1. Introduction

This document provides a comprehensive explanation of the program. The program is designed to create an index from a large collection of documents efficiently and do query processing. This document will cover the program's functionality, how to run it, its internal workings, execution time, index size, design decisions, limitations, and a conclusion.

2.Query processor overview:-

The query processor plays a pivotal role in information retrieval systems, acting as the gateway between user search queries and the vast repository of data, with the ultimate goal of delivering relevant results. At its core, the query processor takes a user-entered search query as input and strives to produce the most pertinent information in the form of the top 10 URL results.

The input, a search query, acts as the trigger for the entire process. This query is the user's expression of information needs, and the query processor must decipher and translate it into actionable instructions for retrieval. The desired output, the top 10 URL results, represents a curated selection of web addresses ranked based on their relevance to the user's query.

The query processing algorithm is designed for efficiency and speed, ensuring that users receive results promptly. Leveraging a document-at-a-time approach, the algorithm minimizes the amount of data loaded into main memory, loading only the Lexicon and URL Table during startup. This optimized strategy ensures rapid response times, with conjunctive query results being returned in less than 50 milliseconds.

The use of BM25 as the ranking function is a key component of the algorithm. This algorithm calculates relevance scores for documents based on the frequency of query terms within the document and the inverse document frequency. This nuanced approach to ranking enables the system to not only identify relevant documents but also prioritize them based on their contextual significance.

One of the notable features of the query processor is its disk optimization strategy. Rather than loading entire inverted lists into memory, the processor selectively seeks and reads only the relevant lists corresponding to the query terms. This disk-efficient approach minimizes unnecessary data retrieval, contributing to the overall speed and responsiveness of the system.

In conclusion, the query processor serves as the linchpin in the information retrieval process, transforming user queries into actionable insights through an intricate dance of Boolean filters, Query Scores, and sophisticated algorithms like BM25. Its commitment to efficiency, manifested through rapid response times and intelligent disk optimization, ensures that users can access the most relevant information with minimal delay.

3.Inverted Index generation Overview:-

The process of generating an inverted index is a crucial step in information retrieval, laying the foundation for efficient and effective search operations. This complex task involves several key steps and utilizes various data structures and compression techniques to optimize storage and retrieval.

The first step in the inverted index generation process is the parsing of gzip-compressed wet files, which contain web page information. During this parsing, each document is processed individually, extracting alphanumeric terms and assigning unique termIds based on the order of word occurrence. Simultaneously, DocIds are assigned in the order the documents are encountered, and essential metadata, such as the URL and the number of terms in the document, is appended to the URL Table.

Intermediate postings are then generated in binary format, with each record consisting of 8 bytes (4 bytes for TermId and 4 bytes for DocId). This forms the basis for constructing the inverted index. To achieve this, an I/O-efficient Merge Sort is performed on the intermediate postings, ensuring that the final sorted postings are correct and ready for further processing.

In the subsequent stage, the sorted intermediate postings are parsed, and a smart O(n) algorithm is applied to remove WordId from each posting. Term frequencies are added, resulting in the creation of the final inverted index. Each record in this index contains 8 bytes, with 4 bytes representing DocId and the next 4 bytes representing term frequency. To conserve disk space, the inverted index undergoes compression.

The inverted index relies on several key data structures to organize and manage the data efficiently. The lexicon serves as a critical mapping, linking each term to its start index, end index, and document frequency. This mapping facilitates quick and effective retrieval of information during search operations.

The URL table is another essential data structure, containing metadata about each document, such as DocId, URL, and the number of terms in the document. This information aids in presenting relevant details to users when displaying search results.

Compression techniques play a vital role in optimizing storage space. The use of JavaFastPFOR library introduces blockwise compression to the inverted index. This library employs coding techniques such as Binary Packing, NewPFD, OptPFD, Variable Byte, Simple 9, among others. The structure of an inverted list within the compressed index includes arrays like Last[], SizeOfDocIds[], SizeOfTermFreqs[], and individual chunks. These arrays and chunks are essential components of the compressed index, ensuring that the information is stored in a compact yet retrievable format.

