File Retrieval Engine

Introduction:

This report details the development and evaluation of a File Retrieval Engine as part of an assignment. The Engine is designed to index files from datasets and facilitate search queries for specific terms within those files. This document provides insights into the architecture, specifications, datasets, performance evaluation, and interpretation of results.

File Retrieval Engine Architecture:

The File Retrieval Engine architecture is designed with a layered approach to ensure modularity, separation of concerns, and scalability. Each component has distinct responsibilities and interacts with the others to facilitate the overall functionality of the system.

AppInterface:

- The AppInterface component serves as the entry point for user interactions with the File Retrieval Engine.

- It provides a command-line interface (CLI) through which users can issue indexing and search commands.

- The AppInterface is responsible for parsing user commands, validating input, and forwarding commands to the ProcessingEngine for execution.

- Additionally, it handles the presentation of results to the user, displaying indexing progress, search results, and any error messages.

Processing Engine:

- The Processing Engine component acts as the core processing unit of the File Retrieval Engine.

- It receives commands from the AppInterface and executes the necessary operations, including indexing and search.

- For indexing, the Processing Engine traverses through the dataset files, extracts terms, and updates the index stored in the IndexStore.

- During search operations, the Processing Engine processes the user's query, retrieves relevant files from the index, and sorts them based on occurrence frequency.

- The ProcessingEngine encapsulates the logic for efficient indexing and retrieval, abstracting away the complexities from the user interface layer.

IndexStore:

- The IndexStore component serves as the data storage and management module for the index generated by the File Retrieval Engine.

- It stores the index data structure, which typically consists of mappings from terms to a list of files containing those terms and their respective frequencies.

- The IndexStore provides methods for updating the index with new terms and frequencies during indexing operations and supports lookup operations for search queries.

- It utilizes appropriate data structures like HashMaps to efficiently store and retrieve index information, ensuring fast access and manipulation of index entries.

Interactions:

- The AppInterface communicates with the ProcessingEngine by invoking its methods and passing parameters corresponding to user commands.

- The ProcessingEngine interacts with the IndexStore to perform indexing and search operations, utilizing its methods to update and query the index as needed.

- While the ProcessingEngine is responsible for executing the core functionalities, the IndexStore serves as a persistent storage mechanism for the index data, maintaining consistency and integrity across different sessions of the Engine.

Benefits:

- The layered architecture promotes separation of concerns, making it easier to maintain, extend, and debug the File Retrieval Engine.

- Each component operates independently, allowing for parallel development and testing of functionalities.

- Modularity facilitates code reuse, as individual components can be reused in other projects or integrated into larger systems.

The layered architecture of the File Retrieval Engine provides a robust foundation for efficient indexing and retrieval of files. By encapsulating distinct functionalities within each component, the architecture promotes scalability, maintainability, and ease of use, ensuring a seamless user experience.

File Retrieval Engine Specification:

The File Retrieval Engine is designed to support various commands aimed at indexing files from specified datasets and processing search queries to retrieve relevant files based on user input. The specification outlines the functionality and behavior expected from each command supported by the Engine.

Commands Supported:

1. index <dataset path>:

- Functionality: The index command triggers the Engine to crawl and locate all files within the specified dataset path, building an index from these files.

- User Interaction: Users input "index <dataset path>" where <dataset path> represents the path to the dataset containing files to be indexed. Upon execution, the Engine traverses the specified dataset, extracts terms from the files, and updates the index accordingly.

2. search <AND query>:

- Functionality: The search command enables users to execute search queries using AND logic, retrieving files that contain all terms specified in the query. The returned files are sorted based on the total occurrence frequency of the query terms in each file, with the top 10 files displayed to the user.

- User Interaction: Users enter "search <AND query>" where <AND query> represents the search query composed of multiple terms separated by the logical AND operator. For example, entering "cats AND dogs" triggers the Engine to retrieve files containing both "cats" and "dogs," sorting them by occurrence frequency and displaying the top 10 results.

3. quit:

- Functionality: This command gracefully closes the application, terminating the File Retrieval Engine.

- User Interaction: Users can enter "quit" in the command-line interface provided by the AppInterface component to exit the Engine.

Execution Flow:

- Upon receiving a command from the user via the command-line interface provided by the AppInterface component, the Engine processes the command and performs the corresponding operation.

