

Master's Thesis

Design and Implementation of a Configurable Preferential Web Search Engine

Alhajras Algdairy

Examiner: Prof. Dr. Hannah Bast

Advisers: M. Sc. Natalie Prange



Albert-Ludwigs-University Freiburg

Faculty of Engineering

Department of Computer Science

Chair of Algorithms and Data Structures

October 31st, 2023

Writing Period

08.05.2023 – 31.10.2023

Examiner

Prof. Dr. Hannah Bast

Second Examiner

Prof. Dr. Thomas Brox

Advisers

M. Sc. Natalie Prange

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.

I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date

Signature

Abstract

In today's business landscape, online content analysis plays a crucial role in making informed decisions. However, existing solutions in this space are often proprietary, needing more transparency in their code and flexibility for customization and scalability. In this thesis, I introduce Scriburg, a prototype for a large-scale search engine that can be configured to suit specific user preferences. Scriburg has been developed to crawl efficiently and index web content, offering a range of settings through a user-friendly interface. The system is designed to handle small and large websites, making it adaptable to various scenarios. Scriburg is particularly well-suited for situations where users have a specific interest in a subset of the web, such as a particular domain or a select group of web pages. Despite the significance of web search engines, there remains a need for further academic research focused on developing and designing open-source search engine solutions that can be readily used and extended by a wide range of users.

Acknowledgments

First and foremost, I would like to thank:

- My parents for supporting me during the master's program.
- My wife for her love and support.
- Prof. Dr. Hannah Bast for accepting my topic and for her supervision.
- M. Sc. Natalie Prange for her competent opinions and suggestions.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Task Definition	2
1.3	Contribution	2
1.4	Chapter Overview	3
2	Related Work	5
2.1	Existing Web Crawlers	5
2.2	High Level Google Architecture	7
3	Background	11
3.1	Web Search Engine	11
3.1.1	Requierments and Features	11
3.2	Cralwer	13
3.2.1	Cralwer Specifications	13
3.2.2	Crawler Architecture	14
3.2.3	Crawler Data Structure	15
3.3	Indexing	18
3.3.1	Tokenization	18
3.3.2	Document Unit	19
3.3.3	Inverted index	19
3.4	Ranking	21
3.4.1	Ranking Refinements	23
3.4.2	Fuzzy search	24
4	Approach	27
4.1	Software Architecture	27

4.2	Crawler Implmentation	29
4.2.1	Links Data Structure	30
4.2.2	Practical Challenges	30
4.3	Indexer Implmentation	32
4.4	User Interface Design	34
5	Evaluation	41
5.1	Testing Environment	41
5.2	Crawler	41
5.2.1	Datasets	42
5.2.2	Experiments	43
5.3	Indexer	57
5.3.1	Datasets	58
5.3.2	Metrics	58
5.3.3	Experiments	59
5.4	User Experience	63
6	Conclusions and Future Work	65
	Bibliography	67

List of Figures

1	High level view of Google web crawlers archeticture [?].	8
2	An overview of a generic search engine system.	12
3	A generic web crawler architecture overview [?].	14
4	Illustration of the Breadth First Search or BFS for a Graph algorithm. [?]	16
5	Illustration of various images	17
6	An illustration of an inverted index featuring three documents. All tokens are included in this example, and the sole text normalization applied is converting all tokens to lowercase. Queries that involve multiple keywords are resolved using set operations. [3]	20
7	High-level view of the software architicture.	28
8	An overview of the Templates table containing inspectors list project- ing the targeted feilds in a document	35
9	An overview of the runners table, showing the runners status and progress.	37
10	An overview of the indwxwea table, showing the their status and progress.	38
11	High-level view of the software architicture.	40
12	Comparing threads performance.	50
13	Threads documents distribution.	51

List of Tables

1	Documents sample	22
2	The first 10 tokens of the result inverted index and the scores of the docuemnts.	22
3	Crawler configurations options	36
4	Indexer configurations options	39
5	Local machine setup	41
6	Crawler configuration	44
7	Crawler configuration	44
8	Crawler configuration	48
9	Crawler results	48
10	Crawler configuration	49
11	Crawler results	49
12	Crawler configuration	52
13	Crawler configuration	53
14	Crawler configuration	53
15	Crawler results	55
16	Crawler configuration	55
17	Crawler results	56
18	Crawler results	57
19	Stack Overflow posts dataset	58
20	Stack Overflow indexing configuration, test the default settings without any changes.	59
21	Stack Overflow indexing configuration	61
22	Stack Overflow indexing configuration	61
23	Stack Overflow indexing configuration	62

24	Stack Overflow indexing configuration	63
----	---	----

1 Introduction

1.1 Motivation

Since the beginning of the Digital Revolution, known as the Third Industrial Revolution, in the latter half of the 20th century, the importance of data has increased as it became the new currency shaping the dynamics of our interconnected world. From social media platforms and e-commerce transactions to information sharing and entertainment consumption, online activities generate enormous amounts of data. The online data is sometimes referred to as the 'new oil' or the 'new currency', as it impacts almost the same economies and societies as oil. Businesses and organizations understand the power of data as they provide insight into consumer behaviour, refine business strategies, and enhance decision-making processes. Furthermore, the rise of artificial intelligence has further amplified the value of Internet data. Natural language processing (NLP)¹ is becoming a new hot topic as all the giant firms race to create their model; however, data is the fuel to power those models. The more data is collected, the better the model can become. Consequently, collecting, analyzing, and leveraging internet data has become a cornerstone of competitiveness, innovation, and progress in the digital age.

Internet data can be harvested by using automated software programs called Web crawlers, also known as web spiders or web bots. Their main goal is to discover, retrieve, and index² information from websites. The applications and use cases of internet crawlers are diverse and valuable; however, the main application that sparked this thesis was market research. Businesses use web crawlers to collect data about their competitors, market trends, and consumer opinion. This information helps in making informed business decisions.

¹Natural language processing is an interdisciplinary field that enables computers to understand and manipulate speech.

²Indexing involves the storage of document indexes to enhance the speed and performance of locating relevant documents in response to a search query.

Search engines like Google, Bing, and DuckDuckGo excel at web crawling and indexing, but businesses, especially in E-commerce, require competitive pricing insights beyond standard search results. Google’s parameters, including brand visibility, user location, SEO³ proficiency, and hidden variables, impact document rankings. Different search engines yield unique results, and companies may want to exclude parts of the internet from indexing. Customization to specific domains and use cases, like price comparison, is necessary.

Businesses often seek a subset of the internet relevant to their domain. Indexing and ranking criteria vary by use case, necessitating tailored configurations. Employing domain expertise is crucial. However, data scientists face initial setup challenges and costs. An infrastructure allowing data scientists to use adaptable scripts with minimal programming knowledge would be valuable.

1.2 Task Definition

Given a set of URLs, denoted as $U = \{U_1, U_2, U_3, \dots U_n\}$, the objective is to identify, for each URL in U , a set of prospective URLs, $U_{new} = \{U_1, U_2, U_3, \dots U_n\}$, to augment the existing URLs in U , creating $U_{tot} = U \cup U_{new}$. Furthermore, for each URL U , the goal is to specify the desired documents for downloading and indexing, forming a collection denoted as $D = \{D_1, D_2, D_3, \dots D_n\}$.

After completing the web crawling process and successfully retrieving the document set D , the subsequent step involves indexing all these documents.

All crawling and indexing processes must be easily configurable and adjustable through a user-friendly interface. Moreover, users should be able to search against the indexed documents. Given a user input query q , we define relevant documents as the subset of the crawled and indexed documents $R \subset D$.

1.3 Contribution

In this thesis, we aim to answer the following questions:

- What are the challenges and bottlenecks to creating a scalable, configurable generic search engine?

³SEO, or Search Engine Optimization, is the practice of optimizing online content and websites to improve their visibility and ranking in search engine results.

- Can we outperform a similar tool like ParseHub?
- How does changing the configurations provided by the user interface affect the results in the crawling and indexing accuracy?
- Can we create a decent User Interface UI that intuitively allows users to crawl and index targeted websites from the internet?
- How well do crawlers react to different websites with different DOM⁴ structures?
- Can we integrate the indexing and crawling processes in the same tool?
- Can we find meaningful evaluation metrics for the implemented search engine?

1.4 Chapter Overview

The organization of this thesis is as follows:

- Chapter 2 dives into the prior research that serves as the foundation for this thesis.
- Chapter 3 clarifies the essential theoretical background for understanding this thesis’s fundamental concepts.
- Chapter 4 presents an overview of the system’s architecture and design, detailing the implementation of the crawling process and addressing the challenges faced in the process.
- Chapter 5 discusses the process of crawling and indexing datasets used for evaluation and the methodologies employed for conducting the evaluation.
- Chapter 6 summarises the findings from the thesis experiments and outlines future opportunities for further enhancements and research.

⁴The DOM (Document Object Model) is a programming interface that represents the structure of a web page as a tree of objects, enabling developers to interact with and manipulate web page elements using scripting languages like JavaScript.

2 Related Work

This chapter will examine commercial and open-source solutions that provide functionalities similar to Scriburg's. In section 2.1, a collection of currently available web crawlers will be presented. In section 2.2, we will provide an overview of the architecture of the Google search engine, as some of its fundamental architectural concepts will be adapted with certain modifications.

2.1 Existing Web Crawlers

The concept of web crawling dates back to the early 1990s when the World Wide Web was still in its infancy.

WebCrawler, created by Brian Pinkerton in 1994 [?], is considered the first actual web crawler-powered search engine. One of the significant innovations of WebCrawler was its full-text¹ searchability. This capability made it famous and highly functional. It continues to operate as a search engine, although not as popular as Google, Yahoo and Bing.

Over the past few decades, web crawlers have evolved significantly, adopting various designs and implementations to crawl and index the internet. They have adapted to address emerging challenges and complexities, such as handling dynamic content, user interactions, authentication, robots.txt files, and ethical concerns. Notably, Google is a state-of-the-art search engine dominating the entire market. As of 2023, Google controls a market share of approximately 84%[?], which surpasses its closest competitor, Bing, by a significant margin of 75%. Bing, a well-known search engine developed by Microsoft, has garnered increased attention recently. Nevertheless, given Google's dominance over other search engines, it is reasonable to focus on its solutions and overlook the rest. Furthermore, Google makes some research papers

¹Full-text search involves electronically searching through extensive text data and retrieving results that contain either some or all of the words from the query.

available online, a practice that has yet to be observed among other search engines like Bing, making it easier to study.

Although the previously mentioned crawlers offer a wide range of features and are used as generic crawlers to fetch all web pages from the entire internet, they still need to be more general and can not be used and configured to specific personal use cases. Moreover, the most powerful search engines are not free; nobody can clone and modify as they wish. In section 2.2, we will understand Google’s robust infrastructure, which we will use as a starting point for the Scribug search engine. However, we must extend it by adding a user interface to make it configurable.

Since using a general search engine like Google directly is currently not an option, data scientists employ various tools to crawl and parse internet content. Each tool has its advantages and disadvantages, depending on distinct use cases. The following list summarizes several widely recognized crawling tools and explains how the proposed solution in this thesis distinguishes itself from them.

