Introduction:

This report offers a detailed look at the Indexer and query processor made during HW2 and HW3 , tools built to manage information efficiently. It consists of key parts like an inverted index, page table, and lexicon data structure, all neatly woven together during the code execution. The final result is a compressed binary file housing the inverted index. This document is your guide, breaking down the construction process, inner workings, and features of the Indexer. We'll explore the steps from Homework 2 and Homework 3, covering how the index is made and how queries are handled, making it accessible to both seasoned computer scientists and those with a basic understanding of inverted indices in information retrieval.

I have used Xcode for my project.

How to Run the Program with Xcode on macOS:

To run my program on macOS using Xcode, follow these detailed steps:

1. **Xcode Installation:**
   - Ensure that you have Xcode installed on your macOS. You can download and install it from the Apple App Store or the official Apple website.

2. **Project Setup:**
   - Open Xcode and create a new C++ project. Name your project and choose a location for it.

3. **Add Source Files and maintain the directory structure:**
   - In your Xcode project, add the following files to your project:
     - ‘main.cpp’
     - 'indexer.cpp '
     - 'page_table.cpp '
     - 'text_processing.cpp '
     - 'file_operations.cpp '

Following is the directory structure:

```
inverted_indexer/
├── main.cpp
├── indexer.h / .cpp
├── page_table.h / .cpp
├── text_processing.h / .cpp
├── file_operations.h / .cpp
├── docs.trec
├── subinverted/ (sub-index files)
├── merged/ (merged index files)
├── output/ (generated output files)
```

- For running the query processor you just need to run main.cpp of the query processor. If you are not following the above directory structure you need to manually input the pagetable.txt and lexicon.txt along with compressedindex.txt file addresses in the code.

4. **Build and Run:**
   - Build your project by clicking the "Build" button in Xcode. This will compile the C++ source code and generate the executable.

5. **Run the Program:**
   - After building, you can run the program by clicking the "Run" button in Xcode.

6. **View Output:**
   - The HW2 program will generate 3 files:
      1) A compressed inverted index binary file named final_compressed_index.bin,
      2) A page table named PageTable.txt
      3) A Lexicon structure called lexicon.txt

   You can find these files in the output folder.

   - The HW3 program will not generate any new files, it will rather perform conjunctive and disjunctive query processing using the main.cpp for query processing.

Program Functionality:

My indexer performs a range of functions, including:

- **Indexing:** It creates an inverted index that maps words to the documents in which they appear, along with their frequencies. Index is created by going through the .trec files in batches of 1000 and identifying documents and writing the index to intermediate text files. These files are merged and sorted and then finally compressed

- Memory limiter: As mentioned in instructions I have a memory limiter in place that restricts the memory usage till 1 GB. If while running the batches memory exceeds 1 GB it directly writes the file to the disc

- **Page Table:** It maintains a page table that associates document IDs with their respective URLs. It is a text file generated while creating the intermediate index files. It is stored in the output folder.

- **Sorting and Merging:** The program sorts and merges intermediate index files efficiently in batches of 100 and then again in batches of 2 until one file is left.

- **Compression:** It compresses the index files in blocks of 10 using a var-byte compression reaching compression up to 55%.

My query processor performs a range of functions, including:

1. **Text Processing:**
   - The code includes a 'sanitizeText' function that cleans input text by removing non-alphabetic characters and converting alphabetic characters to lowercase.

2. **Compressed Index Decoding:**
   - The 'decodeWord' function reads and decodes variable-length integers from a binary file, using the positions stored in the lexicon table providing access to a compressed posting list for a given word. It involves the use of total postings stored in Lexicon as well used for jumping blocks of frequency to find the docID.

3. **Document Retrieval:**
   - The code retrieves document IDs containing a specific word using the 'docIDsByWord' function, which relies on the decoded posting list. Only the specific list is put under decompression during this process.

