Search Engine, Stage 1

Elisa Breeze
Nicolás Trujillo
Luna Hernández
Carlos Suárez
Ivan Anikin

October 2024

GitHub link

Abstract

In this document we will explain the first stage of creating a search engine with an inverted index. We decided to compare two different data structures: a JSON-based index, and a dictionary-based index. The idea is to evaluate how each structure works to see which one is more efficient when finding the words that want to be looked for.We conducted the performance analysis using Benchmark tools to measure execution time and memory usage of the two query engines. The results indicate that both structures are viable, each with its advantages and disadvantages, but in the end we concluded that the best data structure for our project is the JSON-based one, meaning we will continue using this one for the rest of the project.

1 Introduction

The idea of this work is to create a search engine with an inverted index, that stores words and their positions in documents, allowing efficient searches. The index we created, does not only store which book contains a specific words, but also the location where it appears, allowing a more precise outcome.

By using an inverted index, we can implement fast and effective word searching when a term in queried. In this project we implemented two types of indexers, tested them and chose the best one to continue working with in the next Stages of the Project. The two implementations are: dictionaries and JSON files. We tested them using Benchmark tools, to measure the execution times of the search engines of each version, to see which implementation is the most efficient.

In Stage 1 of the project, a part from the Indexer we also created a Crawler, to download the books periodically from gutenberg.org and save them in a Datamart.

And lastly, so that we could test our search engine, we created a minimal query engine, one for each data structure, that takes the word or words to be searched for, looks for it and gives back information about the book the word is in, the books name, author, URL, the occurrences and the paragraph it's in.

2 Methodology

We started by researching how to do each part of the project, and how we were going to implement them, where to put the focus and how to divide the work. In the end, we worked together on all the code, by creating different versions and then deciding which implementation worked the best. With this, we managed to write the code to all the parts of the project successfully, and here you can see the result of each part of the project, with a short explanation of the methodology used for the module.

2.1 Crawler

We used a simple approach to develop the crawler: it is designed to periodically download book from the Gutenberg web page. We used the "request" library to send HTTP requests to retrieve the content of the books, and the library "BeautifulSoup" to extract the metadata from the webpages, like for example the book title.

Code Structure:

get_title(url): this function sends a GET request to the Gutenberg page of a specific book and uses BeautifulSoup to pase the HTML and extract the title of the book by locating the first ¡h1¿ tag in the HTML structure. If it was successfull, it returns the title, and if not, it returns a message showing there was an error.

download_book(book_id, download_route): This function forms the URL of the book's text tile, by using the book's ID. Then it checks if the download directory exists, and if not, it creates it. Once that has been done, the content of the book and also the title with the get_title() function are retrieved, and saved under a name we composed as the title and the book ID.

The crawler goes through a range of book IDs, specified in the code, and downloads and saves each book in the designated directory.

The crawler is important to collect the data for the rest of the project and by periodically downloading new books, it is ensured that the Book Datamart remains up to date and continues to grow.

```python
import os
import requests
from bs4 import BeautifulSoup


def get_title(url):
    """
    Fetch the title of a webpage by retrieving the text from the first <h1> tag.

    :param url: The URL of the webpage to fetch the title from.
    :return: The text of the first <h1> tag if found; otherwise, an error message or indication of failure.
    """
    try:
        # GET request to URL
        answer = requests.get(url)

        # See if request was successful
        if answer.status_code == 200:
            # Analyze HTML
            soup = BeautifulSoup(answer.text, 'html.parser')

            # find the first <h1>
            h1 = soup.find('h1')

            # If found, give back the text
            if h1:
                return h1.get_text().strip()
            else:
                return "There was no <h1> found."
        else:
            return f"Error accessing the page. Status code: {answer.status_code}"

    except requests.RequestException as e:
        return f"Error making the request: {e}"