Beautiful Soup: Beautiful Soup is an open-source library that stands out as a widely used web scraping library that simplifies retrieving data from HTML and XML documents. Beautiful Soup demonstrates exceptional proficiency in parsing HTML documents, streamlining the task of retrieving particular components like headings, paragraphs, tables, and links. Beautiful Soup is not a search engine. It lacks the most fundamental search engine components; hence, it requires programming skills and can only be used to implement a search engine. Beautiful Soup can only parse the first seen page HTML version. Meaning it does not include the Javascript code. This is bad as most modern web pages use Javascript heavily to improve the page’s latency. For example, pagination will be an issue for Beautiful Soup.

Scrapy: It is an open-source, powerful and flexible tool that easily crawls and parses different websites. It allows the creation of custom spiders to crawl multiple pages. Easy to scale makes it suitable for large projects. This tool is perfect for programmers but not for non-technical users, as it requires good knowledge of Python programming. With Scribug, we aim to reduce the programming workload and save time by offering a user-friendly interface that can be effortlessly configured for individual websites.

Selenium: It is an open-source, robust, and adaptable solution for web scraping, enabling the automation of browser actions, interaction with web pages, and data extraction from online sources. It shares some features with the BeautifulSoup as it is an excellent tool for parsing the HTML DOM. Still, it also overcomes the issue previously mentioned about rendering Javascript and supporting dynamic contents as paginations. Interactive browser automation makes it easy to mimic the user's behavior, which makes it easier to navigate toward hidden content that requires events and human interactions. Selenium alone can not be used as a search engine; however, it will be used in this thesis as a fundamental tool for the search engine implemented. Its primary role in the implementation will be loading pages and parsing HTML.

ParseHub: ParseHub stands out as a web crawler tool with an intuitive User Interface, making it a preferred choice for many data scientists. Its most significant advantage lies in its point-and-click interface, simplifying the process of data extraction. ParseHub have both free and paid plans, with the free version allowing users to scrape up to 200 pages per run. While this limitation may prove a bit slow for professional crawling, it suits personal use admirably. However, this tool lacks the ability to fine-tune crawling algorithms and lacks an indexing feature. Given its similarity to the solution implemented in this thesis, it will serve as a valuable point of comparison in the evaluation chapter.

2.2 High Level Google Architecture

As highlighted in the prior section, Google's foundational architecture serves as the blueprint for designing Scriburg. Google's search engine architecture provides a comprehensive framework for building a scalable search engine, making it an ideal starting point for any research in this domain. Most code within the Google search engine was developed in C or C++ to ensure efficiency and compatibility with operating systems like Solaris and Linux [?]. On the contrary, Scriburg is implemented in Python. This choice was made because, in general, Python has a milder learning curve compared to C and C++, making it more appealing to the majority of data scientists for adjustments and modifications.

Google employs a distributed crawler system to retrieve web pages from the in-

ternet. The URL server maintains a list of discovered URLs that require crawling, effectively serving as a load balancer by dispatching these URLs to available crawlers. The crawlers then download the required documents from the web pages, assign a unique identifier known as a "docID" to each page, and store the page content on Store servers. Subsequently, the Store servers compress and archive the pages in a repository.

The next phase involves the indexer component, which decompresses the pages and parses their content. Each document transforms into a set of words referred to as "hits," where each hit records the word and its position within the document. These hits are later organized into "barrels" by the indexer. Furthermore, the indexer collects links within the crawled pages and maintains them in an anchor file. This anchor file contains information about the links and their interrelationships [?].

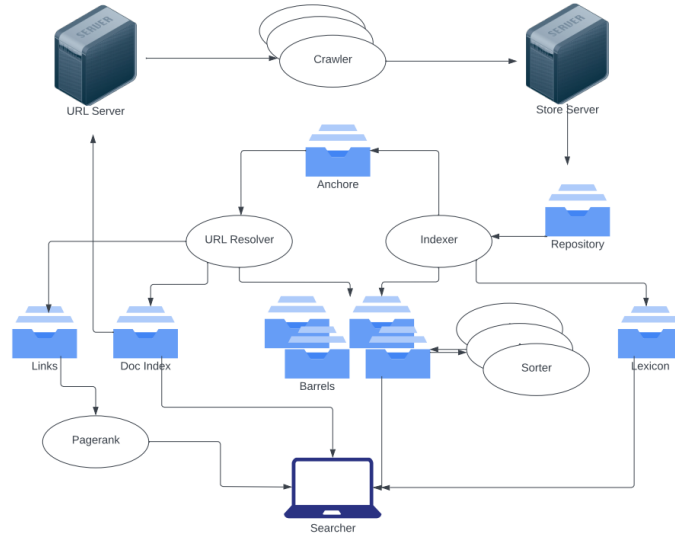


Figure 1: High level view of Google web crawlers archeticture [?].

The URLresolver reads the links from the anchors' file and converts the relative URLs into absolute URLs. The URLs are then assigned to their docID. The links database saves pairs of docIDs that will be used to compute PageRanks for all the documents.

Initially organized by docID, the barrels are then rearranged by the sorter based on wordID. This process generates an inverted index. Moreover, the sorter generates

a list of wordIDs and corresponding offsets within the inverted index.

3 Background

This section tackles the fundamental principles and groundwork of the theory encompassing concepts, terminology, and methodologies related to search engines as applied within this thesis. Section 3.1 dives into the essential components and characteristics required to implement the search engine discussed in this thesis. Section 3.2 provides a comprehensive examination of the crawler's specifications and architecture. Section 3.3 offers an in-depth explanation of the fundamental indexing terms and concepts essential to this thesis, while Section 3.4 explores the ranking score used in this research.

3.1 Web Search Engine

Web Search Engine is software that collects information from the web and indexes them efficiently to optimize the searching process by the end user. When users enter their queries to ask for information, the engine performs queries, looks up the pre-built organized index, and returns relevant results. Search Engine Results Pages, known as SERPs, present the returned results. The result is then ranked based on predefined criteria.

Web search engines use web crawlers or spiders to collect and harvest the internet, jumping from one page to another. Each page can contain several links. The crawler's task is to find the links, visit them, and harvest them. Followed by crawlers, indexing is the next process where information is organized and optimized for search.

3.1.1 Requierments and Features

Regardless of their implementation and design, all search engines share certain features and prerequisites crucial for their effectiveness. Below is a compilation of the most essential features:

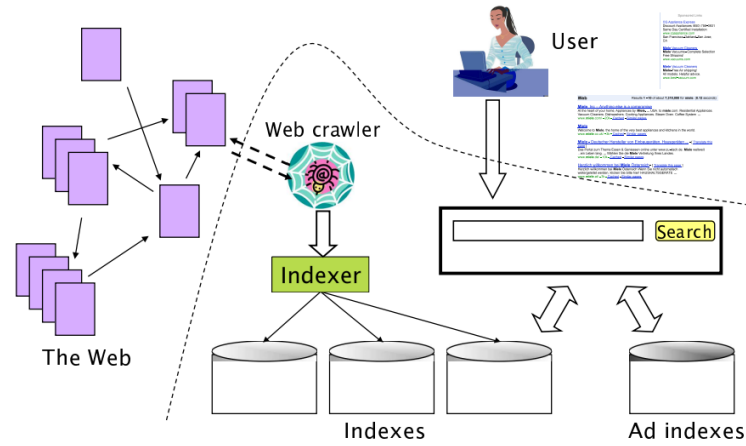


Figure 2: An overview of a generic search engine system.

Web Crawling and Indexing: As shown in Figure 2, the initial step in the search engine’s operation is web crawling. Crawlers initiate the process by connecting to the web and downloading the required pages. Subsequently, indexing comes into play, where the downloaded files are organized and indexed to enhance querying and search efficiency. Parsing the downloaded pages can be carried out in either the crawling phase or during indexing, and Scriburg, this parsing occurs during the crawling process. It is worth noting that, in Scriburg, pages are not downloaded; the targeted documents are parsed, stored in the database, and the pages themselves are discarded.

Ranking and Relevancy: As indicated in Figure ??, when users input a query to search for relevant documents, they face the Search Engine Results Pages (SERP). Users typically focus on the top results while overlooking the lower results. Hence, we must maintain relevancy. Ensuring relevance is a challenging task. Ranking the discovered documents and prioritizing the most pertinent documents at the top and the less relevant ones further down is crucial.

Scalability and Performance: A distributed system is essential for managing the extensive data and traffic demands. A load balancer is critical in distributing the crawling tasks efficiently among nodes and threads. In this, we will discuss the implemented loading balance mechanism to distribute the

crawling tasks among the crawlers.

3.2 Cralwer

Web crawler or spider is a software which gathers pages information from the web, to prived the necessary data to the Indexer to build a search engine. The essential role of crawlers is to effectively and reliably collect as much information from the web as possible. This thesis invests more time on this component than the Indexer as it serves as the bottleneck to the Search engine performance. There are different types and categories of crawlers. The first category is **Universal or Broad crawler**. This category of web crawlers does not confine itself to webpages of a specific topic or domain; instead, they continuously traverse links without limitations, collecting all encountered webpages. Google and Bing use this type of crawler. The second category is called **Preferential crawler (Focused crawler)**. Focused crawlers target specific topics, themes, or domains [?]. They are designed to gather information from a particular domain or subject area, providing specialized search results. In this thesis, a Focused crawler has been implemented and used.

3.2.1 Cralwer Specifications

Crawlers can display a diverse range of features and specifications. Nevertheless, certain essential elements must be incorporated, while others are critical for ensuring a reliable and functional crawler. Further details can be explored in the book referenced as [?].

Robustness: Web crawlers can be fragile and easy to break due to the nature of the dynamic contents on the web and the internet connection. Crawlers may encounter broken links, leading to errors and incomplete indexing. Some websites may block or ban crawlers' IP addresses if they perceive them as causing too much traffic or disruption. Web crawlers must identify those edge cases and obstacles and tackle them.

Politeness: The crawler implementation can be unintentionally dangerous if incorrectly designed. A Denial of service DoS and a Distributed Denial of service DDoS attacks can occur due to an irresponsible crawler implementa-

tion. Hence, crawlers must respect website policies and avoid breaking up web services and loading the servers.

Performance and Efficiency: The crawling system should use various resources, such as processing power, storage capacity, and network bandwidth. Moreover, the crawler should be able to function in distributed microservices across multiple machines.

Freshness: Obtaining recent versions of previously accessed pages, ensuring the search index remains updated.

3.2.2 Crawler Architecture

The simple crawler architecture includes the following fundamental modules, as shown in the following Figure ?? . The Fetch module communicates with the internet and collects the pages passed by the URL Frontier module using HTTP requests from the URLs. The URL frontier module contains a list of the URLs that need to be fetched by the Fetch module. Parsing module that takes the page content found by the Fetch module and parses the page content to find the next links to be passed to the URL frontier and also to parse any value needed from the page, like text and images. The following step involves screening the parsed document to eliminate previously visited URLs, duplicate content, and pages that are prohibited by the website. The Domain Name System (DNS)¹ resolution module identifies the web server from which to retrieve the page indicated by a given URL [?]. The DNS will be excluded in this thesis and will not be discussed.

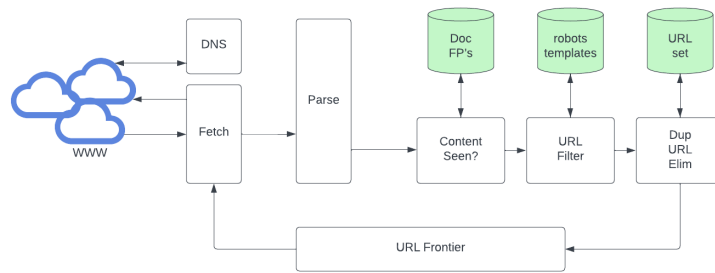


Figure 3: A generic web crawler architecture overview [?].