Despite the efficiency introduced by compression, the generation process faces challenges, such as maintaining accuracy throughout the sorting and compression stages. Rigorous checks and algorithms are implemented to ensure the correctness of the final inverted index. In conclusion, the inverted index generation process, with its intricate steps and thoughtful application of data structures and compression techniques, forms the backbone of a robust and high-performance information retrieval system.

4.Internal Working and the functions involved in query processing:-

1. Initialization in the `main` function:

- **Loading Lexicon:**
  - Lexicon is loaded into memory and stored in a `Map<String, Long[]>` called `lexicon`.
  - The map contains mappings of the form: Term => (StartIndex, EndIndex, DocumentFrequency).

- **Loading URL Table:**
  - URL table is loaded into memory and stored in a `List<PageTable>` named `pageTableList`.
  - Each `PageTable` object represents a row in the URL Table.

2. User Query Processing in `executeQuery`:

- User enters a search query.
- `querySearch.executeQuery` is called with the entered query.

3. `executeQuery(string query)` Function:

- Initializes an array with `ListPointer` data structure.
- For each query term, `openList(queryTerm)` is called, and its `ListPointer` is stored in the array.

4. `openList(queryTerm)` Function:

- Checks if the query term is in the lexicon.
- If present, creates a `ListPointer` object and calls `readLastAndSizeArrays(startIndex)`; otherwise, returns NULL.

5. `readLastAndSizeArrays(startIndex)` Function:

- Seeks to the given offset on the compressed inverted index.
- Reads integer arrays (`last`, `SizeOfDocIds`, `SizeOfTermFreqs`) and stores them in the respective member variables.

6. `sortListPointerAndRemoveNull(lp, q, docFreq)` Function:

- Sorts the `ListPointer` array based on document frequency.
- Removes any NULL pointers.

7. Conjunction Query Processing Algorithm:

- Starts a while loop from `did=0`.
- Gets the next posting from the shorter list.
- Calls `nextGEQ(lp[0], did)`.

8. `nextGEQ(lp[0], did)` Function:

- Finds the next posting in list `lp` with docID >= `did`.
- Returns value > MAXDID if none exists.
- Skips the block, finds the block, and uncompresses the `ListPointer` by calling `uncompress(lp)`.

9. `uncompress(listPointer lp)` Function:

- Checks if the block is already uncompressed; if yes, returns the temporary uncompressed block.
- Finds the index of the doc, uncompresses the block by performing seek and read on the inverted index, and returns the block or list of documents in that chunk.

10. `getFreq(lp[i])` Function:

- Checks if there are entries with the same docID in other list pointers.
- If not in the intersection, changes `did` to `d`; if in intersection, gets all frequencies and calls `getFreq(lp[i])`.

11. `computeBM25Score(did, q, f, docFreq)` Function:

- Computes BM25 scores from frequencies and other data using the provided formula.
- Increases the search for the next post.

12. Disjunctive Query Processing Algorithm:

- Gets list pointers using `openList(lp[i])`.
- Starting from docid 0, computes BM25 scores on the given list pointers.

13. Print Top 10 Results:

- Prints the top 10 results after both searches, including rank scores.
- Uses `TreeMap<Double, TopResult>` to store BM25 scores as keys and metadata associated with the web page as values.

This detailed breakdown provides a comprehensive overview of the program's flow and the role of each function in processing user queries, handling conjunction and disjunction, and computing BM25 scores.

5.Internal Working and the functions involved in index Generation:-

1. **Parsing Wet Files:**
   - `parseWetFile(String wetFilePath)` processes one Wet file at a time, uncompressing and parsing documents.
   - It uses the ArchiveReader Library to check the mime type and read the uncompressed ArchiveRecord into a byte array.
   - Converts the byte array to a string, splitting it using regular expressions to extract alphanumeric terms.
   - Iterates over the terms, assigning each a unique wordId using `wordToWordId` HashMap.

2. **Intermediate Posting Generation:**
   - `generateIntermediatePosting(int wordId, int docId)` takes wordId and docId as input and writes them in binary to the `IntermediateInvertedIndex` file.
   - Converts integers to bytes using ByteBuffer, changes byte ordering to little-endian, and writes to the file using DataOutputStream.