```

```python

def download_book(book_id, download_route):
    """
    Download a book from Project Gutenberg using its book ID and save it to a specified directory.

    :param book_id: The ID of the book to be downloaded from Project Gutenberg.
    :param download_route: The directory where the downloaded book will be saved.
    :return: None
    """
    url = f'https://www.gutenberg.org/files/{book_id}/{book_id}-0.txt'
    if not os.path.exists(download_route):
        os.makedirs(download_route)
    answer = requests.get(url)

    if answer.status_code == 200:
        # Extract content as text
        content = answer.text

        # Call assign_title to find the file name
        file_name = get_title(f"https://www.gutenberg.org/ebooks/{book_id}")
        file_name = os.path.join(download_route, f'{file_name}_{book_id}.txt')

        # Save content in file
        with open(file_name, 'w', encoding='utf-8') as file:
            file.write(content)

        print(f"Book {book_id} downloaded correctly under {file_name}")
    elif answer.status_code == 404:
        print(f"Book {book_id} not found.")
    else:
        print(f"Error when downloading Book {book_id}: {answer.status_code}")
    return answer.status_code
```

2.2 Indexers

As explained before, the two indexers implemented use on one hand Dictionaries, and on the other hand, JSON files. The general idea is to save each word with an ID for the book it belongs to, along with its position. Here you can see the explanation of each indexer:

2.2.1 JSON-based Indexer

This indexer stores each word in its own JSON file, which contains a list of all the occurrences of the word across different books, and the basic information (book name, author and ID).

For each word found, a JSON file is created or updated in the case that it already exists. There are 2 versions, a normal one and one that uses parallelism, which is the default one we are going to use in the rest of the project, as it performs much faster. It uses threading, and allows multiple words to be indexed at the same time, which make the process speed it considerably.

It ensures that for every book, all the words and positions are saved in separate JSON files, which creates a modular and distributed system for word searches.

```
1  def indexer5_parallel(datamart_txt_path,
       datamart_json_path):
2      os.makedirs(datamart_json_path, exist_ok=True)
3      txt_files = [f for f in os.listdir(
           datamart_txt_path) if re.match(r'^.+?␣by␣.+?_\d
           +\.txt$', f)]
4
5      for txt_file in txt_files:
6          match = re.match(r'^(.+?)␣by␣(.+?)_(\d+)\.txt$
               ', txt_file)
7          if not match:
8              continue
9
10         book_name = match.group(1)
11         author = match.group(2)
12         index = match.group(3)
13         dictionary_key = f"{book_name}␣by␣{author}␣-␣{
               index}"
14         txt_file_path = os.path.join(datamart_txt_path
               , txt_file)
15
16         with open(txt_file_path, 'r', encoding='utf-8'
               ) as file:
17             content = file.read()
```

```
18
19          words = re.findall(r'\b[a-zA-
                Z                          ]+\b', content
                .lower())
20          words = [word for word in words if
21                   word not in ['in', 'on', 'at', 'by',
                        'for', 'with', 'about', 'against',
                         'between',
22                               'into', 'through', '
                                    during', 'before', '
                                    after', 'above', '
                                    below',
23                               'to', 'from', 'up', '
                                    down', 'of', 'off', '
                                    over', 'under', '
                                    again',
24                               'further', 'once', 'i',
                                    'me', 'my', 'myself',
                                     'we', 'our', 'ours',
                                     'ourselves',
25                               'you', 'your', 'yours',
26                               'yourself', 'yourselves'
                                    , 'he', 'him', 'his',
                                     'himself', 'she', '
                                    her', 'hers',
27                               'herself',
28                               'it', 'its', 'itself', '
                                    they', 'them', 'their
                                    ', 'theirs', '
                                    themselves', 'what',
29                               'which',
30                               'who', 'whom', 'this', '
                                    that', 'these', '
                                    those', 'am', 'is', '
                                    are', 'was', 'were',
31                               'be',
32                               'been', 'being', 'have',
                                     'has', 'had', '
                                    having', 'do', 'does'
                                    , 'did', 'doing', 'a'
                                    ,
33                               'an',
34                               'the', 'and', 'but', 'if
                                    ', 'or', 'because', '
                                    as', 'until', 'while'
                                    , 'of', 'at',
```

```
                                    'by',
                                    'for', 'with', 'about',
                                        'against', 'between',
                                         'into', 'through', '
                                        during', 'before',
                                    'after',
                                    'above', 'below', 'to',
                                        'from', 'up', 'down',
                                         'in', 'out', 'on', '
                                        off', 'over',
                                    'under',
                                    'again', 'further', '
                                        then', 'once', 'here'
                                        , 'there', 'when', '
                                        where', 'why', 'how',
                                    'all',
                                    'any', 'both', 'each', '
                                        few', 'more', 'most',
                                         'other', 'some', '
                                        such', 'no', 'nor',
                                    'not',
                                    'only', 'own', 'same', '
                                        so', 'than', 'too', '
                                        very', 's', 't', 'can
                                        ', 'will', 'just',
                                    'don',
                                    'should', 'now']]

        with ThreadPoolExecutor() as executor:
            for position, word in enumerate(words,
                start=1):
                executor.submit(process_word, word,
                    position, dictionary_key,
                    datamart_json_path)

    print("Indexation Completed.")
```