¹The Domain Name System (DNS) is a distributed naming system for internet resources, linking information to domain names.

The crawling process begins with adding a seed URL to the URL frontier as a starting point. The crawler retrieves and stores the corresponding page for parsing. The page's textual content, embedded links, and images are extracted during parsing, with the content prepared for use by the search engine's indexer. Each parsed link undergoes filtering to determine if it is eligible for inclusion in the URL frontier.

Following parsing, a filtering process is essential. Firstly, the content's uniqueness is verified using a fingerprint, often a checksum stored in Doc FP's database. Next, newly parsed URLs are filtered based on various criteria, such as excluding URLs outside the target country or restricted URLs. Domain administrators can specify additional filtering rules, often outlined in a Robots.txt² file. The Robots Exclusion Protocol (Robots.txt) file is a widely recognized standard websites use to communicate which parts of the site are accessible to web crawlers and other web robots.

The Robots.txt file can be obtained at the start of the crawling process and cached for efficiency, assuming it will not change during crawling. This approach is more efficient than making repeated HTTP requests for the file, reducing the number of requests and server load. Including Robots.txt in the crawling process aligns with the politeness guidelines discussed in the crawler specifications section 3.2.1.

3.2.3 Crawler Data Structure

Scriburg employs two distinct data structures for its crawling implementation: it utilizes Breadth First Search (BFS) and Depth First Search (DFS).

Breadth First Search (BFS): Considering the link planned for crawling as a vertex, it is worth noting that web pages can be conceptualized as graphs rather than trees. In contrast to trees, graphs can include cycles, which means we might revisit the same vertex. For instance, a basic illustration of this is the home page link, which essentially represents a self-loop in a graph. We can maintain two distinct data structures to prevent processing a vertex multiple times: "Visited" and "Not Visited" vertices. The "Visited" vertices can be stored in a hashmap where the link serves as the key, and the boolean value represents whether the link has already been visited (true for visited, false for unvisited). The second data structure is a queue containing links that still need to be visited.

²More information about Robots.txt can be found in the following link: <https://en.wikipedia.org/wiki/Robots.txt>

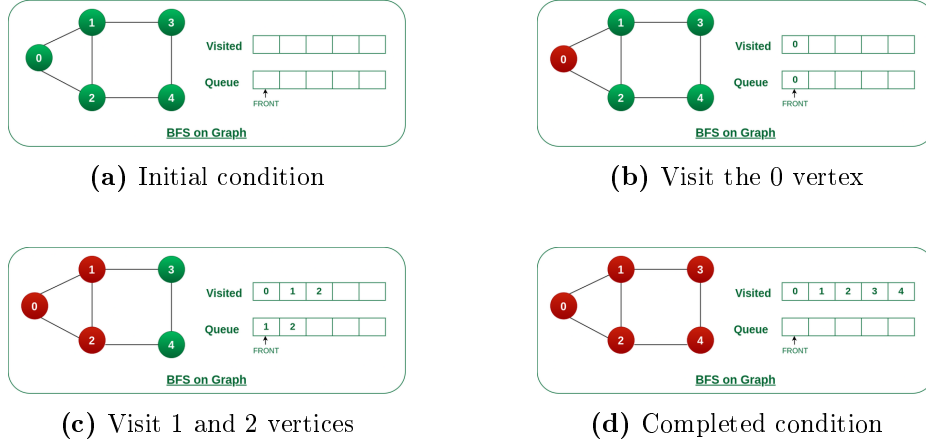


Figure 4: Illustration of the Breadth First Search or BFS for a Graph algorithm. [?]

The traversal proceeds from the root by visiting all the nodes within a specific level before moving on to the next level. This is accomplished by employing a queue. The unvisited nodes adjacent to the current level are enqueued, while the nodes in the current level are marked as visited and dequeued. One way to perceive this is by initiating the crawling process from the seed URL page, which serves as the root node, and the next level represents all the links discovered within the root page.

Figure 4 illustrates the BFS algorithm in operation to enhance visual understanding. Let us visualize a seed URL, denoted as vertex 0. The seed URL page contains two additional links, 1 and 2. The page represented by vertex 1 contains three links: 0, 2, and 3. Similarly, the page associated with vertex 2 includes three links: 0, 1, and 4. Our primary aim is to visit all nodes (links), which is the fundamental objective of the crawler. The crawler must avoid infinite looping and avoid revisiting previously visited nodes (links).

Initially, the queue and the visited hashmap are empty of any entries, which is the ideal state of the crawler. As the crawler launches on its crawling journey, it begins by pushing the seed URL, identified as node 0, into the queue and setting it as visited after visiting it. Subsequently, once the 0 page has been visited, it is dequeued, and the following two discovered links, 1 and 2, are added to the queue. These links, 1 and 2, are similarly dequeued and recorded in the hashmap as visited links. This process persists until the queue is empty, ensuring all the links (nodes) have been visited. It is essential to observe that when the crawler encounters a loop or a link

pointing to a previously visited link, we can verify if the link has been marked as visited in the hashmap before introducing it to the queue.

In the case of a random graph, the time complexity of BFS is denoted as $O(|V| + |E|)$ where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph [?]. This time complexity depends on the graph's topology, where $O(|E|)$ can range from $O(|V|)$ (in the scenario of an acyclic graph) to $O(|V|^2)$ (if all vertices are interconnected). Consequently, the time complexity fluctuates between $O(|V| + |V|) = O(|V|)$ and $O(|V| + |V|^2) = O(|V|^2)$, depending on the specific topology of your graph. The graph assumes an acyclic structure in the existing crawler implementation since it prevents revisiting previously visited links. Consequently, the time complexity for the crawler is $O(|V|)$.

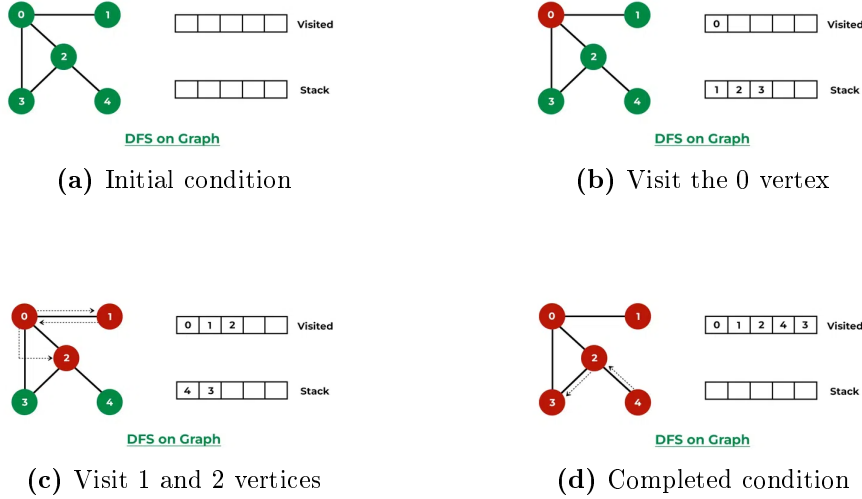


Figure 5: Illustration of various images

Depth First Search (DFS): It operates similarly to Breadth First Search (BFS). However, instead of visiting the nodes (or links) discovered first, it explores the most recently discovered nodes. Unlike BFS, DFS can be implemented using a stack. In Figure 5, there is a visual representation of how DFS operates while crawling a website.

The crawler begins with the root node, labeled 0, where the crawler first explores the seed URL node (0) and identifies the links within it (1, 2, and 3). Each of these links is added to the stack. The next node to be visited is node 1. Since it does not

lead to any further linked nodes, the crawler proceeds to the next node in the stack, node 2. Upon visiting node 2, it becomes noticeable that it contains an additional link, denoted as 4. In contrast to BFS, which would visit node 3 in this scenario, DFS prioritizes node 4 before 3 due to its use of a stack rather than a queue. Similar to the BFS example, we use a hashmap to keep tracking the visited links to avoid loops. Like BFS, DFS has a time complexity of $O(|V|)$.

3.3 Indexing

Within a search engine system, the indexer plays a crucial role in examining and structuring the content found in web pages or documents. Its primary function is to generate an index, an organized data structure that facilitates rapid and effective retrieval of pertinent information when users initiate search queries.

Indexer breaks the content into smaller components, words, and phrases, known as tokens. Afterward, it links these tokens to the respective URLs or documents from which they were created. This structured data is then stored within the index, serving as a vital reference for the search engine. It allows the search engine to swiftly locate and present relevant search results, delivering a seamless and efficient user experience.

3.3.1 Tokenization

Tokenization, within the context of indexing, entails fragmenting a textual document or a text string into smaller components known as tokens. These tokens are typically composed of words or subwords and are the fundamental building blocks for indexing and searching within a text. Tokenization represents a foundational and essential stage in natural language processing. A straightforward approach to tokenization involves dividing the text content based on spaces. For instance, the sentence "university of freiburg" would yield these tokens: "university", "of" and "freiburg". While dividing text by spaces is a straightforward and convenient solution, tokenization is a more convoluted task than it initially seems. For instance, words like "Freiburg.", "Freiburg!", and "Freiburg" should be treated as a single word, "Freiburg." Additionally, words such as "write," "writes," and "writing" essentially convey the same meaning and should not be distinguished from one another. Lastly, in cases where phrases like "Freiburg-University" are connected by a hyphen,

splitting solely by spaces would treat it as a single word, even though it comprises two distinct words.

Various approaches to tokenization exist, and in this thesis, each document undergoes a series of steps:

- Initial text segmentation by spaces.
- Conversion of all words to lowercase.
- Removal of all special characters using regular expressions.

For example, the sentence "What! Is this the 'University of Freiburg '!" will be transformed after undergoing these processes to "what", "is", "this", "the", "of" and "freiburg".

The regular expression used in Python is `[^\w\s]+`. The regular expression `[^\w\s]+` matches one or more characters that are neither word characters (letters, digits, or underscores) nor whitespace characters (spaces, tabs, line breaks). In other words, it matches any sequence of characters that contains characters other than letters, digits, underscores, spaces, tabs, or line breaks.

3.3.2 Document Unit

The term "document" frequently mentions the specific information intended for retrieval from a webpage. While, in some instances, this term encompasses the entirety of a page's content, this holds primarily for Universal crawlers like Google. However, in the case of the Preferential crawler employed in this thesis, the definition of a document unit is adjustable, depending on the nature of the website and the specific data the user aims to collect. For instance, the document unit may be viewed as a single product listing on an E-commerce website featuring product titles, prices, and descriptions. Contrarily, a news website might treat each article as an individual document. The chapter 4 will provide more comprehensive guidance on creating a template corresponding to a document.

3.3.3 Inverted index

As we discussed, crawlers are responsible for collecting data from the web and preparing it for users to search. Let us define "q" as the query string provided by the user

and "D" as the number of documents the user attempts to search for that query. If we were to implement a simple solution, we would need to iterate over each term in "q" and then search each term against every document, each of which contains $|D|$ terms. This approach results in a cubic complexity of $O(|q| * D * |D|)$, which is inefficient. Implementing an inverted index is a more efficient solution to address this issue. An inverted index or inverted file is a data structure used in information retrieval systems, particularly in search engines, to store and efficiently retrieve information about the occurrences of terms (words or phrases) within a collection of documents. It is called "inverted" because it inverts the relationship between terms and documents. In an inverted index, each unique token in the collection of documents is treated as a key, and the value associated with each token is a list of references to the documents where that token appears. This list of references allows for rapid access to all the documents containing a specific token. Some inverted indexes also include the position of the token in the document.