4. **BM25 Scoring:**
   - The 'BM25' function calculates BM25 scores for a word in a specific document, considering factors such as term frequency, document length, and document frequency according to the following formula.

$$BM25(q,d) = \sum_{t \in q} \log\left(\frac{N - f_t + 0.5}{f_t + 0.5}\right) \times \frac{(k_1 + 1) f_{d,t}}{K + f_{d,t}}$$

The values of constants such as k1 and b are taken as 1.2 and 0.75 respectively, also the average length of each document was calculated to be near around 100 hence 100 is taken.

5. **Query Processing:**
   - Two functions, 'conjunctive' and 'disjunctive', handle conjunctive and disjunctive queries, respectively. They find documents based on word presence and return the top 10 results using BM25 scores.

6. **Snippet Generation:**
   - The 'generateSnippet' function creates a snippet of text containing a target word and surrounding words. Snippet contains 10 words before and after the target word.

7. **Document Text Retrieval:**
   - The 'getDocumentText' function retrieves a snippet of text for a specified document ID and target word.

8. **User Interaction and Main Loop:**
   - The 'runQuery' function interacts with the user, processing queries and presenting relevant information. The 'main' function initializes data structures, loads files, and repeatedly calls 'runQuery' until the user chooses to exit.

Internal Working:

- The inverted index program follows a structured set of internal steps to achieve its functionality:

1. **Reading Data:** It reads a TREC data file, extracting text and URL information. The code reads the TREC file, which typically contains a collection of documents in a specific format, to extract and index their content. In the 'file_operations.cpp' file, the 'readTRECFile' function is responsible for this task. It initiates by opening the TREC file, and then it iterates through the file's lines. During this traversal, the code identifies and extracts specific content enclosed within tags like '<TEXT>' and '</TEXT>', which often denote the main body of the document. It also captures the document's URL information.

2. **Indexing:** For each document, it tokenizes and indexes the text data by creating an inverted index. It is primarily accomplished through the 'indexer.cpp' file. As the code reads and processes documents from the TREC file, it begins by sanitizing and tokenizing the text, breaking it down into individual words. For each word, the code converts it to lowercase to ensure uniformity. It then maintains an inverted index, represented as a hash map, associating each word with a list of document pairs. This list contains document IDs and their respective frequencies of occurrence for the given word. If the word is encountered multiple times within the same document, the code increments the frequency accordingly. If the word has not been seen before in the document, a new entry is created in the list. In essence, the code captures the semantic structure of the document collection by mapping words to the documents in which they appear, along with their frequencies.

3. **Page Table:** Simultaneously, it constructs a page table associating document IDs with URLs. The page table is responsible for mapping document IDs to their corresponding URLs, providing a convenient and efficient way to locate the actual documents associated with the indexed data. In the 'page_table.cpp' file, the 'createPageTable' function receives as input a document ID and the URL of the document along with document length and start position of that document in the main trec file. This function opens a file named "PageTable.txt" and appends the document ID and its corresponding URLs, Document length and start pointers, separated by a colon and then spaces, in the file. This operation allows the program to maintain a comprehensive record of all documents in the collection, facilitating rapid access to the full content of indexed documents when needed like in the case of snippet generation.

4. **Merging:** The program performs batched merge sorting on intermediate index files to manage memory efficiently. In this code, the sorting and merging operations are carried out in the 'file_operations.cpp' file. First, during the batched merge sort, the program segregates the

indexed documents into batches, where each batch represents a subset of the entire document collection. These batches are then merged, creating intermediate merged files that contain the sorted and merged data from the individual batches. This step optimizes the process of organizing the data efficiently. Next, in the 'mergeSort' function, two intermediate files are merged at a time, ensuring that the data within them is sorted based on the keys, which are the terms in this case. The resulting merged file contains the indexed information in sorted order. These sorting and merging steps are critical in producing an inverted index that enables fast and efficient retrieval of documents.