2.2.2 Dictionary-based Indexer

This indexer words by storing the words as keys, and for each word, it maps the book's ID along with the positions where it appears.

add_words_to_dict: this function is responsible for inserting the words into the dictionary, and associated each word with the book ID and the positions it has in the text.

The indexer goes through all the files in the Book Datamart, reads the contents of each book, tokenizes it into words and then adds the words to the dictionary.

The dictionary structure is simple: it is divided into smaller partial index files, which are then saves a separate JSON file for more efficient retrieval later on.

```
import json
import os
import re

from indexer.book_reader import read_words,
    save_metadata_to_json
from indexer.path_reader import
    extract_files_from_directory


def add_words_to_dict(words, id_book, dictionary):
    """
    Add words and their indexes to a dictionary.

    :param words: A list of words to be added.
    :param id_book: The ID of the book from which the
        words are extracted.
    :param dictionary: The dictionary where words and
        their indexes will be stored.
    :return: The updated dictionary with words and
        their corresponding indexes.
    """
    for idx, word in enumerate(words):
        if word not in dictionary:
            dictionary[word] = {id_book: [idx]}
        else:
            if id_book not in dictionary[word]:
                dictionary[word][id_book] = [idx]
            else:
                dictionary[word][id_book].append(idx)

    return dictionary


def id_search(filepath):
    """
    Extract the first numeric ID from the given file
```

```
        path.
33
34      :param filepath: The path of the file from which
            to extract the ID.
35      :return: The first numeric ID found in the file
            path, or an empty string if no ID is found.
36      """
37      pattern = r'\d+'
38      coindidencies = re.findall(pattern, filepath)
39
40      if coindidencies:
41          result = coindidencies[0]
42      else:
43          result = ''
44      return result
45
46
47  def save_partial_indexers(indexer, output_directory):
48      """
49      Divide the indexer into smaller parts and save
            them as separate JSON files.
50
51      :param indexer: A dictionary containing the
            indexer data to be divided.
52      :param output_directory: The directory where the
            partial indexer files will be saved.
53      :return: None
54      """
55      if not os.path.exists(output_directory):
56          os.makedirs(output_directory)
57
58      partial_indexers = {}
59
60      for word, data in indexer.items():
61          first_letter = word[0].lower()
62          if first_letter not in partial_indexers:
63              partial_indexers[first_letter] = {}
64          partial_indexers[first_letter][word] = data
65
66      for letter, partial_indexer in partial_indexers.
            items():
67          output_file = os.path.join(output_directory, f
                'indexer_{letter}.json')
68          with open(output_file, 'w', encoding='utf-8')
                as file:
69              json.dump(partial_indexer, file,
```

```
                ensure_ascii=False, indent=4)


def indexer_dict(books_datamart, words_datamart,
    output_directory_metadata, stopwords_filepath):
    """
    Create an indexer from book files, filtering out
        stopwords, and save metadata and partial
        indexers.

    :param books_datamart: The path to the directory
        containing the book files.
    :param words_datamart: The output directory for
        saving partial indexers.
    :param stopwords_filepath: The directory
        containing the stopwords TXT file.
    :param output_directory_metadata: The directory
        containing the metadata JSON file.
    :return: None
    """
    indexer = {}
    directory_path = books_datamart
    output_directory = words_datamart
    filepaths = extract_files_from_directory(
        directory_path)

    for filepath in filepaths:
        try:
            words = read_words(filepath,
                stopwords_filepath)
            if words:
                indexer = add_words_to_dict(words,
                    id_search(str(filepath)), indexer)
            else:
                print(f"Warning: No words found in {
                    filepath}")

            save_metadata_to_json(str(filepath),
                output_directory_metadata)

        except Exception as e:
            print(f"Error processing {filepath}: {e}")

    save_partial_indexers(indexer, output_directory)
```