Creating an Inverted index requires the next steps. The first step is to collect the documents to be indexed. In the context of this thesis, the documents referred to the content inside the crawled pages. The second step is to tokenize the text, turning each document into a list of words known as tokens. The last step is to create a dictionary that maps each term with a list of the ids of the documents that occurred. The tokenized terms are called dictionaries, and the list of ids is called postings.

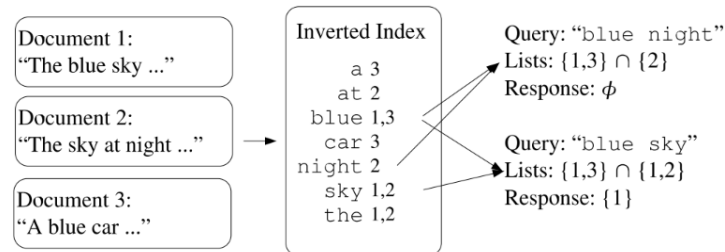


Figure 6: An illustration of an inverted index featuring three documents. All tokens are included in this example, and the sole text normalization applied is converting all tokens to lowercase. Queries that involve multiple keywords are resolved using set operations. [3]

Given that the inverted list can be implemented as a hashmap or dictionary in Python, where the average time complexity of a hashmap operation is $O(1)$, the

process of finding all the documents containing the query $|q|$ has an overall time complexity of $O(|q| * 1)$, which simplifies to $O(|q|)$. The next step involves merging the resulting documents for each term in the query q . The time complexity of merging two sorted lists is $O(n + m)$, where "n" is the length of the first list, and "m" is the length of the second list. Since there will be a resulting list for each term in the query q , the number of lists to merge is $|q|$. Consequently, the time complexity of the inverted list becomes $O(|q| * |L|)$, where $|L|$ represents the average length of the posting list.

It is worth noting that both $|q|$ and $|L|$ are typically small. The essential advantage of using the inverted list is that it makes indexing independent of the document length $|D|$, significantly improving performance.

3.4 Ranking

As discussed indexing process prepares a map that can be looked up to find relevant terms that match the search query; however, one needs to rank the returned result based on relevance. For example, a user searching "for what is Freiburg?" will be expecting a result about Freiburg and not to return all documents that contain tokens like "what" and "is". So how can we find the relevant documents? There are many algorithms for document ranking. However, in this thesis, BM25 will be adopted.

$$BM25_score = tf^* \cdot \log_2\left(\frac{N}{df}\right) \quad (3.1)$$

$$tf^* = \frac{tf \cdot (k + 1)}{k \cdot \alpha + tf} \quad (3.2)$$

$$\alpha = \frac{1 - b + b \cdot DL}{AVDL} \quad (3.3)$$

N = Total number of documents, tf = term frequency, the number of times a word occurs in a document, df = document frequency, The number of documents containing a particular word, DL = document length, $AVDL$ = average document length (measured in number of words) Standard setting for BM25: $k = 1.75$ and $b = 0.75$.

The following example dives into the details of the BM25 equation and how it impacts ranking. Table [] shows a list of documents as an example of an input to

be indexed and ranked against different search queries. We start by calculating the variables needed to find the BM25 scores for each term in a document.

Since we have three documents, the N variable will equal 3. The second step is to find document length DF for each document 1: 26, 2: 21, 3: 49. $AVDL$ will equal 32. Plugging those values into the equation, we get an inverted list as follows:

Document ID	Document content
1	The University of Freiburg, officially the Albert Ludwig University of Freiburg, is a public research university located in Freiburg im Breisgau, Baden-Württemberg, Germany.
2	Freiburg im Breisgau, usually called simply Freiburg, is an independent city in the state of Baden-Württemberg in Germany.
3	A university from Latin universitas 'a whole' is an institution of higher (or tertiary) education and research which awards academic degrees in several academic disciplines. Universities typically offer both undergraduate and postgraduate programs. In the United States, the designation is reserved for colleges that have a graduate school.

Table 1: Documents sample

Examining Table 1, we can note that the tokens that appear in all three documents, like 'the', 'of' and 'is' have scores of 0. If the term searched for is common, we should give less weight or value to the search query. For example, users often search for queries such as 'What is ...', 'who is ..' , and 'Where is ...'. Those queries contain common words that are not informative in documents like 'What', 'Who', 'Where' and 'is', the term coming after those sentences should be more valuable and have more weight. Unique words like 'albert' and 'ludwig' have high scores as they only occur in one document. Words like 'freiburg' and 'university' have different scores for each document depending on the word's appearance relative to the document's length.

User search for 'university of freiburg' will return the next result: (1, 2.14), (2, 0.97), (3, 0.46). The first document with id 1 has the highest score as it contains both words. This is the correct result, as the first document talks about the university of freiburg. The second document 2, which talks about the city Freiburg, is higher than the third because freiburg is mentioned twice in the same document, and the content is shorter. The next section will exemplify how can one refine the ranking of documents.

Token	(Doc. ID, BM25 Score)
the	[(1, 0.0), (2, 0.0), (3, 0.0)]
university	[(1, 1.071), (3, 0.466)]
of	[(1, 0.0), (2, 0.0), (3, 0.0)]
freiburg	[(1, 1.071), (2, 0.975)]
officially	[(1, 1.740)]
albert	[(1, 1.740)]
ludwig	[(1, 1.740)]
is	[(1, 0.0), (2, 0.0), (3, 0.0)]
a	[(1, 0.642), (3, 0.885)]
public	[(1, 1.740)]

Table 2: The first 10 tokens of the result inverted index and the scores of the documents.

3.4.1 Ranking Refinements

Some methods can be implemented to boost the document’s ranking. The first step is to focus on tokenization of the documents.

In the previous example, we can note that the university term has appeared in document 3 only once; however, the term universities appeared twice. In the ranking, the relation between the two words is not achieved because the inverted index will include university and universities separately. This will reduce the score of document 3 when the user searches for university, although both terms are associated and linked and should be accumulated. Stemming and lemmatization, Both stemming and lemmatization aim to simplify inflectional forms and occasionally derivationally related forms of a word, bringing them down to a shared foundational form. As an example:

am, are, is \Rightarrow be
car, cars, car’s, cars’ \Rightarrow car

Stemming typically involves employing a simple rule-based approach to truncate word endings, aiming to achieve accurate results in most cases. This approach often involves eliminating derivational affixes. In contrast, lemmatization follows a more meticulous process that involves utilizing vocabulary and morphological analysis of words. The primary objective is to exclusively remove inflectional endings and to restore words to their fundamental or dictionary forms, which are referred to as

lemmas [5].

We observed that small or frequently occurring tokens such as 'the' and 'of' possess scores of 0, contributing no significance to the overall outcome of the query. Eliminating these terms via stop words can lead to the exclusion of certain frequently used words from the indexing process. A stop words list is a list that holds words that can be excluded from the indexing process. The selection of stop words is language-dependent; each language has its own set of prevalent words. For instance, in English, the subsequent list provides an example of such stop words that could be omitted from the indexing procedure:

a an and are as at be by for from has he in is it its of on that the to was were will with

3.4.2 Fuzzy search

As previously explained, the generated inverted index will contain a list of the tokens found on each document, and to find the most relevant document to the user query would be simply to split the query into tokens and search for each token and find its exact matching in the inverted index, then rank the results based on the BM25 scores. Assuming that users will not make any misspelling errors is a hard assumption, especially for some English words; there are differences between British and American spelling, for example, 'color' and 'colour'. Both have the exact same meaning with a different spelling. It would be bad not to return any result if the user chose one word over the other. Other scenarios can also be that the user is not sure about the spelling. For example, 'Freiburg' can be written as 'Frieburg'.

Fuzzy search is a technique used in natural language processing (NLP) and information retrieval to find approximate matches for a given query or search term, even when the exact spelling or wording might not be present in the target text. This is particularly useful when dealing with typos, misspellings, variations in phrasing, or other types of small deviations from the original text.

Fuzzy search algorithms typically involve techniques like Levenshtein distance (edit distance), which calculates the minimum number of single-character edits (insertions, deletions, substitutions) required to change one string into another. Other techniques include using phonetic algorithms to find similar-sounding words, or tokenization and comparison of word n-grams to identify overlapping substrings.

Considering two character strings, s_1 and s_2 , the edit distance that separates them represents the minimal count of edit operations needed to transform s_1 into s_2 . The typical edit operations permitted for this purpose encompass: (i) the insertion of a character into a string, (ii) the deletion of a character from a string, and (iii) the replacement of a character within a string by another character. In the context of these operations, the term "Levenshtein distance" is sometimes used interchangeably with edit distance. To illustrate, the edit distance between "cat" and "dog" is 3. It's worth noting that the concept of edit distance can be extended to encompass varying weights assigned to different types of edit operations. For instance, assigning a greater weight to the replacement of the character "s" with "p" compared to its replacement with "a" (with the latter being physically closer on the keyboard) can be explored. This weight assignment strategy, dependent on the probability of letter substitutions, proves highly effective in practical scenarios. Nonetheless, the subsequent discussion primarily concentrates on scenarios where all edit operations bear identical weights [5].

Fuzzy search can be used with q-gram to find how similar words are. For a string x , and an integer $q \in \mathbb{N}$, the multiset of q-grams, denoted by $Q_q(x)$, consists of all substrings of length q . $Q_3(\text{"freiburg"}) = \{\text{"fre"}, \text{"rei"}, \text{"eib"}, \text{"ibu"}, \text{"bur"}, \text{"urg"}\}$. We define it as a multiset because the same q-gram may occur multiple times and we want to know when it does

$$Q_3(\text{"ababa"}) = \{\text{"aba"}, \text{"bab"}, \text{"aba"}\}$$

The number of q-grams of a string x is:

$$|Q_q(x)| = |x| - q + 1$$

Similar words have many q-grams in common, that's why it can be used to find similar words. Lemma: for strings x and y : $|Q_q(x) \setminus Q_q(y)| \leq q \cdot \text{ED}(x, y)$. Understand: $A \setminus B$ denotes the set difference, that is, the elements of A without the elements from B . If B is very similar to A , then $A \setminus B$ is small [13]. Example: $x = \text{"freiburgerin"}$, $y = \text{"breifurgerin"}$, $\text{ED}(x, y) = 2$. $Q_2(x) = \{\text{"fr"}, \text{"re"}, \text{"ei"}, \text{"ib"}, \text{"bu"}, \text{"ur"}, \text{"rg"}, \text{"ge"}, \text{"er"}, \text{"ri"}, \text{"in"}\}$. $Q_2(y) = \{\text{"br"}, \text{"re"}, \text{"ei"}, \text{"if"}, \text{"fu"}, \text{"ur"}, \text{"rg"}, \text{"ge"}, \text{"er"}, \text{"ri"}, \text{"in"}\}$. $|Q_2(x) \setminus Q_2(y)| = 3$.

To implement fuzzy search with q-gram, one can use Q-gram index. For each q-gram of a string from D , store an inverted list of all strings from D containing it, sorted lexicographically

\$fr : frankfurt, freiburg, freetown, fresno, ...

