Searching ArchWiki with Lucene

Papastathopoulos Kostas
Department of Information and Electronics Engineering
International Hellenic University
Thessaloniki, Greece
papastathopoulosko@gmail.com

***Abstract*—This document serves as a project report presenting the design and implementation of an ArchWiki indexing and search system by using Apache's Lucene. Key functionalities of the backend of the app include creating a local index of the wiki, searching it, and exposing pagination for clients among with other modifiable properties. The UI is build with dynamic UI elements, supports pagination, and theme switching and can utilizes some of the features exposed by the backend. Challenges addressed include modularity and state synchronization, while limitations include single-language support and query complexity handling. Future work aims to enhance scalability, multi-language support, and user experience features. The project demonstrates a maintainable, modular approach to integrating efficient search algorithms with a modern UI**

***Index Terms*—Lucene, Kotlin, Compose, Ktor, Jsoup, StandfordCoreNLP**

I. Introduction

Two modules have been created and are responsible for independent tasks to make the code more organized and improve maintainability for future work while also offering separation of concerns. The modules that got created are 'searchengine' and 'searchengineui'. 'searchengine' is focused on creating an index and perform searches on it, forming the backend of the system. 'searchengineui' depends and interacts with 'searchengine' to provide a user-friendly interface for interacting with it.

II. SearchEngine Module

The 'SearchEngine' module is a library responsible for handling the backend processes of retrieving, indexing, searching, and ranking documents. It takes advantage of multiple third-party libraries, such as Ktor for downloading the wiki's html pages, Jsoup to parse the HTML pages and remove unnecessary tags and attributes Stanford NLP for doing lemmatization and finally Apache's Lucene for tokenazation, indexing and searching. It contains four classes that will get further analyzed in the following sections.

A. *SearchEngine Class*

This class serves as a Facade, providing a unified interface to the complex operations of the logic module. It encapsulates the internal implementation details and exposes a streamlined API for the UI module, adhering to principles of modularity and separation of concerns. When it is initialized it checks if the provided input directory exists (defaults to cwd/data/site/ if not provided) and if not, downloads the arch-wiki-docs package using ktor. Then uses luben's zstd-jni to decompress the Zstandard ('.zst') part of the file, and finaly apache common compression library to extract the ('.tar') part of the file. Finally, filters the results and keeps only the directory containing the english pages, the images and the singular css file that exists so that opening a page will have the same formating as the online version of the wiki. In case where the directory existed these steps are entirely skipped.

B. *Lucene Class*

This class represents the core logic of the module, implementing the domain-specific responsibilities such as searching, indexing. It is the primary component responsible for executing the critical operations of the module, adhering to principles of separation of concerns and encapsulation. By isolating this functionality, the system ensures high reusability, testability, and maintainability. It has two functions, createIndex and searchIndex.

- **createIndex :** It is utilizing EnglishAnalyzer for English language processing (stop words) and tokenizing the input. It checks in the existing indices to avoid re-indexing by quering the path of the file. Creates and updates the Lucene index while providing live progress updates by calling the lambda parameter with the updated values. The file processing (html, lema, stop words, tokenazation) occurs asynchronously with multiple threads using Kotlin coroutines to speed up the creation of the index.
- **searchIndex** This function builds Lucene queries based on user inputs by Combining QueryParser, PhraseQueries, TermQueries, PrefixQueries and FuzzyQueries to match user input against index fields. The queries are wrapped in BoostQueries to give each of the query a different weight and impove the search results. It also tries to match snippets of the query for every document and return a string for highlighting by using the Lucene Highlighter, in case of a failure a standard string is returned. Results can be sorted by the modification date by passing true to the sortByDate parameter. Users of the library can set custom result restrictions by passing a value to resultsPerPage (default limit is 10) for less cluttering of the search results. Supports BM25 (default) or TF-IDF-based search via Lucene's customizable similarity implementations.

C. *Lemmatizer Class*

This class holds a single pipeline object with custom properties and exposes a lemmitaze method which takes an input and lemmitizes it with sayed properties. It is used by the HTML class to perform text preprocessing after removing the HTML tags and by the Search function of the Lucene class to make sure we apply the same word transformations to the index and the searches.

D. *HTMLParser Class*

This class takes a single lambda that takes a string a returns a string as a parameter and stores it. It exposes a signle function called parseHtmlToDocument that takes a file argument, then by using jsoup it removes unnecessary tags that should not be indexed like navigation, scripts etc, and returns a Lucene Document that contains the following fields.

- **Title** The title element but without the " -ArchWiki" part at the end.
- **Path** The filepath were the HTML file is stored in the system.
- **Last Modification Time String** The last modification time in a string format so that it can easily be displayed.
- **Last Modification Time Long** The last modification time in a long format so that it can be used for sorting.
- **Document content** All of the important text in the document.

E. *Wrapper classes*

There are two wrapper classes in module and are both used expose information outside of the module.

- **WikiDocumentResult** A result object representing a search matched document and containing data to be displayed to the user, with data like title, maching string, score, etc.
- **LuceneWrapper** A object that holds a list containing every match as a **WikiDocumentResult**, along with the number of total matches found, and the number of total pages.

III. SearchEngineUI Module

This module is responsible for handling the user interaction by using a GUI. It has a dependency on SearchEngine module to compile and use. Uses Compose Multiplatform for building the UI along with Material 3, and Material Icons to make it look modern. The entry point and the three composable functions that act ass screens in this module will be further analyzed bellow.

A. *Main*

They entry for the ui is the 'main' composable function. It is responsible for observing the state of the searchengine library and showing the appropriate screen. When created it initializes the SearchEngine class and stores it across re-compositions as to not loose data and degrade the performance, also initializes other variables that are responsible for the state mangment of the app. Values such as isDarkMode isIndexing etc. The transitions of each screen are handled with AnimatedVisibility to animate the screen transision and create a pleasent user expirience.

B. *DownloadScreen Composable*

This composable function is called while we are retriving and extracting data. It has a dummy circular progress indicator that does not change based on the percentage of the action since the action should take no more than a couple of seconds.

C. *IndexScreen Composable*

This function takes two arguments, on string for displaying the item that's currently processing and one float that reprecents the progress in a scale of 0 to 1

D. *SearchScreen Composable*

This composable is what the user will be interacting with the most. It has a search bar that checks for updates and for every update re-executes a search. It has a button to toggle the dark theme on and off, a create index button to trigger the creation of the index, and a gear icon that animates the visibility of more advanced options. The more advanced options are, "search for title only" and "search with tf-idf" wich are both off by default.

IV. Future Work

- Add advanced filter options (e.g., date range, file size) to refine search queries.
- Explore features like autosuggestion and query history in the search bar.
- Multi-language support for lemmatization, indexing searching.
- Make the parsing and index structure more abstract for uses with other wiki pages.
- Create tests for searching, currently only basic functions are tested.

References

[1] GitHub Repository for ComposeSearchEngine
[2] AI Log Documentation on GitHub