table of Contents

Table Of Figures

1. Introduction

Information retrieval is a critical component of modern search systems, enabling users to quickly locate relevant content from vast datasets. Whether in search engines, document management systems, or data analysis tools, the ability to index and query data plays a pivotal role. This assignment focuses on implementing an Index, a data structure commonly used in information retrieval to map terms to the documents in which they appear, along with the frequency of their occurrence.

2. Objective

There were two objectives for this assignment:

- **Word-based Searches:** To implement a search engine capable of finding all occurrences of a given word across multiple documents and returning the relevant document titles along with their term frequencies.
- **Title-based Searches:** To enable the search engine to locate a specific document based on its title from a given directory and retrieve its content efficiently.

3. Document Collection

Names of the documents collected are written below. Each document has content.

- Health and Fitness.txt
- Nature and Wildlife.txt
- Space Exploration.txt
- Technology and Programming.txt
- Travel and Adventure.txt

4. Indexing Approaches

There were two possible approaches for building the indexer:

- Forward Index
- Inverted Index

Aspect	Forward Index	Inverted Index
Definition	Maps a document ID to its contents, listing all the terms present in the document.	Maps terms to the list of document IDs where they occur, along with additional details (e.g., frequency).
Structure	Document ID → [Terms in the document]	Term → [Document IDs, frequencies, positions, etc.]
Storage	Requires less storage space	Consumes more storage
Time	O(n)	O (1)

I have chosen the inverted index approach for this assignment due to its:

- Fast search operations suitable for word-based and title-based queries.
- Scalability for handling large document collections.
- Ability to efficiently support the assignment's objectives, despite slightly higher insertion and storage costs compared to the forward index.

5. Steps to Create Index

- Metadata Extraction
- Case Folding
- Tokenization
- Remove Stopwords
- Creating Index

5.1. Meta Data Extraction

Extract important information such as document titles and unique identifiers (IDs). This ensures each document is uniquely identifiable in the indexing process.

5.1.1. Workflow

Text documents (individual or from a directory) are provided as input for indexing.

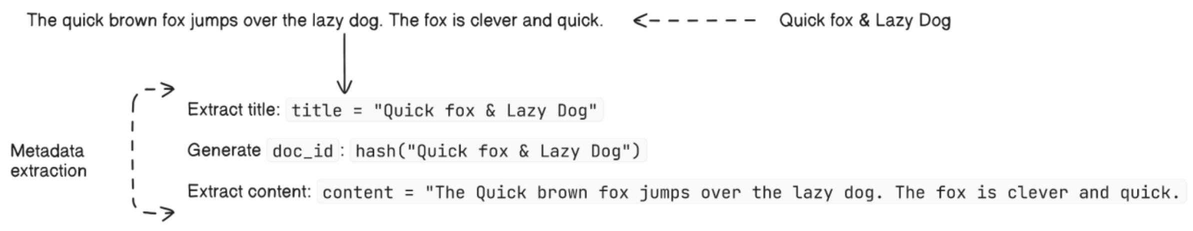

The quick brown fox jumps over the lazy dog. The fox is clever and quick.
Quick fox & Lazy Dog
Metadata extraction
Extract title: title = "Quick fox & Lazy Dog"
Generate doc_id : hash("Quick fox & Lazy Dog")
Extract content: content = "The Quick brown fox jumps over the lazy dog. The fox is clever and quick.

Figure 1: Metadata Extraction

5.1.2. Code

Code Reference:

add_single_document(file_path) processes a single document.

add_documents_from_directory(directory_path) processes multiple documents.