- For indexing commands, the Engine traverses the specified dataset path, extracts terms from each file, and updates the index stored in the IndexStore component.

- During search operations, the Engine parses the user's query, retrieves relevant files from the index based on the specified terms, sorts them by occurrence frequency, and presents the top 10 results to the user.

Benefits:

- The specified commands provide users with a straightforward and intuitive interface to interact with the File Retrieval Engine, enabling them to index datasets and perform efficient searches.

- By supporting AND queries, users can retrieve highly relevant files that contain all specified terms, facilitating precise information retrieval.

- The Engine's functionality aligns with common user expectations for file retrieval systems, enhancing usability and user satisfaction.

The File Retrieval Engine's specification outlines a set of commands that enable users to index datasets and execute search queries effectively. By providing clear functionality and user

interactions, the Engine simplifies the process of file retrieval and supports precise information retrieval based on user-defined criteria.

Datasets:

The evaluation of the File Retrieval Engine's performance is conducted using the Gutenberg Project Datasets, which consist of ASCII TXT documents representing various books collected from the internet. These datasets serve as a standardized benchmark for assessing the Engine's indexing efficiency and search effectiveness across different dataset sizes.

Key Features of the Datasets:

1. Dataset Variety:

- The Gutenberg Project Datasets comprise a diverse collection of books covering various genres, subjects, and authors. This diversity ensures a comprehensive evaluation of the Engine's ability to handle different types of textual content.

2. File Structure:

- Each dataset is organized into multiple folders, with each folder containing a collection of ASCII TXT documents representing individual books. This hierarchical structure enables systematic indexing and retrieval of textual data from distinct sources.

3. Increasing Size and File Counts:

- The datasets are designed to vary in size and number of files, allowing for a progressive evaluation of the Engine's performance under different workload conditions. The increasing size and file counts challenge the Engine's scalability and efficiency in handling larger datasets.

Dataset Characteristics:

1. Content Complexity:

- The TXT documents within the datasets may vary in terms of content complexity, including vocabulary richness, sentence structures, and thematic diversity. This variability tests the Engine's ability to index and retrieve information from diverse textual sources effectively.

2. Language Coverage:

- The datasets may contain texts in multiple languages, reflecting the global nature of the Gutenberg Project's collection. This language diversity presents challenges in terms of language processing and indexing for multilingual support, if applicable.

3. Metadata Availability:

- Each TXT document may be accompanied by metadata providing information about the book title, author, publication year, and other relevant details. Leveraging this metadata can enhance the Engine's search capabilities by enabling faceted search and result filtering based on metadata attributes.

Evaluation Strategy:

1. Indexing Performance:

- The Engine's indexing performance is evaluated by measuring the time taken to index each dataset. Wall time measurements are captured to assess the total execution time for indexing, while indexing throughput is calculated by dividing the total dataset size by the indexing execution time, yielding a metric in MB/s.

2. Search Effectiveness:

- The Engine's search effectiveness is assessed through query-based evaluations using predefined test queries. The relevance and ranking of search results are analyzed to determine the Engine's ability to retrieve relevant files based on user-specified search criteria.

The Gutenberg Project Datasets provide a standardized and diverse set of textual data for evaluating the File Retrieval Engine's performance across different dataset sizes, content types, and languages. By leveraging these datasets, the Engine's indexing efficiency, search effectiveness, and scalability can be systematically assessed, leading to informed optimizations and enhancements for real-world deployment scenarios.

5. Performance Evaluation:

The performance evaluation of the File Retrieval Engine involves assessing its efficiency in indexing files from the Gutenberg Project Datasets and processing search queries. This section discusses the methodology, metrics, and insights derived from the performance evaluation process.

Methodology:

1. Indexing Execution:

- The Engine is executed five times, each time indexing a different dataset from the Gutenberg Project Datasets. The indexing process involves traversing the dataset directories, parsing and processing individual TXT files, and building the index in the IndexStore component.

2. Wall Time Measurement:

- Wall time, also known as real time, is measured using system-level time tracking utilities or built-in Java libraries. It represents the actual elapsed time from the start to the end of the indexing process, inclusive of all external factors such as I/O operations and system load.

3. Indexing Throughput Calculation:

- Indexing throughput is computed by dividing the total dataset size by the wall time of the indexing process. The dataset size is measured in megabytes (MB), and the throughput is expressed in megabytes per second (MB/s), indicating the rate at which data is indexed.