2.3 Minimal Query Engine

2.3.1 Minimal Query Engine for JSON

This query engine is responsible for searching books that contain the keywords provided by the user.

find_book(book_id, book_folder): This method looks for a specific book file. "book_id" identifies the book and "book_folder" is the folder where the files are located.

query_engine(input, book_folder, index_folder, max_occurrences): This function searches for books based on the keywords entered, and loads indices from JSON files, searching for matches in the book contents.

What it does exactly is convert the input to lowercase and split it into single words if more than one word was entered. Then, it loads JSON files from the index and builds a dictionary of words, checking if all the words are present in the dictionary. It then identifies books that contain all the keywords, extracting relevant paragraphs from each book, returning a list of dictionaries, each containing all the information about the book and the paragraphs, so that we can print out the information looked for to the user.

```
import glob
import json
import os
import re


# uncomment if using memory usage test
# from memory_profiler import profile


def find_book(book_id, book_folder):
    """
    Searches for a book file in the specified folder by its ID.

    This function looks for a file that ends with '_{book_id}.txt' in the provided
    book folder and returns the full path if found.

    :param book_id: The ID of the book to search for.
    :param book_folder: The folder where the book files are stored.
    :return: The full path to the book file if found, otherwise None.
```

```
    """
    for filename in os.listdir(book_folder):
        if filename.endswith(f"_{book_id}.txt"):  # find file that ends with _{book_id}.txt
            return os.path.join(book_folder, filename)
    return None


# uncomment if using memory usage test
# @profile
def query_engine(input, book_folder, index_folder, max_occurrences=3):
    input = input.lower()
    words = input.split()
    results = []
    loaded_words = {}

    # save the JSON objects
    for filepath in glob.glob(f"{index_folder}/*.json"):
        with open(filepath, "r") as file:
            data = json.load(file)

            if "id_name" in data and "dictionary" in data:
                word_key = data["id_name"]
                dictionary_info = data["dictionary"]
                loaded_words[word_key] = {"dictionary": dictionary_info}
            else:
                print(f"Invalid structure in file: {filepath}")

    # Check if all the words are there
    words_looked_for = all(word in loaded_words for word in words)

    if words_looked_for:
        books_in_common = None
        for word in words:
            word_info = loaded_words[word]["dictionary"]
            if books_in_common is None:
                books_in_common = set(word_info.keys())
            else:
```

```
58                  books_in_common &= set(word_info.keys
                        ())
59
60          if books_in_common:
61              for book_key in books_in_common:
62
63                  book_info = book_key.split("␣by␣")
64                  book_name = book_info[0].strip()
65                  author_and_id = book_info[1].split("␣-
                        ␣")
66                  author_name = author_and_id[0].strip()
67                  book_id = author_and_id[1].strip()
68
69                  book_filename = find_book(book_id,
                        book_folder)
70
71                  if book_filename:
72                      try:
73                          with open(book_filename, "r",
                                encoding="utf-8") as file:
                                 # we have to specify the
                                encoding
74                                text = file.read()
75
76                          paragraphs = text.split('\n\n'
                                )
77                          relevant_paragraphs = []
78                          occurrences = 0
79
80                          word_pattern = re.compile(rf"\
                                b{input}\b", re.IGNORECASE)
81
82                          for paragraph in paragraphs:
83                              if word_pattern.search(
                                    paragraph):
84                                  occurrences += len(
                                        word_pattern.
                                        findall(paragraph))
85
86                                  highlighted_paragraph
                                        = word_pattern.sub(
                                        f"\033[94m{input
                                        }\033[0m",
                                        paragraph)
87                                  relevant_paragraphs.
                                        append(
```

```
                                    highlighted_paragraph
                                    .strip())
88
89                          if relevant_paragraphs:
90                              results.append({
91                                  "book_name": book_name
                                        ,
92                                  "author_name":
                                        author_name,
93                                  "URL": f'https://www.
                                        gutenberg.org/files
                                        /{book_id}/{book_id
                                        }-0.txt',
94                                  "paragraphs":
                                        relevant_paragraphs
                                        [:max_occurrences],
95                                  "total_occurrences":
                                        occurrences
96                              })
97
98                      except FileNotFoundError:
99                          print(f"Error: The Book {
                                book_filename} was not
                                found.")
100
101     return results
102
103 # uncomment if doing memory usage tests
104 # book_datamart_folder = "../Books_Datamart"
105 # indexer_folder = "../Words_Datamart"
106 # query_engine("wife", book_datamart_folder,
      indexer_folder)
```