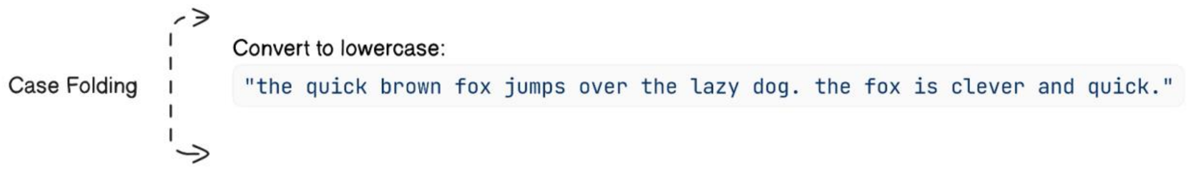
```python
def add_single_document(self, file_path):
    """Add a single document from a file."""
    with open(file_path, 'r', encoding='utf-8') as file:
        title = os.path.splitext(os.path.basename(file_path))[0]  # Extract title from file name
        content = file.read()
        doc_id = hash(file_path)  # Use a hash of the file path as a unique ID
        # Indexer
        self.create_indexer(doc_id, title, content)

def add_documents_from_directory(self, directory_path):
    """Adds all text files from a directory to the index."""
    for file_name in os.listdir(directory_path):
        if file_name.endswith(".txt"):
            file_path = os.path.join(directory_path, file_name)
            self.add_single_document(file_path)
```

5.2. Case Folding

Convert all text to lowercase to ensure uniformity during processing. For example, "Fox" and "fox" will be treated as the same term.

5.2.1. Workflow



```
"the quick brown fox jumps over the lazy dog. the fox is clever and quick."
```

Figure 2: Case Folding

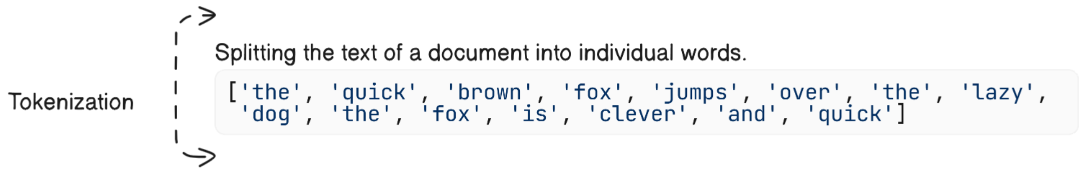
5.2.2. Code

```python
def _case_folding(self, text):
    """Convert text to lowercase."""
    return text.lower()
```

5.3. Tokenization

Split the document text into individual words or tokens. For instance, the sentence "The fox jumped over the dog" becomes tokens: ["the", "fox", "jumped", "over", "the", "dog"].

5.3.1. Workflow



```
['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy',
 'dog', 'the', 'fox', 'is', 'clever', 'and', 'quick']
```

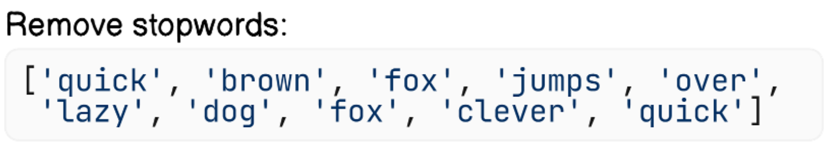
5.3.2. Code

```python
def _tokenization(self, text):
        """Split text into words."""
        return re.findall(r'\b\w+\b', text)
```

5.4. Remove Stopwords

Eliminate common and less meaningful words like "the," "is," and "and" to reduce index size and improve efficiency. This step retains only the significant terms.

5.4.1. Workflow



```
['quick', 'brown', 'fox', 'jumps', 'over',
 'lazy', 'dog', 'fox', 'clever', 'quick']
```

Figure 3: Remove Stopwords

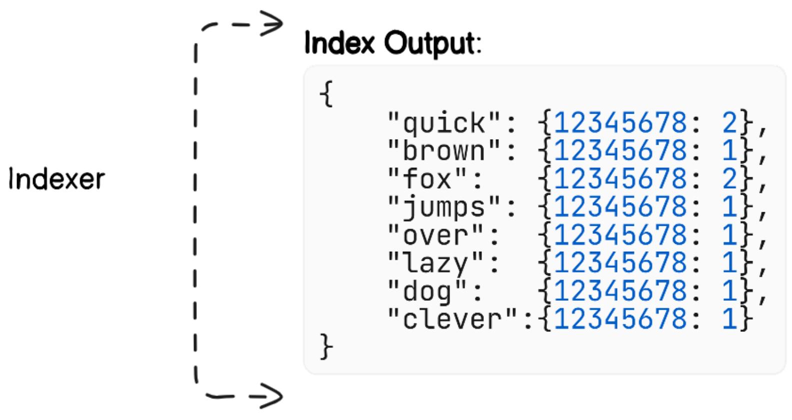
5.4.2. Code

```python
def _remove_stopwords(self, words):
     """Remove stopwords from a list of words."""
     return [word for word in words if word not in self.stopwords]
```