Q.1. What is the difference between the wall time and the CPU time? What method or function did you use to measure the wall time?

Answer: Difference between Wall Time and CPU Time:

Wall time represents the total elapsed time from the start to the end of a process, encompassing all system-level activities such as I/O operations, context switches, and waiting times. On the other hand, CPU time measures the actual time spent executing CPU instructions, excluding any time spent waiting for external events. In the context of the File Retrieval Engine, wall time provides a comprehensive view of the indexing process's real-world performance, considering all factors impacting execution time, including disk I/O and system load. CPU time, while useful for isolating CPU-bound processing, does not account for external factors and is therefore less indicative of real-world performance.

Q.2. How big are Dataset 1 and Dataset 5, measured in MB and MiB (Megabytes and Mebibytes, respectively)?

Answer: Dataset sizes are quantified in both megabytes (MB) and mebibytes (MiB) to provide clarity and standardization. While megabytes are based on the decimal system (1 MB), mebibytes adhere to the binary system (1 MiB = 2^20 bytes), resulting in slightly different values for the same dataset size. By presenting dataset sizes in both units, users gain a comprehensive understanding of the data volume involved, accounting for differences in measurement standards.

Q.3. What data structure(s) did you use to implement the IndexStore component?

Answer: The Index Store component employs HashMap data structures to efficiently store and manage the index. HashMaps offer constant-time average performance for key-value insertion, retrieval, and deletion operations, making them well-suited for indexing tasks requiring fast access to term-document mappings. Additionally, HashMaps dynamically resize to accommodate varying index sizes, ensuring scalability and adaptability to changing data volumes.

Q.4. What is the difference between compute-intensive, memory-intensive and IOintensive applications?

Answer: Distinctions between Compute-Intensive, Memory-Intensive, and IO-Intensive Applications:

- Compute-Intensive Applications: These applications primarily focus on CPU-bound processing tasks, such as mathematical calculations, simulations, and cryptographic operations. They heavily rely on computational resources to perform complex computations and often exhibit high CPU utilization during execution.

- Memory-Intensive Applications: Memory-intensive applications prioritize efficient memory utilization and management to handle large datasets or concurrent operations. They frequently access and manipulate data stored in memory, requiring optimized memory allocation and access patterns to minimize latency and maximize throughput.

- IO-Intensive Applications: IO-intensive applications prioritize efficient handling of input/output operations, such as reading from and writing to disk storage, network communication, and file

processing. They often involve frequent disk I/O operations, necessitating optimized file access, buffering, and caching mechanisms to minimize latency and enhance performance.

Q.5. Is the Processing Engine component compute-intensive, memory-intensive or IO-intensive, and why?

Answer: The Processing Engine component of the File Retrieval Engine can be categorized as IO-intensive due to its heavy reliance on input/output (IO) operations during both indexing and search processes. In indexing, the Engine reads and processes a large volume of textual data from dataset files, which involves frequent interactions with disk storage to access and parse individual files. Similarly, during search operations, the Engine retrieves indexed data from disk storage and performs file lookups to identify relevant documents based on user queries. These operations require significant disk I/O activity, making the Processing Engine IO-intensive.

The IO-intensive nature of the Processing Engine stems from its extensive reliance on disk storage for reading and writing data. Unlike compute-intensive tasks that primarily focus on computational processing or memory-intensive tasks that heavily utilize system memory, the Processing Engine's workload is dominated by interactions with disk storage. As a result, optimizing file I/O operations and minimizing disk access latency are critical for improving the overall performance and responsiveness of the ProcessingEngine.

Conclusion:

The File Retrieval Engine effectively fulfills the specified requirements by providing robust indexing and searching capabilities. Through the implementation of the App Interface, Processing Engine, and Index Store components, the Engine offers a user-friendly command-line interface for interacting with datasets and performing search queries. While the current implementation demonstrates functionality, there remains room for enhancement. Future iterations could focus on optimizing indexing and search algorithms to further improve performance and scalability. Additionally, incorporating parallel processing techniques and implementing advanced data structures may enhance efficiency, particularly for handling larger datasets. Overall, the File Retrieval Engine serves as a solid foundation for further refinement and optimization to meet evolving user needs and requirements.