3. **URL Table Update:**
   - `addEntryToPageTable(String url, int size)` appends a new row to the `URLTableFile` representing a document's URL, DocId, and the number of terms.

4. **I/O Efficient Merge Sort:**
   - `mergeSort(String memoryLimit)` runs an I/O-efficient merge sort program written in C using the ProcessBuilder class.
   - **Sort Phase:**
     - Reads in `memsize` bytes from the intermediate Inverted List, sorts them in chunks, and creates temporary sorted files (`temp0`, `temp1`, ..., `temp39`).
     - Writes the list of generated files to the "list" file.
   - **Merge Phase:**
     - Uses a min-heap to merge the sorted files, selecting the record with the minimum wordId at the root.
     - Writes the selected records to the output file (`result0`) and updates the heap with the next record from the corresponding file.
     - Writes the filename of the merged result file to "list2."
   - **Check Correctness Phase:**

   - `Checkoutput` compares consecutive records, checking if they are sorted by wordId and docId.

5. **Reformatting Postings:**
   - `reformatPostings()` reads the `result0` file generated by the merge sort.
   - Converts bytes to little-endian using ByteBuffer, updates `docIdToTermFrequency` TreeMap in a single iteration.
   - This linear-time algorithm updates the Lexicon using the TreeMap.

6. **Generate Lexicon File:**
   - `generateLexiconFile()` iterates over the keyset of the `lexicon` TreeMap, writing out terms in alphabetical order along with their startIndex, endIndex, and document frequency to `Lexicon.txt` file.

The program combines Java and C components to efficiently parse, sort, and format the inverted index. Data structures like ArrayLists, HashMaps, and TreeMaps are utilized to manage information, and the I/O-efficient merge sort facilitates sorting large datasets with limited memory. The final Lexicon and URL Table files provide metadata, while the Inverted Index offers a structured and compressed representation of the document-term relationships for efficient retrieval.

6.Document at a time query Processing:-

Document-at-a-time query processing is a pivotal algorithm in information retrieval systems, focusing on efficiently finding intersections among inverted lists and computing BM25 scores for relevant documents. This approach offers a step-by-step mechanism for processing queries and scoring documents, aiming to provide accurate and timely results.

The algorithm begins by opening inverted lists for reading using the `openList()` function. These lists represent the postings of terms in documents and are traversed simultaneously from left to right. The primary objective is to efficiently identify document intersections and calculate BM25 scores for the relevant documents.

Algorithm Steps:
1. **Initialization:**
   - Example: Open 3 inverted lists using `openList()` – obtaining pointers `lp0`, `lp1`, and `lp2` to the starts of these lists.

2. **Traversal and Intersection:**
   - Call `d0 = nextGEQ(lp0, 0)` to get the docID of the first posting in `lp0`.
   - Call `d1 = nextGEQ(lp1, d0)` to check for a matching docID in `lp1`.
   - If `d1 > d0`, restart at the first list by calling `d0 = nextGEQ(lp0, d1)`.
   - If `d1 = d0`, call `d2 = nextGEQ(lp2, d0)` to see if `d0` is also in `lp2`.

- If `d2 > d0`, restart at the first list by calling `d0 = nextGEQ(lp0, d2)`.
- If `d2 = d0`, then `d0` is in all three lists.

3. **Scoring and Result Insertion:**
   - Compute the score for `d0` based on the BM25 scoring algorithm.
   - Continue at the first list and call `d0 = nextGEQ(lp0, d2 + 1)`.
   - Whenever a score is computed for a docID, check if it should be inserted into a TreeMap of the current top-10 results.

4. **Result Presentation:**
   - At the end of the processing, return the results in the TreeMap containing the top-10 documents based on their BM25 scores.

Page 2: Efficiency and Optimizations

The document-at-a-time query processing algorithm demonstrates efficiency through its careful traversal and scoring mechanisms. It employs a strategy of checking for matching docIDs in inverted lists, restarting if necessary, and computing scores for relevant documents. This approach minimizes unnecessary computations and focuses on the most promising intersections.