More information about the implementation will be discussed in the approach section.

4 Approach

This chapter introduces the distinctive concept of a configurable search engine and outlines the comprehensive software architecture. It thoroughly explores and showcases all the required components for constructing the search engine, along with discussing implementation specifics. This encompasses the pivotal models and classes employed for the components and algorithms. Furthermore, the chapter delves into the user interface, clarifying how it enhances user experience and facilitates configuring the search engine.

4.1 Software Architecture

Figure 4 provides an overview of the software architecture employed by the search engine. Microservices architecture was used to make scalability easier and also to split the responsibilities of each component. Docker is used to enforce this architectural pattern where the Ubuntu:18.04 image is used for each component. Below is a compilation of the utilized technology stack:

- **Frontend (Angular & PrimeNG):** Positioned closest to the user, this component encompasses all pages and views. Angular is leveraged in conjunction with the contemporary CSS library, PrimeNG. To communicate with the backend, it employs the REST API.
- **Backend (Django):** Serving as the core intelligence of the search engine, the backend houses both the crawler and indexer modules. It facilitates interaction with the Head node to initiate crawling based on user-defined configurations. Moreover, it establishes a connection with PostgreSQL for the storage of crawler and indexer configurations, along with job-related information.
- **Head Node (PBS):** Operating as the central hub, this node orchestrates job management and determines the allocation of tasks to Crawler nodes, which

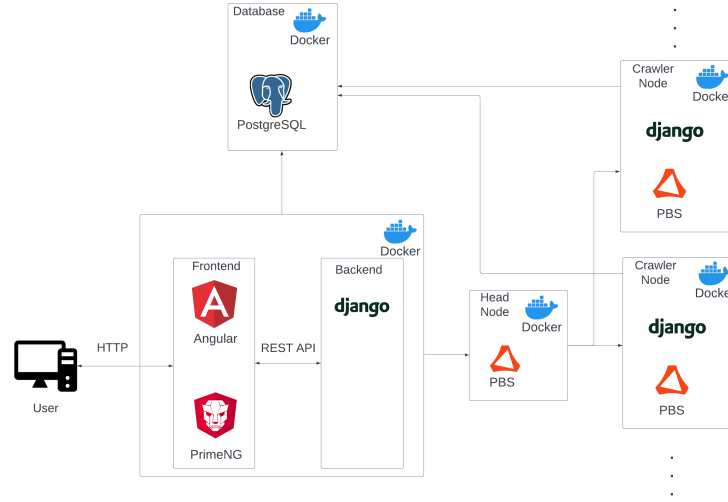


Figure 7: High-level view of the software architecture.

are responsible for traversing the specified websites.

- **Crawler Node (PBS):** These instances are designated to execute the crawling process and store the resulting data in the PostgreSQL database.

The application setup initiates with a minimal requirement of four microservices to operate the entire search engine. One Docker container containing both Angular and Backend logic must establish connections with two other containers: the Database and PBS Head Node. The PBS Head Node, in turn, should connect to one or more Crawler Node containers responsible for executing the crawling task. The Crawler Node will perform the crawling job and save the results to the shared Database.

The workflow begins with a user-friendly interface presented by Angular and PrimeNG, encompassing all the configurations and tools enabling users to crawl quickly and index various websites. Users can modify configurations and submit a crawling job to the Head Node. The Head Node, in response, identifies an available Crawler Node to execute the task. It's worth noting that the PBS cluster can be bypassed, and the crawling process can be run locally on a localhost server. Users can monitor the progress of the running job from the browser. Once crawling is completed and the user is happy with the result, the user can start indexing. The indexing job also does not support a distributed architecture and will be executed locally and not on the PBS cluster.

4.2 Crawler Implementation

PBS Crawler Node runs the crawling job or can also run locally. The job supports multithreading. As illustrated by the pseudo-code shown in Algorithm [1], the crawler starts by loading the configuration submitted by the user from the Database. More details about the configuration are in the user interface section. Based on the configuration of a thread pool, the number of threads is read by the configuration. The thread pool contains all the threads crawling the site, where each thread contains a queue of URLs that it crawls from. The thread pool makes sure that if one thread has no URLs anymore, it can ask other threads to help. Algorithm [2] explains how this sharing URLs mechanism works.

Algorithm 1 Start Crawling

```
1: load_crawler_configurations()
2: thread ← create_threads_pool()
3: urls_queue ← get_thread_urls_queue(thread)
4: seed_url ← get_seed_url()
5: add_url_to_queue(urls_queue, seed_url)
6: robot_file ← get_robot_file_content()
7: while urls_queue not empty or all threads not done do
8:   if urls_queue is empty then
9:     urls_queue ← get_thread_urls_queue(thread)
10:  else
11:    current_url ← urls_queue_next_url()
12:    filter_unwanted_urls(current_url)
13:    request_page(current_url)
14:    execute_automated_actions()
15:    find_next_urls_and_add_them_to_urls_queue()
16:    docs ← find_and_download_targeted_documents()
17:    filter_unwanted_documents(docs)
18:  end if
19: end while
```

A seed URL is added to the current thread queue. The seed URL represents the starting point for the crawling, and the user configuration defines it. Using the seed

url, the robots.txt content is downloaded once and can be reused for the rest of the crawling process.

Each thread goes into an infinite loop that will continue to run either if the URLs queue still contains URLs to be fetched or at least one thread is still running. This guarantees that although one thread is running, it can be that that thread contains a lot of URLs that need help with crawling, and the free threads can share the load with it. If the thread queue is empty, it will ask the pool to find the next URLs to fetch. Otherwise, the first URL in the queue will be fetched, and a request using Selenium will be made. Afterwards, automated actions such as scrolling down, waiting and clicking defined by the user are executed. Those actions give the power to control the browser by the user to mimic real agent behaviour. The action chain will be discussed more in the User Interface section.

After the page is rendered and the automated actions are executed, the next step is to collect the next URLs and add them to the URLs queue. The last step is to parse the documents needed from the page and filter the duplicated documents.

4.2.1 Links Data Structure

The website's page navigation algorithm can be likened to a Level Order Traversal. The tree structure is established in the following manner: the seed URL acts as the tree's root node, representing level 0. After you explore the root page's content and gather its URLs, these URLs are assigned to the next level, level 1. Each page within level 1 is then visited, its contained URLs are collected, and these newly collected URLs are assigned to level 2. This process continues as you move deeper into the website until a maximum depth is reached, which is pre-defined by the user. This algorithm offers an advantage in that it makes it straightforward to prioritize pages based on their respective levels. For instance, in some scenarios, pages closer to the initial seed URL may receive higher priority, potentially yielding better outcomes. In other cases, deeper pages within the structure may hold more significance than those closer to the seed URL. Choosing the proper algorithm is possible in the UI.

4.2.2 Practical Challenges

- **Avoiding Loops:** Looping is when a web crawler repeatedly visits and requests the same web pages or URLs in a never-ending cycle, often resulting in

excessive traffic to the same content. This is problematic as it wastes resources can also be inefficient, and can prevent the crawler from continuing. The first method to prevent looping is to record all the URLs visited and crawled. Before making a new request, we check if the URL is in this list. If it is, skip crawling it again to prevent loops. The second method is to use URL normalization. Normalize URLs by removing unnecessary components such as query parameters, fragments, or trailing slashes. This helps ensure that URLs with different representations (e.g., with and without a trailing slash) are treated as the same URL.

- **Duplicated Content:** While the same web crawler avoids revisiting identical URLs to prevent content duplication, it's important to note that identical content may exist in different URLs paths within the same website. For instance, a men's shoe might be accessible via various links like `"/winter/shoes/"`, `"/men/shoes/"`, or `"/sales/shoes/"`. Relying solely on the URL as a unique identifier to prevent content duplication is not foolproof. A more effective approach involves comparing the content itself with the database after parsing. Instead of a straightforward content check against the database, which can pose performance challenges, we employ a more efficient method. We generate a unique hash code using the SHA-1 hashing algorithm based on the content string intended for storage. This hash code is then stored in the database. Before saving any new content, we can verify if the hash code already exists in the database. This method ensures content uniqueness, even when it appears under different URLs on the same site, without the computational overhead of directly comparing lengthy content strings in the database.
- **Dynamic Content:** Crawling dynamic websites presents a distinct set of challenges compared to static websites. Dynamic sites generate content on the client side through technologies like JavaScript, adding complexity to the task of accessing and extracting data. A primary concern lies in uncovering concealed content that necessitates user interaction. For instance, certain websites hide lengthy content portions, revealing them only upon clicking a "read more" button. Additionally, most websites implement lazy loading, fetching content on-demand via AJAX requests. To address these challenges, Selenium establishes a genuine session and fully renders the webpage. This approach allows

for emulating user interactions using action chains, which simulate actions such as waiting, scrolling, and clicking. More details regarding this can be found in the User Interface section.

- **Termination Conditions:** Crawlers can be brought to a halt by establishing specific criteria to ensure termination. The initial criterion involves defining a maximum depth, which restricts the number of page transitions to a single level. Additionally, monitoring and restricting the total count of visited pages and collected documents is possible. Another method is to employ a wall time measurement to monitor the crawler's runtime duration and trigger an abort if the crawler exceeds the expected time frame.
- **Avoiding DOS:** Increasing the number of requests and expanding the crawler's capacity by adding more threads or nodes may seem enticing to boost performance. However, this approach carries a significant risk of overwhelming the targeted servers, potentially resulting in Distributed Denial of Service (DDoS) or Denial of Service (DoS) attacks. Servers can perceive this surge in requests as an attack, which could lead to the crawler being blacklisted and subsequently banned. To mitigate this risk, it's crucial to introduce a waiting period between each request made by the same crawler. Additionally, when using Selenium, a deliberate delay of at least one second or more is already integrated to allow for the complete rendering of web pages. Nevertheless, these precautions alone may not prevent users from adding more nodes and executing DDoS attacks. Consequently, it is strongly advisable to exercise cautious management by monitoring and regulating the number of threads and nodes. This approach demonstrates respect for the targeted servers and helps prevent overloading them.

4.3 Indexer Implementation

The indexing phase follows the crawling step, as it necessitates the downloading of documents. Unlike crawling, which can be computationally intensive, indexing is relatively lightweight. Hence, it is carried out on the localhost with multithreading support. Algorithm [2] clarifies the indexing process. Firstly, the algorithm loads the user-defined indexing configurations. You can find more details about these

configurations in the User Interface Design section. Subsequently, an empty inverted list is initialized. An inverted list is a data structure used to associate words or terms with the documents in which they appear. In Python, it can be implemented as a dictionary (Dict[str, list[int]]), where each word serves as a key, and the corresponding value is a list of document IDs containing that word.

Algorithm 2 Create Inverted List

Require: *documents* not empty

```

1: config ← load_indexer_configurations()
2: inverted_list ← init_inverted_list()
3: threshold ← get_small_words_threshold(config)
4: stop_list ← stop_words_list(config)
5: for doc in documents do
6:   doc_length ← 0
7:   words ← tokenize(doc)
8:   for word in words do
9:     doc_length ← 0
10:    if word > threshold and word not in stop_list then
11:      add_word_and_doc_id_to_inverted_list(word, doc.id)
12:      doc_length += 1
13:    end if
14:  end for
15:  add_to_doc_lengths_list(doc_length)
16: end for
17: calculate_bm25_score(inverted_list)
18: cache(inverted_list)

```

The user specifies three variables: "threshold" and "stop_list," which are retrieved from the database. The "threshold" represents the minimum word length required for tokenization from a document. The "stop_list" is a predefined list of terms that should be omitted from the indexing process. The algorithm then iterates through all the documents, performing the following steps for each one: Initializes the document length as a counter, initially set to zero. Tokenizes the document to obtain a list of words. Iterates through the word list, checking each word's length against the

threshold and verifying if it is not in the `stop_list`. If these conditions are met, the word is added to the inverted list, and the document length counter is incremented by one.