2.3.2 Minimal Query Engine for Dictionary

The dictionary based query engine looks for books, using a different index structure, based on dictionaries.

load_json_index(word index_folder): This function loads the JSON index file for a word based on its first letter and returns a dictionary with the index data, or an empty dictionary if the file is not found.

load_metadata(book_id, metadata_folder): This function loads the metadata of the book, based on its ID, returning a dictionary with the book's metadata or None if not found.

extract_paragraphs(book_filename, occurrences_dict, search_words: This function extracts relevant paragraphs from a book based on occurrences dictionary and the searched words, and returns a list of relevant paragraphs containing the words searched for.

query_engine(input, index_folder, metadata_folder, book_folder, max_occurrences): This function looks for the books that contain all the words in the input, and returns the information and paragraphs. It converts the input to lowercase and splits it into words, then loads the word indices for each one in the query, finds common books that contain all the words. Finally it loads the metadata and content of each book, extracting the relevant paragraphs using its function, and returns a list of dictionaries, each containing information about the book and relevant paragraphs, associated to the searched word or words.

```
import json
import os
import re


# uncomment if using memory usage test
# from memory_profiler import profile


def load_json_index(word, index_folder):
    """
    Loads the JSON file for the word's index based on its first letter.

    :param word: The word whose index is to be loaded.
    :param index_folder: The directory containing the index JSON files.
    :return: A dictionary with the index data, or an empty dictionary if the file is not found.
    """
    first_letter = word[0].lower()
    json_path = os.path.join(index_folder, f'indexer_{first_letter}.json')

    if os.path.exists(json_path):
        with open(json_path, 'r', encoding='utf-8') as file:
            return json.load(file)
    else:
        print(f"Index file for letter '{first_letter}' not found.")
```

```
        return {}


def load_metadata(book_id, metadata_folder):
    """
    Loads the metadata of a book based on its book ID.

    :param book_id: The ID of the book whose metadata
       is to be loaded.
    :param metadata_folder: The directory containing
       the metadata JSON files.
    :return: A dictionary with the book's metadata, or
        None if not found.
    """
    hundred_range = (int(book_id) // 100) * 100
    json_filename = f"books_metadata_{hundred_range}-{
       hundred_range + 99}.json"
    json_filepath = os.path.join(metadata_folder,
       json_filename)

    if os.path.exists(json_filepath):
        with open(json_filepath, 'r', encoding='utf-8'
           ) as file:
            books_data = json.load(file)
            for book in books_data:
                if book['id_book'] == book_id:
                    return book
    return None


def extract_paragraphs(book_filename, search_words):
    """
    Extract paragraphs from the book based on
       occurrences dictionary and search words.

    :param book_filename: The path to the book file
       from which to extract paragraphs.
    :param search_words: A list of words to search for
        in the book paragraphs.
    :return: A list of relevant paragraphs containing
       the search words.
    """
    try:
        with open(book_filename, "r", encoding="utf-8"
           ) as file:
            text = file.read()
```

```
61
62          paragraphs = text.split('\n\n')
63          relevant_paragraphs = []
64          occurrences = 0
65          word_patterns = {word: re.compile(rf"\b{word}\
                b", re.IGNORECASE) for word in search_words
                }
66
67          for paragraph in paragraphs:
68              for word, pattern in word_patterns.items()
                    :
69                  if pattern.search(paragraph):
70                      occurrences += len(pattern.findall
                            (paragraph))
71
72                      highlighted_paragraph = pattern.
                            sub(f"\033[94m{word}\033[0m",