Efficiency Highlights:
1. **Simultaneous Traversal:**
   - The algorithm traverses inverted lists simultaneously, reducing the need for repeated passes over the same data.

2. **BM25 Scoring:**
   - The use of the BM25 scoring algorithm allows for the computation of relevance scores, taking into account term frequencies and document lengths.

3. **Top-10 Results TreeMap:**
   - Results are efficiently managed using a TreeMap for the current top-10 documents. This ensures constant-time insertion and easy retrieval of the most relevant results.

Optimizations:
1. **Early Termination:**
   - The algorithm has the potential to terminate early when certain conditions are met, avoiding unnecessary computations for documents that cannot improve the current top-10.

2. **Traversal Restart:**
   - Restarting traversal when a matching docID is not found in a list helps avoid unnecessary comparisons and ensures the algorithm focuses on promising intersections.

In conclusion, the document-at-a-time query processing algorithm excels in its efficiency and systematic approach. By combining simultaneous traversal, BM25 scoring, and careful result management, it provides a streamlined mechanism for retrieving the most relevant documents in response to user queries.

7.Forward seeks:-

Forward seeks are a critical aspect of query processing, playing a key role in efficiently accessing and retrieving information from compressed inverted indices. This process involves seeking to specific positions in the compressed index to read necessary data for query evaluation. The strategies employed in forward seeks, along with blockwise compression techniques, contribute significantly to the overall efficiency of the query processing system.

Functions involving Forward Seeks:

1. **`readLastAndSizeArrays(startIndex)` in ListPointer Class:**
   - This method seeks to a given offset on the compressed inverted index and reads integer arrays (last, SizeOfDocIds, SizeOfTermFreqs), storing them in the respective member variables.
   - This strategy optimizes forward seeks by efficiently retrieving necessary information directly from the compressed index, reducing the need for unnecessary data decompression.

2. **`uncompress(listpointer lp)`:**
   - Checks if the block is already uncompressed. If so, it returns the temporarily uncompressed block.
   - If the block is not yet uncompressed, it finds the index of the document, performs seek and read operations on the inverted index to uncompress the block.
   - It checks for entries with the same docID in other list pointers. If they are not in the intersection, the docID is changed to 'd'. If they are in the intersection, it retrieves all frequencies by calling `getfreq(lp[i])`.

Benefits of Blockwise Compression:

1. **Efficient Seek and Retrieval:**
   - Blockwise compression allows the system to skip unnecessary decompression of entire blocks when seeking specific information. This significantly speeds up the forward seek process.

2. **Reduced I/O Operations:**
   - By working with compressed blocks, the system minimizes the amount of data that needs to be read from disk. This reduction in I/O operations contributes to faster query processing times.

3. **Optimized Storage Utilization:**
   - Storing and retrieving data in compressed blocks optimizes storage utilization, enabling the system to handle large amounts of data efficiently without sacrificing performance.

4. **Selective Decompression:**
   - The system can selectively decompress only the required blocks, avoiding the overhead of decompressing entire indexes. This is particularly beneficial when seeking information related to specific query terms or document IDs.

In conclusion, forward seeks, particularly when implemented in conjunction with blockwise compression techniques, enhance the efficiency and speed of query processing. These strategies reduce the computational load associated with seeking information in compressed inverted indices, contributing to the overall responsiveness of the system. The balance between seeking specific information and leveraging compression technologies is crucial for optimizing the performance of information retrieval systems.

8.Results and Performance:-

The performance of the document indexing and query processing system is a critical metric for evaluating its efficiency and practical usability. The provided dataset, along with the details of the program execution, offers insights into the processing speed, size of the resulting inverted index files, and the responsiveness of the query processor.

For the given dataset, the program exhibited a commendable performance. It parsed approximately 3.2 million pages in just 83 minutes, demonstrating a processing speed of 371 pages per second. The execution time of the program is expected to be influenced by factors such as the size and complexity of the input dataset. In this case, the program took around 3 hours to create the final index, reflecting the substantial computational effort required for the given volume of data.