5.5. Creating Index

Map each token to the documents where it appears, along with its frequency and other details.

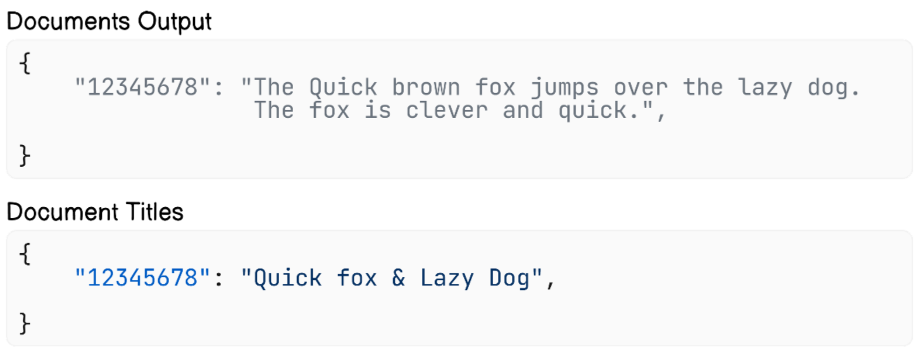
5.5.1. Workflow



```
{
    "quick": {12345678: 2},
    "brown": {12345678: 1},
    "fox":   {12345678: 2},
    "jumps": {12345678: 1},
    "over":  {12345678: 1},
    "lazy":  {12345678: 1},
    "dog":   {12345678: 1},
    "clever":{12345678: 1}
}
```

Figure 4: Creating Index

It also keeps track of documents and document titles with their Ids.



```
{
    "12345678": "The Quick brown fox jumps over the lazy dog.
                 The fox is clever and quick.",
}
```

```
{
    "12345678": "Quick fox & Lazy Dog",
}
```

Figure 5: Document IDs and Titles alongside Indexing

5.5.2. Code

```python
def create_indexer(self, doc_id, title, text):
    """Add or update a document in the index."""
    # CASE FOLDING AND TOKENIZATION
    tokens = self._tokenize(text)
    if doc_id in self.documents:
        self.remove_document(doc_id)

    self.documents[doc_id] = text
    self.document_titles[doc_id] = title

    # CREATING INDEXER
    for token in tokens:
        if token not in self.index:
            self.index[token] = {}
        if doc_id not in self.index[token]:
            self.index[token][doc_id] = 0
        self.index[token][doc_id] += 1
```

6. Word Based Search

The word-based search functionality allows users to retrieve a list of documents containing a specific word. This is accomplished by querying the index, which maps terms to the documents in which they appear.

