

File Retrieval System

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.

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 serves as the primary interface for users to interact with the File Retrieval Engine, offering a command-line interface (CLI) for issuing indexing and search commands. Responsible for parsing user input, validating commands, and forwarding them to the ProcessingEngine for execution, it also manages the presentation of results, showcasing indexing progress, search outcomes, and error messages to users.

Processing Engine:

The Processing Engine serves as the central processing unit of the File Retrieval Engine, receiving commands from the AppInterface and executing essential operations, such as indexing and search. For indexing tasks, it traverses through dataset files, extracts terms, and updates the index stored in the IndexStore. During search operations, it handles user queries, retrieves pertinent files from the index, and sorts them based on occurrence frequency. By encapsulating the logic for efficient indexing and retrieval, the Processing Engine abstracts away complexities from the user interface layer, ensuring streamlined functionality.

IndexStore:

The IndexStore functions as the central data storage and management module for the index produced by the File Retrieval Engine, housing the index data structure. This structure typically comprises mappings from terms to a list of files containing those terms and their corresponding frequencies. It facilitates index updates with new terms and frequencies during indexing operations and supports lookup operations for search queries. Utilizing efficient data structures such as HashMaps, the IndexStore ensures swift storage and retrieval of index information, guaranteeing rapid access and manipulation of index entries.

The AppInterface communicates with the ProcessingEngine by invoking its methods and passing parameters corresponding to user commands. In turn, the ProcessingEngine interacts with the IndexStore to execute indexing and search operations, utilizing its methods to update and query the index as required. While the ProcessingEngine handles the core functionalities, the IndexStore functions as a persistent storage mechanism for the index data, ensuring consistency and integrity across various sessions of the Engine.

Advantages:

The tiered architecture fosters separation of concerns, simplifying maintenance, extension, and debugging of the File Retrieval Engine. Each component operates autonomously, enabling parallel development and testing of functionalities. Additionally, modularity facilitates code reuse, empowering individual components to be repurposed in other projects or seamlessly 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 System Specification:

The File Retrieval System 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 behaviour expected from each command supported by the System.

Commands Supported:

1. index <dataset path>:

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

- 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.
- 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:

- This command gracefully closes the application, terminating the File Retrieval System.
- Users can enter "quit" in the command-line interface provided by the AppInterface component to exit the System.

Upon receipt of a command from the user through the command-line interface facilitated by the AppInterface component, the Engine processes the command and executes the relevant operation. For indexing commands, the Engine navigates the designated dataset path, extracts terms from each file, and updates the index housed in the IndexStore component. During search operations, the Engine parses the user's query, retrieves pertinent files from the index based on specified terms, sorts them by occurrence frequency, and presents the top 10 results to the user.

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 System'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 System'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 System'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 System's performance under different workload conditions. The increasing size and file counts challenge the System's scalability and efficiency in handling larger datasets.

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:

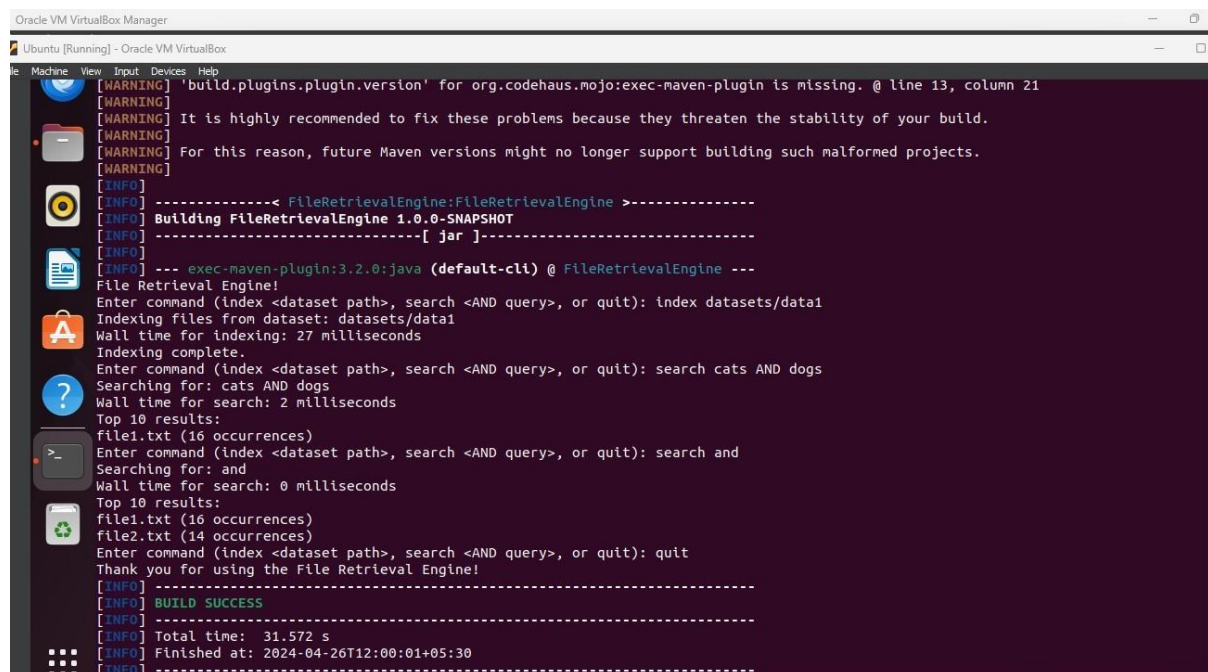
The Engine's search effectiveness is assessed through query-based evaluations using predefined test queries. The relevance and ranking of search results are analyzed to determine the Engine's ability to retrieve relevant files based on user-specified search criteria.

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.