The size of the resulting inverted index files is a crucial aspect of system efficiency and storage utilization. Before compression, the inverted index reached a size of 3 GB, reflecting the extensive information captured during the indexing process. However, through effective compression techniques, the final index size was significantly reduced to around 1.5 GB. This reduction showcases the program's ability to optimize storage utilization without compromising the integrity and accessibility of the indexed information.

In this scenario, the query processor demonstrated impressive speed, returning the top 10 results in just 50-100 milliseconds. In conclusion, the results obtained from processing the dataset highlight the program's capability to handle a substantial volume of data efficiently. The parsing speed, index sizes before and after compression, and the responsiveness of the query processor collectively indicate a well-optimized system for information retrieval.

Design decisions and limitations:-

9.Design Decisions:

Query Processor:
1. **Size Arrays in Compressed Inverted List:**
   - Decision: Two size arrays (SizeOfDocIds[], SizeOfTermFreqs[]) were implemented as headers in the compressed inverted list instead of a single size array. This separation was chosen to enhance compression efficiency and optimize list traversal.

2. **JavaFastPFOR Compression Library:**
   - Decision: The JavaFastPFOR integer compression library was chosen for compression due to its efficiency, especially in terms of decompressing integers at a high rate, exceeding 1.2 billion per second (4.5 GB/s).

Index Generation:
1. **Java for Parsing and Initial Processing:**
   - Decision: Java was selected for the initial part of generating intermediate postings from documents, leveraging its strong support for WET parsing libraries.

2. **C for Sorting and Merging Postings:**
   - Decision: C was chosen for the sorting and merging of postings due to its superior I/O efficiency, particularly advantageous for handling large datasets.

3. **Java for Reformatting and Lookup Structure Updates:**
   - Decision: Java was used for the final part of reformatting sorted postings and updating lookup structures. Manipulating bytes and writing out Lexicon and URL Table to disk is more convenient in Java.

4. **Endian Notation Consideration:**
   - Decision: Recognition of the difference in byte order notation (BIG_ENDIAN in Java and LITTLE_ENDIAN in C) was addressed to ensure consistency and prevent issues when reading bytes across languages.

5. **Selective Uncompression:**
   - Decision: The decision to uncompress only the document that needs parsing at the moment was made, prioritizing efficient use of disk space.

6. **WordIdToWord HashMap:**
   - Decision: Implemented a WordIdToWord HashMap to efficiently replace WordId with the actual word in the final Inverted Index.

10.Limitations:

1. **HashMap Memory Growth:**
   - Limitation: HashMaps in Java can experience significant memory growth and may affect performance after a certain point. Adjusting initial capacity and load factor values was explored to optimize performance. Writing them out to disk and emptying them after reaching a threshold could be a potential solution.

2. **Heap Space Allocation:**
   - Limitation: The Query Processor program requires a considerable amount of heap space due to loading Lexicon and URL table into HashMaps. While this may impact memory usage, the query processing performance in terms of time remains excellent.

Designing an efficient information retrieval system involves a delicate balance between language-specific advantages, compression techniques, and memory management. The decisions made in this design aimed to maximize efficiency, but limitations in memory handling highlight the ongoing challenges in optimizing large-scale data processing systems.

11. Conclusion:-

The described program is an efficient and robust search engine system that encompasses both the index generation and query processing phases. The system is designed to handle the given corpus of documents and execute search queries with high performance.
The use of an I/O efficient merge-sort in C ensures optimal sorting and merging of intermediate postings, resulting in a final inverted index.The query processor is designed to efficiently process both conjunctive and disjunctive queries using the BM25 ranking function.It loads only the essential data structures, namely Lexicon and URL Table, into memory to conserve resources.The document-at-a-time query processing algorithm efficiently traverses inverted lists and computes scores, resulting in fast response times.The system demonstrates impressive performance metrics, processing conjunctive queries in less than 50-100 milliseconds.For the given dataset, the program exhibited a commendable performance. It parsed approximately 3.2 million pages in just 83 minutes, demonstrating a processing speed of 371 pages per second.In conclusion, the presented program successfully combines efficient index generation with a responsive query processor.