5. **Compression:** It compresses the final index data in block size of 10, using a variable-byte encoding scheme for space optimization. The compression is achieved through the 'compress' function in the 'file_operations.cpp' file. After the inverted index has been built and saved, the code reads the index from the file and processes it line by line. For each line representing a word and its associated document frequencies, the code tokenizes and extracts the document IDs and their corresponding frequencies. It creates blocks of size 20 where each block contains 10 frequencies and 10 DocIDs. It then encodes these values in chunks of 10 values at a time using a variable byte encoding scheme, a method commonly used for compressing integer values. The compressed bytes for document IDs and frequencies are then written to an output file. The resulting compressed data occupies significantly less space while preserving the essential information needed for efficient retrieval. Additionally, a lexicon file is created to store information about the terms in the index. The information includes document frequency, total postings, start and end positions.

- The query processor program follows these structured set of internal steps to achieve its functionality:

1. **Decompressing:** In this code, the process of decoding plays a crucial role in retrieving information from a compressed index file efficiently. The 'decodeWord' function utilizes variable-byte encoding to decode the postings list for a given word stored in the lexicon. It seeks the corresponding positions in the compressed index file, reads the variable-byte encoded integers using the positions stored in lexicon while jumping frequency blocks of size 10, and reconstructs the original document IDs. Similarly, the 'docIDsByWord' function uses 'decodeWord' to obtain document IDs associated with a specific word. The decoding process involves efficiently extracting information about document frequencies, start and end positions, and total postings from the compressed index. This decoding mechanism enables the subsequent retrieval of document-specific information, such as term frequencies and BM25 scores, contributing to the effective processing of conjunctive and disjunctive queries.

2. **Scoring**: Scoring plays a pivotal role in evaluating the relevance of documents to a given query in this codebase. The BM25 scoring model is employed to assess the importance of specific words within documents. The 'BM25' function calculates the BM25 score for a given word in a particular document by considering factors such as term frequency, document frequency, document length, and average document length. It combines these elements to generate a relevance score indicative of a term's significance in the context of a document. The

'BM25forWords' function extends this scoring mechanism to multiple words within a single document, producing a combined relevance score. Furthermore, the 'BM25Combine' function aggregates scores across multiple documents, providing a comprehensive assessment of document relevance for a set of query words. The scoring process is essential for ranking and retrieving the most relevant documents in response to user queries, facilitating effective information retrieval from the indexed data.

3. **Conjunctive Querying:** Conjunctive querying is a fundamental aspect of the information retrieval process within this codebase. The 'conjunctive' function orchestrates the retrieval of documents that satisfy the conjunctive conditions specified in a user's query. Given a set of query words, this function utilizes the 'docIDsByWord' function to obtain the document IDs associated with each individual word. It then intersects these document sets by iterating through different DocIDs by jumping chunks of 10 to identify documents that contain all the specified query words. The subsequent application of the BM25 scoring model to the intersected document set ensures that the most relevant documents are prioritized. The function eventually returns the top 10 documents based on their BM25 scores, providing users with a concise and relevant set of documents that collectively satisfy the conjunctive query conditions. Conjunctive querying proves crucial in narrowing down search results to documents that encompass all specified terms, enhancing the precision and relevance of information retrieval.

4. **Disjunctive Querying:** Disjunctive querying is a key feature embedded in the information retrieval process in this codebase. The 'disjunctive' function is responsible for retrieving documents that meet the disjunctive conditions specified in a user's query. Given a set of query words, this function utilizes the 'docIDsByWord' function to collect the document IDs associated with each individual word. The unique set of document IDs is then determined, representing documents that contain at least one of the specified query words. Subsequently, the BM25 scoring model is applied to assess the relevance of each document, resulting in a set of scores associated with the disjunctive query. The function ultimately returns the top 10 documents based on their BM25 scores, offering users a diverse selection of documents that individually fulfill the disjunctive query conditions. Disjunctive querying broadens the scope of information retrieval, accommodating scenarios where documents need only contain one or more of the specified terms, thus providing a more comprehensive and inclusive search experience.