Once the inverted list is constructed, we calculate the MB25 score based on equation 3.4. Afterwards, it is saved into a cache for future retrieval and use.

4.4 User Interface Design

The core focus of this thesis extends beyond merely building and designing a search engine. It also encompasses the vital objective of enabling users to effortlessly configure and utilize it. In the forthcoming section, we will delve into the User Interface design, workflow, and the user-facing configurations.

The user begins their journey on the homepage, where they can access documentation explaining how to utilize the application. The application's workflow commences with the creation of templates, which serve as blueprints for specifying the document fields to be extracted from web pages. It is a prerequisite to establish a unique template for each page, although the same template can also be applied across different websites. These templates consist of components referred to as "inspectors," each comprising the following attributes:

- **Name:** This denotes the inspector's identifier, such as "Price" or "Title."
- **Selector:** It encompasses the XPath expression pinpointing the chosen element, for instance, `//*[contains(@class, 'product-title')]`.
- **Type:** This can assume values like "Text," "Link," or "Image," signifying the nature of the content to be extracted.
- **Variable Name (*Optional*):** This is an optional shorthand representation of the selector, facilitating its use during the indexing process to enhance search results (Ranking).
- **Clean-up Expression List (*Optional*):** Here, you can specify a regular expression utilized to refine the extracted value from the inspector. This proves beneficial in eliminating unwanted noise. A use case for this can be to remove the currency when extracting a price.

- **Attribute (*Optional*):** This field allows you to specify an HTML element attribute, such as "src," "name," or "href," as an optional parameter.

The following phase is contingent upon the specific characteristics of the website being targeted, referred to as the "Actions Chain" tab. An "Actions Chain" constitutes an array of sequenced actions that replicate user interactions. This functionality proves valuable for tasks such as acknowledging cookies, scrolling to load additional content, or patiently waiting for the website to finish rendering in cases where the process may exceed the expected duration.

Templates

+ Create a template

Name	Created at	Actions	
Flaconi	4 months ago	Edit	Delete

Inspectors

⚡ Actions

Inspectors

+ Create an inspector

Name ↑↓	Type ↑↓	Selector ↑↓	Created at ↑↓	Actions
Title (x)	text	//*[contains(@class, 'BrandName')]	4 months ago	<div>EditDelete</div>
Price (p)	text	//*[contains(@class, 'PriceValue')]	4 months ago	<div>EditDelete</div>
Image (iiii)	image	//*[contains(@class, 'ProductPreviewSliderstyle__CarouselWrapper-sc')]/img	4 months ago	<div>EditDelete</div>

>

douglas

3 months ago

EditDelete

>

Facebook

3 months ago

EditDelete

>

Douglas flat list

3 months ago

EditDelete

Figure 8: An overview of the Templates table containing inspectors list projecting the targeted feilds in a document

Once the Template is created, the subsequent step is to access the Crawlers page and initiate the creation of a new Crawler. A Crawler comprises various essential configurations, including:

Once you've set up the crawler with the appropriate configurations tailored to the specific website you're targeting, the next step is to create a job referred to as a "runner." Multiple runners can be associated with each crawler, allowing them to run on different days or machines. It's important to note that each runner can utilize multi-threading based on the crawler's configurations and employ distinct crawler settings. This approach provides an effective means to assess the crawler's performance until

Attribute	Description
Name	A user-defined identifier for the crawler
Template	The template utilized by the crawler to recognize HTML elements for crawling and storage
Max pages	The upper limit for the number of pages to be visited during the crawling process
Max collected docs	The maximum number of documents to be collected
Max pages	The upper limit for the number of pages to be visited during the crawling process
Seed URL	The initial URL from which the crawler will commence the crawling process. Ensure that the URL includes the protocol (e.g., https://)
Robots file URL	The URL where the robots.txt file can be located
Threads	Specifies the number of threads to be employed in the crawling process.
Links Scope (Pagination)	Defines the specific divs on which the crawler should concentrate. Typically, you'd want the crawler to avoid collecting links from areas like headers and footers. Example: <code>//*[contains(@class, 'product-overview')]</code>
Excluded URLs	Identifies URLs that the crawler must steer clear of and refrain from visiting.
Timeout in minutes	Sets the duration for which the crawler should continue crawling.
Retry	Determines how many retry attempts should be made if the crawler encounters difficulties while crawling a page.
Max depth	Establishes the depth to which the crawler should navigate through pages.
Sleep in ms	Specifies the number of milliseconds the crawler should wait before making the next request.
Show browser	This option, when enabled, displays the browser interface during crawling, which is beneficial for debugging purposes.

Table 3: Crawler configurations options

the desired outcome is achieved. Every runner instance necessitates the presence of the crawler and a designated machine IP where it will execute. The chosen machine must be registered within the PBS Head Node and must be online. By default, "localhost" is set as the value, which can be employed for testing purposes and for circumventing the PBS cluster.

It's crucial to keep a close eye on the crawler runner to monitor its performance and configuration effectiveness before finalizing it. This proactive approach helps save time and guarantees that the crawler is collecting the accurate data. The "runners" table provides a straightforward and informative progress overview through four primary status indicators. The initial status is "New," representing the run-

ID	Status	Crawler	Progress	Actions
Uni ranking Local #132	New	Uni ranking #9	Started at: a month ago / Completed at: a month ago #Documents: 2389 Current URL: timeshighereducation.com/world-university-rankings...	[Menu] [Download CSV]
Uni ranking #131	New	Uni ranking #9	Started at: a month ago / Completed at: a month ago	[Menu] [Download CSV]
docker sim 2 #130	New	sdf #8	Started at: a month ago / Completed at: -	[Menu] [Download CSV]
Docker #129	New	sdf #8	Started at: a month ago / Completed at: a month ago	[Menu] [Download CSV]
Hacker News #116	New	Hacker News #6	Started at: a month ago / Completed at: a month ago #Documents: 435 Current URL: news.ycombinator.com/item?id=36834675	[Menu] [Download CSV]
Random walk #111	New	Random walk #5	Started at: a month ago / Completed at: a month ago #Documents: 798 Current URL: services.amazon.de/programme/prime-durchverkauf/...	[Menu] [Download CSV]

Figure 9: An overview of the runners table, showing the runners status and progress.

ner's initialization before the crawling process begins. "Running" signifies that the crawling process is currently underway. "Exit" indicates a status change that occurs when an error occurs, leading to the termination of the crawling process. Finally, "Completed" marks the last status, indicating that the runner has finished its task and exited. During the crawling process, various statistics about the runner are gathered, including information such as the total number of visited pages, the average number of documents discovered per page, and the various HTTP status codes encountered. One can retrieve the documents collected by the runner by selecting the "Download CSV" option from the actions dropdown menu.

Once the runner has finished its task and the user is satisfied with the results, the collected documents can be indexed and prepared for future searches. To access this feature, you can navigate to the Indexers page, where you'll find a table displaying the current indexers and their respective statuses.

As previously mentioned, indexers are responsible for handling documents, each of which contains a string field suitable for indexing. In this context, inspectors come into play. These inspectors are responsible for mapping the fields extracted from the document, such as Name, Title, Price, and Image. Users have the option to select which inspectors or fields to index from a dropdown menu, with the choice limited

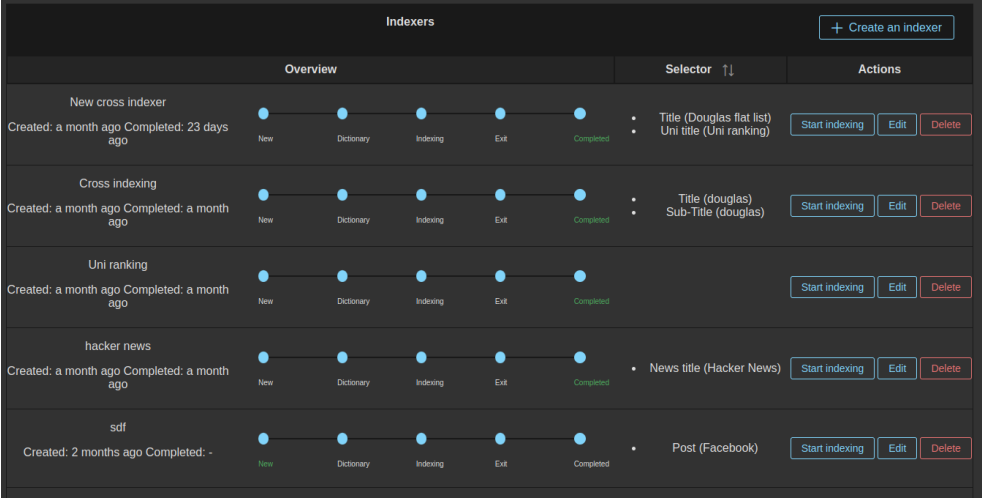


Figure 10: An overview of the indwxwea table, showing the their status and progress.

to only text fields, images and links are excluded from this selection.

For instance, if you are gathering product information, you can opt to index fields like the Title and description. Additionally, cross-indexing is also possible, allowing the indexing of documents from two different crawlers. This capability proves valuable in scenarios where two distinct crawlers are fetching different products from the same website.

In Table [4], you can find the configurations related to the indexer. While some of these configurations are already explained within the table, others may benefit from further clarification. The 'Weight words list' refers to a list of words along with their associated weights. If a term containing one of these words is present in a query, its score will be augmented and contribute to the overall query score.

The 'Boosting formula' is a feature that can be employed in conjunction with inspector variables to influence the ranking process. For example, if you want to rank products based not only on text relevance but also consider factors like reviews or prices (which are numeric values rather than text), you can utilize the 'Boosting formula.' To do this, you can assign a variable name to an inspector, such as 'review.' Then, in the 'Boosting formula' field, you can input an expression like 'log(review),' which will convert the numeric value in the inspector field 'review' into a numerical score. This score is then incorporated into the ranking formula, contributing to the final ranking score.

Attribute	Description
Name	A user-defined identifier for the indexer
Inspectors	Checklist of all the available inspectors used by the crawlers.
B parameter	B parameter for the BM25 formula.
K parameter	K parameter for the BM25 formula.
Stop words list	List of words that should be excluded during indexing process.
Small Words Threshold	The threshold of which the word can be considered small and will be skipped, by default 0 meaning empty spaces will be skipped from indexing process.
Weight words list	Boost some words and punish other by giving them weight, e.g. 'Freiburg=5' will add more 5 points to the score when Freiburg word is found.
Boosting formula	This formula result will be added to the final score, it uses inspectors variable.
Dictionary file name	The name of the dictionary file name, this file help the suggestions list by using synonymes.
Use Synonyms	Enable using synonyms in the suggestions list, for example, typing 'USA' will result on 'United States of America'.
Q-Gram	How many letters should the word be split to, for example q=3,word=freiburg, will result in the next grams 'fre', 'rei', 'eib', 'ibu', 'bur', 'urg'.