                            paragraph)
73                      relevant_paragraphs.append(
                            highlighted_paragraph.strip())
74                      break
75          return relevant_paragraphs, occurrences
76
77      except FileNotFoundError:
78          print(f"Error: Book file not found: {
                book_filename}")
79          return [], 0
80
81
82  # uncomment if using memory usage test
83  # @profile
84  def query_engine(input_query, index_folder,
        metadata_folder, book_folder, max_occurrences=3):
85      """
86      Searches for books containing the words in the
            input query and returns relevant paragraphs.
87
88      :param input_query: The search query (a string of
            words).
89      :param index_folder: Directory where the word
            index files are stored.
90      :param metadata_folder: Directory where the book
            metadata JSON files are stored.
91      :param book_folder: Directory where the book files
             are stored.
92      :param max_occurrences: Maximum number of
```

```
        paragraphs to return for each book.
93      :return: List of dictionaries with book
            information and paragraphs containing the
            search words.
94      """
95      input_query = input_query.lower()
96      words = input_query.split()
97      results = []
98
99      # Dictionary to store word occurrences across
            books
100     word_occurrences = {}
101
102     # Step 1: Load word indices for all search words
103     for word in words:
104         index_data = load_json_index(word,
                index_folder)
105         if word in index_data:
106             word_occurrences[word] = index_data[word]
107         else:
108             print(f"Word '{word}' not found in any
                    index.")
109             return []  # Exit early if any word is
                    missing
110
111     # Step 2: Find common books that contain all the
            search words
112     common_books = None
113     for word, occurrences in word_occurrences.items():
114         if common_books is None:
115             common_books = set(occurrences.keys())
116         else:
117             common_books &= set(occurrences.keys())
118     # Step 3: Process each book that contains all the
            words
119     if common_books:
120         for book_id in common_books:
121             # Step 4: Load book metadata
122             metadata = load_metadata(book_id,
                    metadata_folder)
123             if not metadata:
124                 print(f"Metadata for book ID '{book_id
                        }' not found.")
125                 continue
126
127             book_name = metadata["book_name"]
```

```
            author_name = metadata["author"]
            url = metadata["URL"]

            # Step 5: Load the book content
            book_path = os.path.join(book_folder, f"{
                book_name} by {author_name}_{book_id}.
                txt")

            # Step 6: Extract relevant paragraphs
            paragraphs, occurrences =
                extract_paragraphs(book_path, words)
            if paragraphs:
                results.append({
                    "book_name": book_name,
                    "author_name": author_name,
                    "URL": url,
                    "paragraphs": paragraphs[:
                        max_occurrences],
                    "total_occurrences": occurrences
                })

    return results

# uncomment if doing memory usage tests
# indexer_folder = "../Words_Datamart_Dict"
# metadata_datamart_folder = "../Books_Metadata_Dict"
# book_datamart_folder = "../Books_Datamart"
# query_engine("wife", indexer_folder,
    metadata_datamart_folder, book_datamart_folder)
```

2.3.3 Rest of the code

To make the project work and ensure that all functions work well together, we included several controllers: a crawler controller, an indexer controller for the JSON structure, an indexer controller for the dictionary structure, and a query engine controller. Each of these controllers is executed from the main function within their respective files, allowing the entire search engine to operate correctly.