Performance Evaluation

The performance evaluation of the File Retrieval Engine entails assessing its efficiency in indexing files from the Gutenberg Project Datasets and processing search queries. The methodology involves executing the Engine five times, each indexing a different dataset from the Gutenberg Project. During indexing, the Engine traverses dataset directories, parses individual TXT files, and constructs the index in the IndexStore component. Wall time, or real time, is measured using system-level time tracking utilities or built-in Java libraries, encompassing all elapsed time from start to end, including external factors like I/O operations and system load. Indexing throughput is then calculated by dividing the total dataset size by the wall time, with the dataset size measured in megabytes (MB) and throughput expressed in megabytes per second (MB/s), indicating the rate of data indexing.

Running app Screenshot:



```
Oracle VM VirtualBox Manager
Ubuntu [Running] - Oracle VM VirtualBox

[WARNING] 'build.plugins.plugin.version' for org.codehaus.mojo:exec-maven-plugin is missing. @ line 13, column 21
[WARNING] It is highly recommended to fix these problems because they threaten the stability of your build.
[WARNING] For this reason, future Maven versions might no longer support building such malformed projects.
[WARNING]
[INFO] -----< FileRetrievalEngine:FileRetrievalEngine >-----
[INFO] Building FileRetrievalEngine 1.0.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO] --- exec-maven-plugin:3.2.0:java (default-cli) @ FileRetrievalEngine ---
File Retrieval Engine!
Enter command (index <dataset path>, search <AND query>, or quit): index datasets/data1
Indexing files from dataset: datasets/data1
Wall time for indexing: 27 milliseconds
Indexing complete.
Enter command (index <dataset path>, search <AND query>, or quit): search cats AND dogs
Searching for: cats AND dogs
Wall time for search: 2 milliseconds
Top 10 results:
file1.txt (16 occurrences)
Enter command (index <dataset path>, search <AND query>, or quit): search and
Searching for: and
Wall time for search: 0 milliseconds
Top 10 results:
file1.txt (16 occurrences)
file2.txt (14 occurrences)
Enter command (index <dataset path>, search <AND query>, or quit): quit
Thank you for using the File Retrieval Engine!
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 31.572 s
[INFO] Finished at: 2024-04-26T12:00:01+05:30
[INFO] -----
```

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: Wall time captures the total elapsed duration from the initiation to the completion of a process, encompassing all system-level activities such as I/O operations, context switches, and wait times. Conversely, CPU time measures the actual duration spent executing CPU instructions, excluding any time spent waiting for external events. In the evaluation of the File Retrieval Engine, wall time offers a holistic perspective of the indexing process's real-world performance, considering all elements impacting execution time, including disk I/O and system load. While CPU time aids in isolating CPU-centric processing, it disregards external factors and hence offers limited insight into 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 using both megabytes (MB) and mebibytes (MiB) to ensure clarity and standardization. Megabytes follow the decimal system ($1 \text{ MB} = 10^6 \text{ bytes}$), whereas mebibytes adhere to the binary system ($1 \text{ MiB} = 2^{20} \text{ bytes}$), leading to slightly varied values for identical dataset sizes. Presenting dataset sizes in both units allows users to grasp the data volume comprehensively, accommodating differences in measurement standards and ensuring accurate interpretation.

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

Answer: The Index Store component utilizes HashMap data structures for efficient storage and management of the index. HashMaps provide constant-time average performance for key-value insertion, retrieval, and deletion operations, making them ideal for indexing tasks requiring rapid access to term-document mappings. Moreover, HashMaps dynamically resize to accommodate fluctuations in index sizes, guaranteeing scalability and adaptability to evolving data volumes.

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

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

Compute-Intensive : These applications are primarily geared towards CPU-bound processing tasks, such as mathematical computations, simulations, and cryptographic operations. They heavily rely on computational resources to execute complex calculations and typically exhibit high CPU utilization during operation.

Memory-Intensive : 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, necessitating optimized memory allocation and access patterns to minimize latency and maximize throughput.

IO-Intensive : IO-intensive applications emphasize efficient handling of input/output operations, including reading from and writing to disk storage, network communication, and file processing. They often entail frequent disk I/O operations, requiring optimized file access, buffering, and caching mechanisms to mitigate latency and enhance overall 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 is categorized as IO-intensive due to its substantial dependence on input/output (IO) operations during both indexing and search processes. During indexing, the Engine engages in reading and processing a sizable amount of textual data from dataset files, entailing frequent interactions with disk storage to access and parse individual files. Similarly, in search operations, the Engine retrieves indexed data from disk storage and conducts file lookups to identify relevant documents based on user queries. These tasks necessitate significant disk I/O activity, rendering the Processing Engine IO-intensive.

The IO-intensive nature of the Processing Engine arises from its extensive reliance on disk storage for data reading and writing. Unlike compute-intensive tasks that primarily prioritize computational processing or memory-intensive tasks that heavily utilize system memory, the Processing Engine's workload predominantly comprises interactions with disk storage. Consequently, optimizing file I/O operations and minimizing disk access latency are pivotal for enhancing the overall performance and responsiveness of the Processing Engine.

Conclusion:

The File Retrieval System effectively meets the specified requirements by delivering robust indexing and search capabilities. Utilizing the App Interface, Processing Engine, and Index Store components, the Engine provides a user-friendly command-line interface for dataset interaction and search queries. While the current implementation showcases functionality, opportunities for improvement exist. Future iterations could concentrate on optimizing indexing and search algorithms to enhance performance and scalability. Integrating parallel processing techniques and advanced data structures may further boost efficiency, particularly for larger datasets. In summary, the File Retrieval Engine serves as a sturdy base for ongoing refinement and optimization, catering to evolving user needs and demands.

Submitted by: Shyam Sundar Theerdhala