Table 4: Indexer configurations options

In the end, all the indexed documents can be easily located on the search page, which consists of three primary components. First, there's the search bar, which leverages the suggestions list dictionary configured during the indexing phase. The second component is a dropdown menu containing all the cached indexers that have previously been indexed. The last component is the result table, which boasts a dynamic layout depending on the inspectors used for each document.

For example, a product with inspectors like Image, Title, Name, and Price will have four columns displaying the relevant data. The data accommodates various inspector types, including text, links, and images. It's worth noting that, at present, only the top 25 results are displayed, and this limit is not customizable.

dior

New cross indexer



Image		Price	Title
	<div>Christian Dior</div> <div>diorite</div> <div>Christian Dior</div> <div>diorama</div> <div>Diori Hamani</div>	75.99	DIOR
	Miss Dior Eau de Parfum – Limitierte Edition zum Valentinstag Geschenkset	132.00	DIOR

Figure 11: High-level view of the software archititcture.

5 Evaluation

In this chapter, our objective is to assess and examine the existing search engine implementation. The evaluation process is structured into three primary segments: Crawler, Indexer, and User Experience.

5.1 Testing Environment

The evaluation procedure is significantly influenced by the specific computing device executing the tests. Displaying information about the testing machine utilized can provide enhanced clarity and facilitate meaningful comparisons.

Operating System	Ubuntu 22.04.3 LTS
CPU	Intel(R) Core(TM) i7-10510U @ 1.80GHz; 4 cores; 8 threads
RAM	32GB
Machine	Lenovo ThinkPad P15s Gen 1

Table 5: Local machine setup

5.2 Crawler

To evaluate the web crawler, the following criteria can be employed for measurement:

Coverage: This metric measures the proportion of relevant web pages that the crawler can locate and fetch from the internet.

Scalability: It evaluates the crawler’s facility for efficiently scaling up to add more computing power to crawl more content.

Versatility: Can the crawler be applied to explore diverse types of content from various websites, encompassing text, multimedia, and links?

Robustness: The crawler’s ability to adeptly manage challenging scenarios and errors.

Politeness: the extent to which the crawler respects the rules and policies of the web servers and avoids overloading them with requests and forbidden links.

5.2.1 Datasets

Evaluating a web crawler necessitates the utilization of a static website as a reliable reference point. The crawler-test¹ website serves as an excellent choice for this purpose due to its diverse range of content and links, containing a wide range of scenarios that a crawler might encounter. This website effectively employs robots.txt to provide guidance to the crawler, allowing for an assessment of its politeness. Moreover, it includes a section containing links that yield various HTTP request status codes, such as 4xx and 5xx, which proves valuable for ensuring the crawler’s robustness. Additionally, it incorporates multiple instances of page redirection, including scenarios like infinite redirection, which serves the dual purpose of evaluating the crawler’s ability to avoid traps and enhancing its overall resilience.

To enhance the coverage and versatility of our crawler testing, we are considering two additional websites encompassing a broader range of use cases, ensuring that the crawler can effectively handle various HTML structures and more generic scenarios. The first use case involves extracting product information from an E-commerce platform like Douglas². This website offers over 160,000 diverse products, making it an ideal candidate for testing different content types, including images. We will categorize our testing into three distinct datasets: small (with 100 products), medium (with 1,000 products), and large (with 10,000 products) to evaluate the crawler’s performance under different data sizes thoroughly. The second website, Times Higher Education³, specializes in annually ranking universities. This unique characteristic allows us to evaluate content changes occurring on a yearly basis, setting it apart from other websites that undergo daily alterations, which can prove challenging for accurate evaluation.

¹<https://crawler-test.com/>

²<https://www.douglas.de/>

³<https://www.timeshighereducation.com/world-university-rankings/2023/world-ranking>

5.2.2 Experiments

The crawler-test website is selected to assess and experiment with various crawler configurations. This choice is attributed to its stable, non-changing nature, which facilitates straightforward and meaningful comparisons of results across different configurations.

The first testing case is the coverage. It is vital to ensure the crawler can find the links inside the page, navigate to them and crawl them.

Table 6 displays the testing configurations used in the experiment. The Seed URL serves as the initial point from which the crawler begins its procedure, set to the root path of the crawler-test. The "Allow Multi Elements" checkbox is disabled (set to False) because the objective is not to gather a list of documents; each page contains a single text field. The Max Pages parameter is configured to a limit of 500, ensuring that the crawler does not exceed this number of pages. This figure can be adjusted based on the website's size to be crawled; for instance, smaller websites with around 50 pages may require a lower limit. The number of threads is set to the default value of 1. We have set the Max Depth to 1 to enable easier coverage testing. This choice allows for easier comparison between the number of visited pages and the number of URLs discovered in the site's root path, which, upon simple page inspection, contains 415 links. Since the expected maximum number of pages is 415, the Max Docs parameter can be constrained to 500. The inspectors are set to target the content of each page; thus, one inspector is only needed. No automated actions, such as scrolling or waiting, are necessary for this use case; therefore, they can be left. Any properties not explicitly mentioned can be left at their default settings.

After creating a runner that runs, starts the crawling process and is completed, the result of the crawler should be similar to the one shown in Table 7. Looking at the Links row, we can note that the crawler found 406 out of the expected 415 links. This is normal as the crawler will normalize all the found links, including skipping the link fragments, duplicated links and any broken links. 402 pages out of 406 found links are crawled correctly. The other four pages are categorized as Cross Site links, meaning they do not belong to the Seed URL hostname crawler-test.com. This is important to evaluate to ensure that the crawler stays focused, does not jump to sites out of the intended scope, and does not spend valuable resources. Already Visited links is a counter that checks how many times the crawler found a link that

Seed URL	https://crawler-test.com/
Allow Multi Elements	False
Max Pages	500
Threads	1
Max Depth	1
Pagination	None
Actions	None
Inspectors	//*[contains(@class, 'large-12 columns')]
Max Docs	500

Table 6: Crawler configuration

has already been visited and skipped, in this case, 0. When no multithreading is used, and the Max Depth is only set to 1, the Already Visited is expected to be 0 because the duplicated links will be already excluded in the normalizing process before starting to crawl.

Links	Collected: 406/415 . Visited Correctly: 402. Cross Site: 4 . Already Visited: 0
Time	Tot. Spent: 671.19 s. Avg. Processing: 1.68 s. Avg. Page Rendering: 0.697 s
Status Codes	101: 1 102: 1 200: 321 201: 1 202: 1 203: 1 206: 3 207: 1 226: 1 300: 1 305: 2 306: 1 400: 1 401: 1 402: 1 403: 2 404: 11 405: 1 406: 1 407: 1 408: 1 409: 1 410: 1 411: 1 412: 1 413: 1 414: 1 415: 1 416: 1 417: 1 418: 1 419: 1 420: 1 421: 1 422: 1 423: 1 424: 1 426: 1 428: 1 429: 1 431: 1 440: 1 444: 1 449: 1 450: 1 451: 1 494: 1 495: 1 496: 1 497: 1 498: 1 499: 1 500: 2 501: 1 502: 1 503: 1 504: 1 505: 1 506: 1 507: 1 508: 1 509: 1 510: 1 511: 1 520: 1 598: 1 599: 1
Docs & Content	Tot. Docs Found: 255. Duplicated Content:24. Avg. Docs Per Page:1. Average page size: 0.000346
Robots.txt Exists	True
Tot. Errors	6

Table 7: Crawler configuration

Evaluating the performance of a web crawler is challenging as it depends on different aspects, such as the page’s size and how fast the page loads. Moreover, if the site uses pagination, this can add extra waiting time to render the rest of the content. However, some valuable matrices can be helpful to give a good insight like those shown in the Time row. The total time spent to crawl 406 pages took approx-

imately 11 minutes. To give a better perspective, the enterprise solution Parsehub⁴ the free tier without IP Rotation, can crawl 200 pages in 40 minutes, and the Standard expensive tier with IP Rotation that costs \$189 can crawl 200 pages in 10 minutes. This means crawling the 406 pages inside the crawler-test will take 20 minutes. Comparing the crawler with the Standard tier, it is faster by 2X. Note that in this evaluation, the performance can be increased by using more threads or nodes to distribute the loads, which we will evaluate.

```
1 <body>
2   <h1 id="h1"></h1>
3   <p id="par"></p>
4   <script>
5     var title = "Some random text with no result in Google so
6       we can see if the page ranks for this text";
7     var content = "Same idea here for the content of the
8       paragraph. Improve him believe opinion offered met and
9       end cheered forbade. Friendly as stronger speedily by
        recurred. Son interest wandered sir addition end say.
        Manners beloved affixed picture men ask.";
7     document.getElementById("h1").innerHTML = title;
8     document.getElementById("par").innerHTML = content;
        </script>
9 </body>
```

Listing 5.1: The dynamically-inserted-text link content before rendering

Improving the average page rendering time (0.679) is achievable by avoiding using rendering engines like Selenium. Processing a simple HTTP request is often quicker than rendering the entire page. However, it's essential to note that the rendering step is crucial in handling dynamic content. For instance, consider the dynamically inserted text, indicated by the link⁵. This link is a straightforward example to illustrate the significance of the rendering process for web crawlers. Using a simple `wget` command in Linux, you can download the content depicted in 5.1. This content reveals that the two HTML tags, `h1` and `p`, lack the inner text that the crawler can collect as a document. Although there's a `script` tag designed to replace the `innerHTML` for each tag with text, without a rendering engine, the JavaScript logic

⁴<https://www.parsehub.com/pricing>

⁵<https://crawler-test.com/javascript/dynamically-inserted-text>

remains unexecuted. Consequently, gathering the inner text of these tags becomes an impossibility. On the other hand, 5.2 shows the same page content after rendering where both tags are updated, and both contain the right content that the crawler can see and download. While this is a simplified example, it's easy to visualise more complicated websites employing advanced JavaScript frameworks and libraries for client-side rendering. This complexity increases the rendering time due to the execution of all JavaScript logic and the subsequent updating of the HTML DOM. Given that the crawler's objective is to locate and collect all HTML content, striking a balance between efficiency and comprehensiveness is a challenge that must be addressed.

```
1 <body>
2   <h1 id="h1">
3     "Some random text with no result in Google so we can see
4       if the page ranks for this text"
5   </h1>
6   <p id="par">
7     "Same idea here for the content of the paragraph. Improve
8       him believe opinion offered met and end cheered
9       forbade. Friendly as stronger speedily by recurred.
10      Son interest wandered sir addition end say. Manners
11      beloved affixed picture men ask."
12   </p>
13 </body>
```

Listing 5.2: The dynamically-inserted-text link content after rendering

The Status Codes metric reveals the variety of distinct status codes encountered

while crawling. Given that this website serves as a testing ground for various scenarios, it naturally exhibits a range of status codes. Evaluating the crawler's resilience across these diverse cases is vital for enhancing its stability. Notably, the crawler should remain operational even when encountering a status code other than 200. In the specific test case, it's worth noting that the crawler confronted 66 different status codes without terminating and completed the crawling operation. This outcome encompasses status codes from the 1xx, 2xx, 3xx, 4xx, and 5xx ranges.