5. **Snippet generation:** Snippet generation and text processing are integral components of the information presentation layer in this codebase. The ‘generateSnippet’ function is designed to create concise and contextually relevant snippets for a given word within a document. The process begins by sanitizing the document's content using the ‘sanitizeText’ function, ensuring that only alphabetic characters and spaces are retained. The content is then split into individual words, and the target word's index is identified. The function constructs a snippet by extracting a window of 10 words around the target word, enhancing the user's understanding of the context in which the word appears.
Moreover, the ‘sanitizeText’ function itself serves a critical role in text processing. It iterates through each character in a given line, retaining only alphabetic characters and spaces while converting alphabetic characters to lowercase. This process ensures uniformity in text

representation and aids in subsequent operations such as word extraction and matching. The combined efforts of these functions contribute to the effective generation of informative snippets, offering users a glimpse into the relevant context of a queried word within a document.

6. **Output:** The output generated by this code corresponds to the results of user queries, offering valuable information retrieval insights. Upon entering a query, the program processes the conjunctive or disjunctive conditions specified by the user, retrieves relevant documents based on these conditions, and presents the top 10 matching documents. For each document, the output includes essential details such as the document ID, BM25 relevance score, associated URL, and informative snippets highlighting the context in which the queried words appear. This output serves as a user-friendly presentation layer, facilitating a quick and comprehensible overview of the most relevant documents in response to the user's search criteria. Additionally, the program prompts users to decide whether they want to continue querying, providing an interactive and iterative exploration of the indexed data.

Performance:

The inverted index program is designed to efficiently process data and create the index files in a reasonable amount of time. In this case it took *4 hours* to completely create all files along with the compressed binary file. The performance may vary depending on the dataset size and system specifications, but it's optimized for use on macOS.

For the querying part, each query takes less than a second for conjunction and disjunction. More time is taken in case of very common words with the highest number of postings. Disjunctive querying is slower than conjunctive querying.

Index File Size:

Inverted index without compression: 11 GB
Inverted index after compression: 5.01 GB
Page Table Size: 286 MB
Lexicon Size: 400 MB

Design Decisions:

The code for inverted index creation exhibits several design decisions aimed at achieving a well-structured and functional inverted index, Lexicon and Page table. One key design choice is modularity, as the code is organized into separate header and source files, each responsible for distinct tasks. This modularity enhances code maintainability and readability. Another design decision is error handling; the code provides error messages and checks for file operations, ensuring robustness and fault tolerance. Moreover, the code employs data structures effectively, utilizing hash maps for the inverted index, facilitating efficient word-document mapping. The use of variable byte encoding for compression is another notable design choice, optimizing storage space and retrieval performance. The code also reflects a clear separation of concerns, with

functions specializing in distinct operations like indexing, page table creation, and merging. Finally, the code employs hard-coded file paths, which, while pragmatic for smaller-scale projects, might not be the most scalable approach. In summary, the design of this code is characterized by modularity, robust error handling, effective data structures, and a clear separation of concerns, with some scope for improvements in file path management for larger-scale applications.

The design of the querying code program reflects several thoughtful decisions aimed at achieving efficiency, modularity, and user-centric information retrieval. The use of a compressed index file, implemented through variable-byte encoding, demonstrates a commitment to optimizing storage and retrieval efficiency. The code's modularity is evident in the organization of functions, each serving a specific purpose within the broader context of information retrieval, whether it be decoding, scoring, or querying. The adoption of the BM25 scoring model enhances the relevance assessment of documents, contributing to the precision of search results. Furthermore, the incorporation of conjunctive and disjunctive querying mechanisms allows users flexibility in specifying search conditions. The snippet generation and text processing components cater to a user-friendly presentation layer, offering concise and contextually relevant information. Overall, the design decisions in this code strike a balance between computational efficiency, modular code organization, and user-centric features, collectively contributing to a robust information retrieval system.

Limitations:

- The inverted index program may not be optimized for extremely large datasets and may require additional memory resources.
- The query processing program might take a noticeable amount of time while running disjunctive queries on words with very high postings.
- Error handling in the code is minimal, and it may not gracefully handle all possible input scenarios.