3 Experiments and Results

To analyze the performance of the two indexers, we executed benchmarks to analyze the execution times, so that we could see which option has a better performance.

To do so, we included a benchmark function in a separate file, to do testing on the two query engines, and measure the execution time and memory usage of each one to find out which one is the best.

In the Benchmark file, there is code for the benchmark test. To execute the tests, use the command:

```
pytest benchmark.py --benchmark-group-by=func
```

To save the results, you can run:

```
pytest benchmark.py --benchmark-save indexer_benchmarks
```

Additionally, to test memory usage, in the actual queryEngine files, there is a `@profile` command and a function call at the bottom so that `memory_profiler` can do the test. To do this, just un-comment those two lines of code.

Here you can see the code for the benchmark:

```
from queryEngine import query_engine,
    query_engine_dict


def test_query_engine(benchmark):
    benchmark.extra_info['unit'] = 'ms'
    book_datamart_folder = "../Books_Datamart"
    indexer_folder = "../Words_Datamart"
    benchmark.pedantic(
        target=query_engine.query_engine,
        args=("wife", book_datamart_folder,
            indexer_folder,),
        iterations=100,
        rounds=10,
        warmup_rounds=5
    )


def test_query_engine_dict(benchmark):
    benchmark.extra_info['unit'] = 'ms'

    indexer_folder = "../Words_Datamart_Dict"
    metadata_datamart_folder = "../Books_Metadata_Dict
        "
    book_datamart_folder = "../Books_Datamart"

    benchmark.pedantic(
        target=query_engine_dict.query_engine,
```

```
        args=("wife", indexer_folder,
            metadata_datamart_folder,
            book_datamart_folder,),
        iterations=100,
        rounds=10,
        warmup_rounds=5
    )

# to execute => pytest benchmark.py --benchmark-group-by=func
# to save results => pytest benchmark.py --benchmark-save indexer_benchmarks
# to execute memory usage => in query engine module, I added @profile, then execute in there, first uncomment the profile and function call at the enc:
# python -m memory_profiler query_engine.py
# all of these test, from inside the package
```

And here you can see the results we obtained for each query engine:

Index Type	Execution Time (ms)	Memory Usage (MiB)
Dictionary-based	2.81013e-05	44.1
JSON-based	1.79022e-05	43.8

Table 1: Benchmark test results for the indexers

4 Discussion

It's paradoxical that, although the dictionary type is the fastest during indexing, when it comes to the query engine, it's the opposite.

The performance of the benchmark and memory usage tests show that they are quite similar, but that the JSON query engine is slightly faster and has a slightly better memory usage, with 1.79e-05ms compared to 2.81e-05ms; and 43.8MB compared to 44.1MB.

It's important to note, however, that these figures reflect the query engine performance, that is, the part the user interacts with. When it comes to indexing, the JSON-based indexer actually takes longer to complete the indexing process.

5 Conclusion

After finishing the code and comparing the two indexing approaches for our search engine, we have concluded, that even though they are very similar, for our approach to the whole project, it is better to use JSON-based indexers. As

we observed, the query engine that uses the files generated by the JSON-based indexer provides a slight improvement in speed. All this, together with the flexibility afforded by the modular structure, we have decided to continue with the JSON-based version for the next stages of the project.

On the other hand, the dictionary-based indexer, while also a viable option, did not perform as efficiently, but its simpler structure and fewer file accesses still offer benefits if we see that in the future stages of the project handling a large number of JSON files becomes a problem.