IN6226 Information Retrieval Analysis / Assignment 1

Name	Matric	Email
HUANG YIJING	G2405655L	yijing005@e.ntu.edu.sg
LIANG HOU	G2401631D	HLIANG008@e.ntu.edu.sg
XIE QINGLING	G2405624J	QINGLING001@e.ntu.edu.sg
ZENG HUA	G2405796B	HUA002@e.ntu.edu.sg

1 Introduction

Efficient indexing of large-scale text collections is crucial for information retrieval. Traditional in-memory methods struggle with massive datasets due to memory constraints. To address this, we implement the **Blocked Sort-Based Indexing (BSBI)** algorithm, which processes documents in small blocks, sorts them, and merges the results into an inverted index. Our evaluation demonstrates that BSBI efficiently constructs the index while maintaining reasonable time and memory usage.

2 Data Collection & Preprocessing

2.1 Tokenization

Before building the inverted index with BSBI/SPIMI algorithms, the system must first process the raw text documents to extract standardized terms.Therefore, we will first use NLTK (Natural Language Toolkit) for text segmentation and preprocessing.

1. **Retrieving File Paths (`list_files(directory)`)** This function scans a given directory and returns a list of file paths for text processing.

2. **Downloading Required Resources** Downloads the Punkt tokenizer, which is essential for splitting text into tokens.

3. **Tokenization and Preprocessing (`preprocess_text(text)`)** Uses `nltk.word_tokenize(text)` to split text into individual words or symbols.

2.2 Linguistic Processing

The preprocessing pipeline consists of two main stages:

1. **Normalization**: This step standardizes the textual data by converting tokens to a uniform format. It removes unwanted characters such as punctuation and digits and converts all letters to lowercase.

2. **Stemming**: This step reduces each normalized token to its root form. Using the Porter Stemmer algorithm, morphological variants of a word are grouped under a single stem, which is useful for various tasks like information retrieval or text classification

The detailed processing steps are as follows:

Normalization Process

The normalization process is encapsulated in the function *normalize_token(token)*. This function performs three key operations:

- **Lowercasing** All characters in the token are converted to lowercase. This step is essential to eliminate case sensitivity issues, so that ”Release”, ”release”, and ”RELEASE” are all treated identically.
- **Punctuation Removal** A regular expression is used to remove any punctuation. This step ensures that tokens like ”release!” or ”release,” are reduced to a clean form.
- **Digit Removal** Another regular expression is used to strip out any numeric characters. This is based on the assumption that digits might be irrelevant or could introduce noise in the analysis.
- **Stopword Removal**: Eliminating common stopwords (e.g., ”the”, ”is”, ”and”) using NLTK's stopword list.

Stemming Process

After normalization, the token is passed to the `stem_token(token)` function which uses the Porter Stemmer from the NLTK library. The Porter Stemmer applies a series of heuristic rules to remove suffixes and reduce the word to its base or ”stem.”

3 Methodology

The BSBI (Block-Sort-Based Indexing) algorithm is an efficient approach for constructing an inverted index for large-scale document collections while optimizing memory usage. It consists of two main phases: **block-wise sorting** and **multi-way merging**.

3.1 Block-wise Sorting

In the first phase, the document collection is processed in blocks to avoid excessive memory consumption. Each document is tokenized, preprocessed, and converted into a list of $(\text{term}, \text{doc\_id})$ pairs. These pairs are accumulated until a predefined block size limit is reached, at which point they are sorted lexicographically by term and stored as a temporary block on disk. The process is implemented in the `BSBIIndexer` class:

Listing 1: Sorting and storing blocks

```
class BSBIIndexer:
    def __init__(self, block_size=100000, output_dir="bsbi_blocks"):
        self.block_size = block_size
        self.output_dir = output_dir
        os.makedirs(self.output_dir, exist_ok=True)

    def index_documents(self, directory):
        """Construct the BSBI index"""
        doc_files = list_files(directory)
        block_id = 0
        term_doc_pairs = []

        for doc_id, file_path in enumerate(doc_files):
            with open(file_path, 'r', encoding='utf-8') as f:
                text = f.read()
                tokens = preprocess_text(text)
```

```
                term_doc_pairs.extend([(token, doc_id) for token in
                    tokens])
            if len(term_doc_pairs) >= self.block_size:
                self.write_block(term_doc_pairs, block_id)
                term_doc_pairs = []
                block_id += 1
        if term_doc_pairs:
            self.write_block(term_doc_pairs, block_id)
            block_id += 1
```

Once a block reaches the predefined threshold, it is sorted and stored on disk:

Listing 2: Sorting and storing blocks

```
def write_block(self, term_doc_pairs, block_id):
    """Sorts a block and writes it to disk"""
    start_time = time.time()
    term_doc_pairs.sort()
    block_file = os.path.join(self.output_dir, f"block_{block_id}.
        json")
    with open(block_file, 'w', encoding='utf-8') as f:
        json.dump(term_doc_pairs, f)
    end_time = time.time()
    print(f"Block␣{block_id}␣written␣with␣{len(term_doc_pairs)}␣pairs
        .␣Sorting␣took␣{end_time␣-␣start_time:.2f}␣seconds.")
```

This block-wise sorting ensures that each block is independently sorted, enabling an efficient merging process.

3.2 Multi-way Merging

The second phase merges the sorted blocks into a final inverted index. A priority queue (min-heap) is used to efficiently merge multiple sorted streams, ensuring that terms and their associated document IDs are retrieved in sorted order. The merging process follows these steps:

- Loads sorted blocks from disk.
- Uses a min-heap to maintain the smallest term from each block.
- Iteratively extracts the smallest term, aggregates document IDs, and writes them to the final index.

The merging process is implemented as follows:

Listing 3: Sorting and storing blocks

```
import heapq

def merge_blocks(block_files, output_index):
    """Merge sorted blocks into final index"""
    heap = []
    file_handles = [open(f, 'r', encoding='utf-8') for f in
        block_files]
    iterators = [json.load(f) for f in file_handles]
    for i, it in enumerate(iterators):
        if it:
```

```
            heapq.heappush(heap, (it[0][0], i, 0))  # (term,
                file_index, term_index)
    final_index = {}

    while heap:
        term, file_idx, term_idx = heapq.heappop(heap)
        if term not in final_index:
            final_index[term] = []
        final_index[term].append(iterators[file_idx][term_idx][1])  #
            Store doc_id

        if term_idx + 1 < len(iterators[file_idx]):
            heapq.heappush(heap, (iterators[file_idx][term_idx +
                1][0], file_idx, term_idx + 1))
    with open(output_index, 'w', encoding='utf-8') as f:
        json.dump(final_index, f)
    for f in file_handles:
        f.close()
```

By employing this two-phase strategy, our BSBI implementation balances computational efficiency and memory constraints, making it suitable for handling large text corpora.

4 Experiments & Evaluation

To evaluate the performance of our BSBI implementation, we indexed the "HillaryEmails" dataset, consisting of multiple text documents. The experiment was conducted with a block size of 100,000 term-document pairs. The following performance metrics were recorded:

- Block Sorting Time: The sorting of each block took between 0.18 and 0.33 seconds, depending on the block size and term distribution.
- Indexing Time: The overall indexing process, including block-wise sorting and writing, was completed in **46.42 seconds**.
- Memory Usage: During block processing, memory usage remains almost unchanged, which aligns with the design principle of independent block processing + disk storage. During the merging phase, memory usage increases by **+58.75MB**, primarily due to loading multiple blocks simultaneously for merge sorting.
- Merging Time: The final merging phase was completed in **2.60 seconds**, demonstrating the efficiency of our heap-based merge approach.

These results highlight the efficiency of the BSBI algorithm in constructing an inverted index for large document collections while keeping memory usage manageable.

References