6.1. Workflow

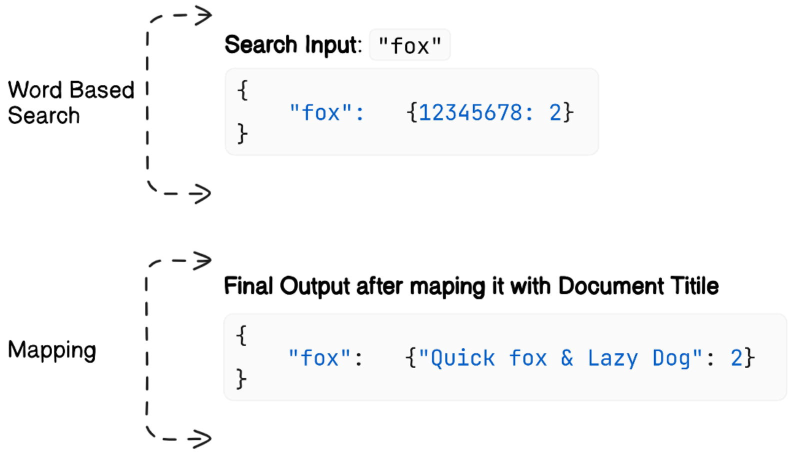
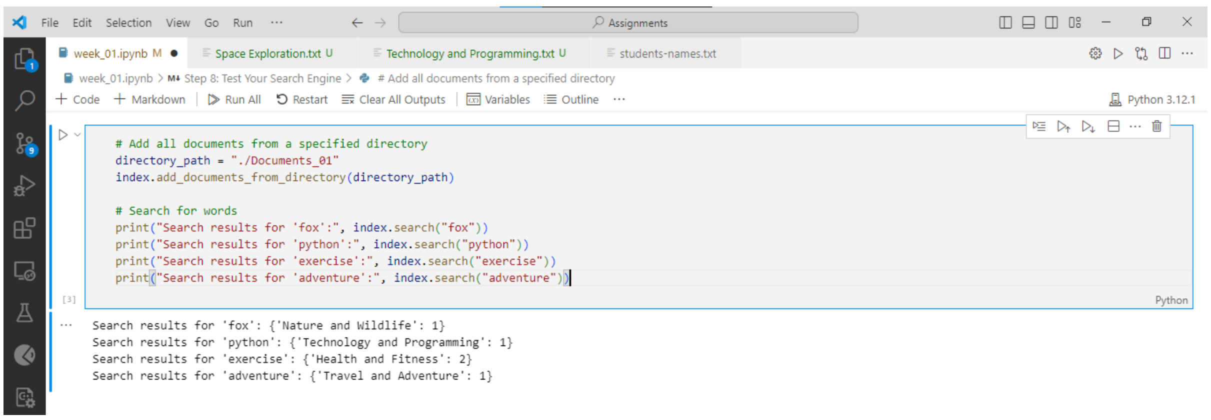

Word Based Search
Search Input: "fox"
{
"fox": {12345678: 2}
}
Mapping
Final Output after maping it with Document Titile
{
"fox": {"Quick fox & Lazy Dog": 2}
}

Figure 6: Word Based Search workflow

6.2. Code

```python
def search(self, word):
    """Search for a word in the index."""
    word = word.lower()
    if word in self.index:
        results = self.index[word]
        return {self.document_titles[doc_id]: count for doc_id, count in results.items()}
    return {}
```

6.3. Screenshot


File Edit Selection View Go Run ···
Assignments
week_01.ipynb M
Space Exploration.txt U
Technology and Programming.txt U
students-names.txt
week_01.ipynb > Step 8: Test Your Search Engine > # Add all documents from a specified directory
+ Code + Markdown Run All Restart Clear All Outputs Variables Outline
Python 3.12.1
Python

```python
# Add all documents from a specified directory
directory_path = "./Documents_01"
index.add_documents_from_directory(directory_path)

# Search for words
print("Search results for 'fox':", index.search("fox"))
print("Search results for 'python':", index.search("python"))
print("Search results for 'exercise':", index.search("exercise"))
print("Search results for 'adventure':", index.search("adventure"))
```