Major Functions and Modules:

Here are the major functions and modules of the inverted index code:

- **main.cpp:** The main program entry point.
- **indexer.h / indexer.cpp:** Indexing functions and data structures.
- **page_table.h / page_table.cpp:** Page table creation and management.
- **text_processing.h / text_processing.cpp:** Text processing and cleaning.
- **file_operations.h / file_operations.cpp:** File I/O, sorting, merging, and compression functions.

1. **Main Function (main.cpp):** The 'main' function serves as the entry point for the program. It orchestrates the overall workflow, including reading the TREC file, batching, merging, sorting, and compression. It also coordinates the various modules and functions.

2. **File Operations** (file_operations.h and file_operations.cpp): This module contains functions related to file operations. The key functions include 'readTRECFile' for parsing TREC documents, 'mergeFiles' for merging intermediate index files, 'mergeSort' for sorting and merging pairs of files, and 'batchedMergeSort' for batched sorting and merging. Additionally, the module includes the 'compress' function for compressing the inverted index.

3. **Indexer** (indexer.h and indexer.cpp): The Indexer module is responsible for building the inverted index. The 'indexer' function takes a list of words and creates an index that maps each word to the documents where it appears along with the term frequencies. The 'saveIndex' function writes the index data to an output file.

4. **Page Table** (page_table.h and page_table.cpp): The Page Table module handles the creation of a page table, associating document IDs with their corresponding URLs. The 'createPageTable' function appends these mappings to a file named "PageTable.txt."

5. **Text Processing** (text_processing.h and text_processing.cpp): This module is responsible for text preprocessing. The 'sanitizeText' function removes non-alphanumeric characters and ensures uniformity in text by converting it to lowercase.

6. **Compression:** The compression of the inverted index is done in the 'compress' function. It encodes document IDs and frequencies using variable byte encoding, producing a more space-efficient representation.

These modules work in tandem to read, process, index, merge, sort, and compress documents, ultimately creating an efficient inverted index for information retrieval purposes. The modularity and division of responsibilities in the code enhance readability, maintainability, and the ease of future updates or extensions to the system.

- Here are the major functions and modules of the query processing code:

1. **Decoding Module:**
   - Functions:
      - 'decodeWord': Decodes variable-byte encoded postings lists for a given word, extracting document IDs and associated information from the compressed index file.
      - 'docIDsByWord': Utilizes 'decodeWord' to retrieve document IDs for a specific word, facilitating subsequent processing.

2. **Scoring Module (BM25):**
   - Functions:

- 'BM25': Computes the BM25 relevance score for a given word in a specific document, considering factors such as term frequency, document frequency, and document length.
- 'BM25forWords': Extends BM25 scoring to multiple words within a single document, producing a combined relevance score.
- 'BM25Combine': Aggregates BM25 scores across multiple documents for a set of query words.

3. **Querying Modules:**
- Conjunctive Querying:
  - Functions:
    - 'conjunctive': Orchestrates the retrieval of documents satisfying conjunctive query conditions, utilizing 'docIDsByWord' and BM25 scoring.
- Disjunctive Querying:
  - Functions:
    - 'disjunctive': Retrieves documents meeting disjunctive query conditions, using 'docIDsByWord' and BM25 scoring.

4. **Snippet Generation and Text Processing:**
- Functions:
  - 'generateSnippet': Creates concise and relevant snippets for a given word within a document, enhancing user comprehension.
  - 'sanitizeText': Iterates through characters in a line, retaining alphabetic characters and spaces while converting alphabetic characters to lowercase.

5. **User Interaction and Query Processing:**
- Functions:
  - 'runQuery': Initiates user interaction, processing queries, and presenting results. Integrates querying, scoring, and snippet generation functionalities.

6. **Main Execution:**
- Function:
  - 'main': The entry point of the program, responsible for loading necessary files, initializing data structures, and orchestrating the overall workflow.

Future Improvements:

To enhance the search engine's capabilities, future improvements could focus on:
- Better compression: taking differences of postings instead of the original numbers to reduce the file size of the compressed file.
- Query Optimization: Reducing disjunctive query processing time for querying of multiple words with highest postings in the index.
- Better Snippet generation: Adopting various machine learning techniques to generate a smarter and better snippet after query.