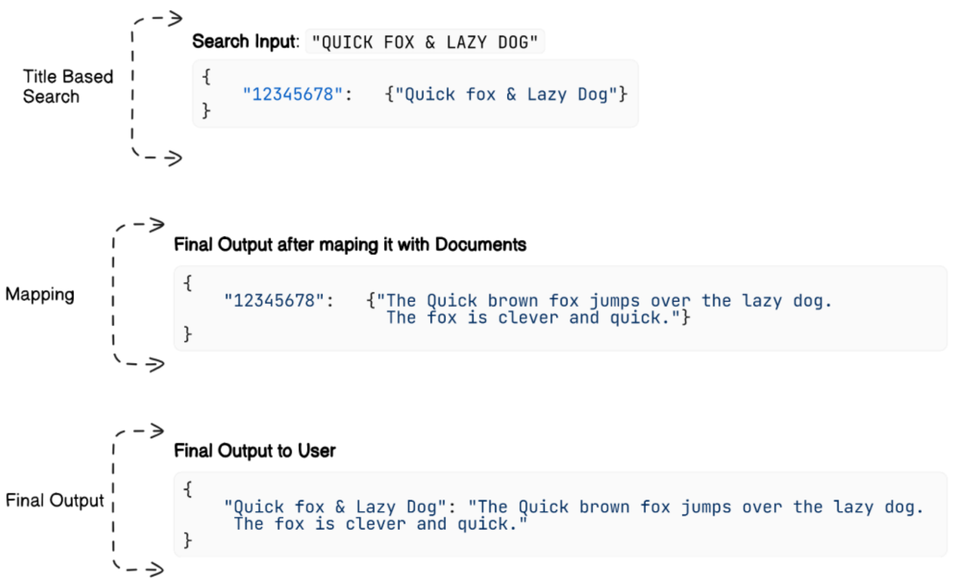
```
Search results for 'fox': {'Nature and Wildlife': 1}
Search results for 'python': {'Technology and Programming': 1}
Search results for 'exercise': {'Health and Fitness': 2}
Search results for 'adventure': {'Travel and Adventure': 1}
```

Figure 7: Word Based Search execution

7. Title Based Search

The title-based search allows users to retrieve the full content of a document by specifying its title. This is useful when the user knows the exact document they are looking for.

7.1. Workflow



```
"QUICK FOX & LAZY DOG"
```

```
{
    "12345678":    {"Quick fox & Lazy Dog"}
}
```

```
{
    "12345678":    {"The Quick brown fox jumps over the lazy dog.
                     The fox is clever and quick."}
}
```

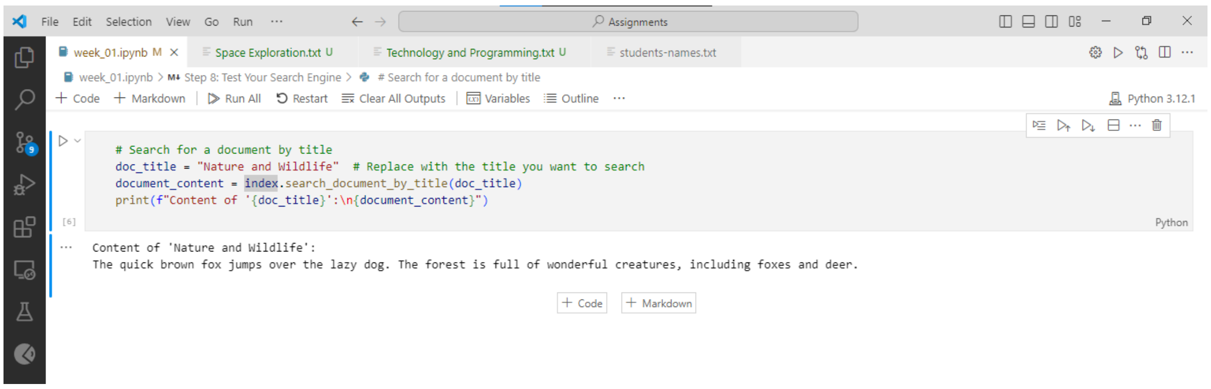
```
{
    "Quick fox & Lazy Dog": "The Quick brown fox jumps over the lazy dog.
     The fox is clever and quick."
}
```

Figure 8: Title Based Search workflow

7.2. Code

```python
def search_document_by_title(self, title):
    """Search and return the content of a document by its title."""
    for doc_id, doc_title in self.document_titles.items():
        if doc_title.lower() == title.lower():
            return self.documents[doc_id]
    return f"Document with title '{title}' not found."
```

7.3. Screenshot


File Edit Selection View Go Run ···
Assignments
week_01.ipynb M
Space Exploration.txt U
Technology and Programming.txt U
students-names.txt
week_01.ipynb > M↓ Step 8: Test Your Search Engine > # Search for a document by title
+ Code + Markdown | Run All Restart Clear All Outputs | Variables Outline ···
Python 3.12.1
[6]
Python
+ Code + Markdown

```python
# Search for a document by title
doc_title = "Nature and Wildlife"  # Replace with the title you want to search
document_content = index.search_document_by_title(doc_title)
print(f"Content of '{doc_title}':\n{document_content}")
```

```
Content of 'Nature and Wildlife':
The quick brown fox jumps over the lazy dog. The forest is full of wonderful creatures, including foxes and deer.
```

Figure 9: Title Based Search execution

8. Data Flow Diagram

The Data Flow Diagram (DFD) provides a high-level representation of the data movement within the system. It captures the flow of information during the indexing and searching processes. Here is the dataflow diagram for the Index.

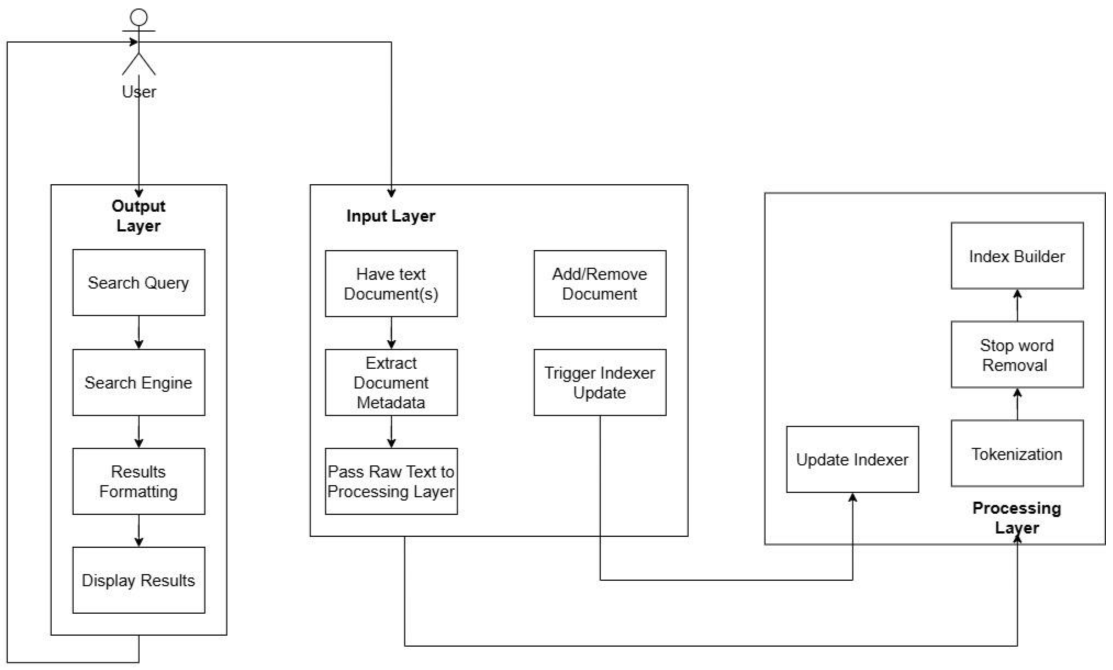


Figure 10: Dataflow Diagram

9. Conclusion

In this project, I have implemented an index-based search engine capable of performing efficient word-based and title-based searches across a collection of documents. By using the inverted index approach, I have optimized the search process, ensuring quick retrieval of relevant documents and their frequencies.

The system is designed to handle dynamic updates, allowing new documents to be added at runtime without compromising performance. Key features like stop-word removal, tokenization, and case normalization were included to improve the accuracy and relevance of search results.

Overall, this project demonstrates the importance of indexing in information retrieval systems and highlights how an indexer can provide a fast and scalable solution for handling large document collections. The implementation not only meets the project objectives but also lays a strong foundation for extending the system with advanced features like phrase searches and ranking mechanisms in the future.