The "Docs & Content" section provides information regarding the collected and downloaded content. The "Tot. Docs found" metric indicates that 255 documents have been successfully downloaded. Among these, 25 documents are duplicates. This duplication is intentional, as this testing site contains repetitive information designed to verify the functionality of this feature. The "Avg. Docs per page" value is 1, indicating that the site typically presents one document per page. The presence of the "Robots.txt Exists" flag covers the commitment to polite crawling behaviour. The flag is set to True when this file is located and successfully downloaded. It's essential to emphasize the importance of respectful crawling and commitment to the Robots.txt file protocol. Lastly, the "Tot. Errors" metric accounts for various Selenium exceptions⁶ that may arise during the crawling process. As mentioned, many errors are expected since this is a testing site with different edge cases, and not terminating under those conditions is a good sign.

Checking if changing the depth will increase the coverage.

The next step is to evaluate if the crawler jumps between pages and can increase the coverage. We will change the Max Depth from 1 to 10 to evaluate this. This will allow the crawler to jump up to 10 levels deeper, collecting more links and documents. Table 8 shows the configurations used for this test case.

Unfortunately, knowing exactly how many links the crawler-test site contains is challenging. However, since we have evaluated how the crawler behaves when the Max Depth is, one can give some assurance on the rest of the results. The table 9 shows 895 links has been found which was expected as the depth of the crawler has increased from 1 to 10.

The next step is to evaluate if the crawler jumps between pages and can increase the coverage. We will change the Max Depth from 1 to 10 to evaluate this. This

⁶<https://www.selenium.dev/selenium/docs/api/py/common/selenium.common.exceptions.html>

Seed URL	https://crawler-test.com/
Allow Multi Elements	False
Max Pages	1000
Threads	1
Max Depth	10
Pagination	None
Actions	None
Inspectors	//*[contains(@class, 'large-12 columns')]
Max Docs	1000

Table 8: Crawler configuration

will allow the crawler to jump up to 10 levels deeper, collecting more links and documents. Table 8 shows the configurations used for this test case.

Unfortunately, knowing exactly how many links the crawler-test site contains is challenging. However, since we have evaluated how the crawler behaves when the Max Depth is, one can give some assurance on the rest of the results. The table 9 shows 895 links has been found which was expected as the depth of the crawler has increased from 1 to 10. One excluded link indicates that the found link is excluded as it is restricted by the Robots.txt file. We can not that both the average processing and rendering time have increased this is a sign that indicates more biggere slower pages have been found.

Links	Collected: 895 . Visited Correctly: 697. Cross Site: 198 . Already Visited: 0. Excluded: 1
Time	Tot. Spent: 760.84 s. Avg. Processing: 3.26 s. Avg. Page Rendering: 1.03 s
Status Codes	101: 1 102: 1 200: 220 201: 1 202: 1 203: 1 206: 3 207: 1 226: 1 300: 1 305: 2 306: 1 400: 1 401: 1 402: 1 403: 2 404: 7 405: 1 406: 1 407: 1 408: 1 409: 1 410: 1 411: 1 412: 1 413: 1 414: 1 415: 1 416: 1 417: 1 418: 1 419: 1 420: 1 421: 1 422: 1 423: 1 424: 1 426: 1 428: 1 429: 1 431: 1 440: 1 444: 1 449: 1 450: 1 451: 1 494: 1 495: 1 496: 1 497: 1 498: 1 499: 1 500: 2 501: 1 502: 1 503: 1 504: 1 505: 1 506: 1 507: 1 508: 1 509: 1 510: 1 511: 1 520: 1 598: 1 599: 1
Docs & Content	Tot. Docs Found: 375. Duplicated Content: 667. Avg. Docs Per Page:1. Average page size: 0.0562
Robots.txt Exists	True
Tot. Errors	21

Table 9: Crawler results

Multi threading

Seed URL	https://crawler-test.com/
Allow Multi Elements	False
Max Pages	1000
Threads	4
Max Depth	10
Pagination	None
Actions	None
Inspectors	//*[contains(@class, 'large-12 columns')]
Max Docs	1000

Table 10: Crawler configuration

Links	Collected: 895 . Visited Correctly: 697. Cross Site: 199 . Already Visited: 227. Excluded: 1
Time	Tot. Spent: 760.84 s. Avg. Processing: 3.26 s. Avg. Page Rendering: 1.03 s
Status Codes	101: 2, 102: 2, 200: 823, 201: 2, 202: 2, 203: 2, 204: 1, 206: 5, 207: 2, 226: 2, 300: 2, 305: 4, 306: 2, 400: 2, 401: 2, 402: 2, 403: 4, 404: 66, 405: 2, 406: 2, 407: 2, 408: 2, 409: 2, 410: 2, 411: 2, 412: 2, 413: 2, 414: 2, 415: 2, 416: 2, 417: 2, 418: 2, 419: 2, 420: 2, 421: 2, 422: 2, 423: 2, 424: 2, 426: 2, 428: 2, 429: 2, 431: 2, 440: 2, 444: 2, 449: 2, 450: 2, 451: 2, 494: 2, 495: 2, 496: 2, 497: 2, 498: 2, 499: 2, 500: 4, 501: 2, 502: 2, 503: 2, 504: 2, 505: 2, 506: 2, 507: 2, 508: 2, 509: 2, 510: 2, 511: 2, 520: 2, 598: 2, 599: 2
Docs & Content	Tot. Docs Found: 533. Duplicated Content: 333. Avg. Docs Per Page: 1. Average page size: 0.0562
Robots.txt Exists	True
Tot. Errors	21

Table 11: Crawler results

It is crucial to assess the scalability of the crawler, especially since it employs intelligent multi-threading for link sharing. We must thoroughly test it. We will use the same configurations detailed in Table 8 but adjust the threads parameter and monitor the overall time spent. The results are presented in Figure 12, illustrating the outcomes of eight different runs using identical crawler settings while varying the number of threads. When we switch from one thread to two, we observe a reduction in time from 722 seconds to 595 seconds, representing a 17.5% improvement,

increasing the threads to four results in a time of 438 seconds, which is a 39.33% improvement compared to using just one thread. However, further increasing the threads to six or eight does not yield any additional performance enhancements; in fact, it begins to have a detrimental effect, causing the time spent to increase. This is primarily because all threads communicate to share unvisited links and avoid re-visiting links already processed by other threads. Consequently, as the number of threads increases, the communication overhead escalates. Users must fine-tune this parameter until they find the optimal setting, which appears to be four threads.

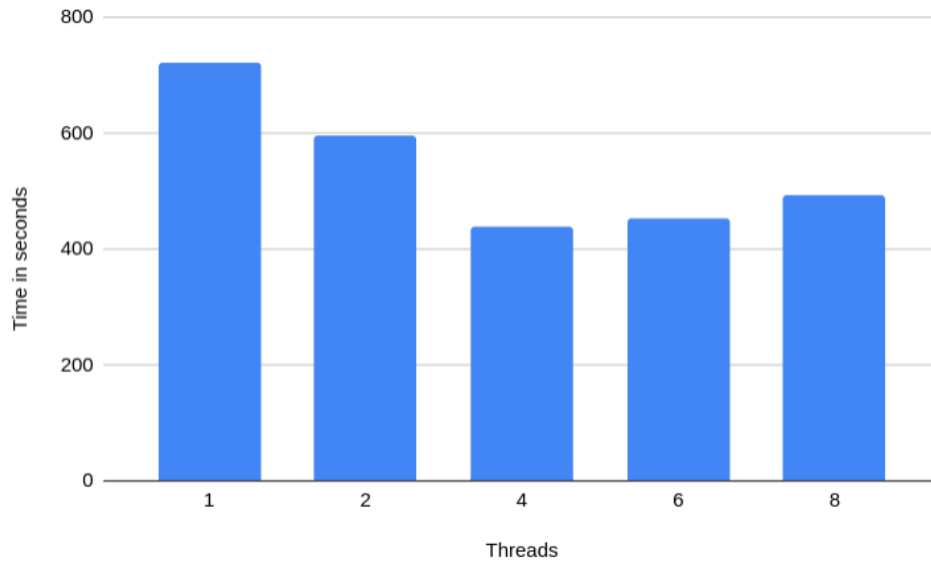


Figure 12: Comparing threads performance.

Another vital step in evaluating the effectiveness of multithreading is examining the distribution of shared links among threads. This is crucial to prevent one thread from overburdening while others remain idle, which would be inefficient, especially when using cloud services like AWS, where resources come at a cost. We'll perform this evaluation using the same configurations from Table 8, employing four threads.

Figure 13 displays the results of four runs, with each run showcasing a distinct distribution of crawled links among threads. While the ideal scenario would involve each thread handling 25% of the discovered links, the averages in the columns reveal variations. Thread 4 crawls more than 25%, while Thread 1 crawls less, which is normal due to considerations like communication overhead and thread-specific

conditions. Additionally, threads won't split links if their queue contains fewer than five links, further affecting equal distribution.

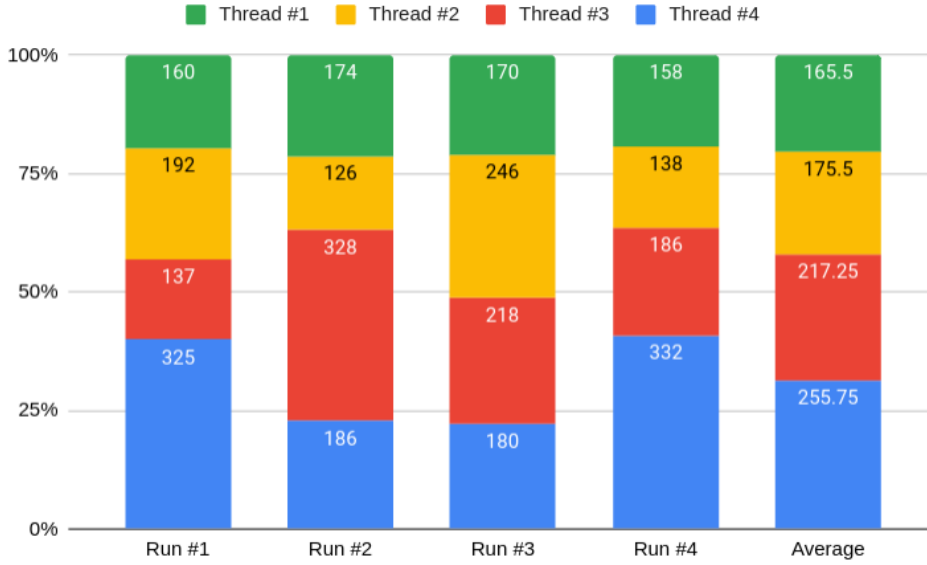


Figure 13: Threads documents distribution.

In order to assess the versatility of the web crawler and its adaptability to various usage scenarios, we will test three additional websites. The initial test case involves crawling a university ranking website to retrieve comprehensive information about all the universities listed, including their titles, respective countries, and current world rankings. The configuration parameters used for crawling this university-ranking website are detailed in Table 12. To accommodate the structure of the website, where each page displays a table containing 25 universities, the "Allow Multi Element" flag is set to True. Considering the pagination feature on the website, which goes up to page 94, we have set the "Max Pages" parameter to 100, as it is unlikely that there will be more than 100 pages to crawl. Since we aim to extract three distinct pieces of information from the table, namely each university's Title, Location, and Ranking, we require three separate inspectors. Given that each page comprises 25 universities, and there are 94 pages in total, we estimate a maximum of 2,350 documents to be collected during the crawling process.

The collected and visited links are accurate and match the expected count of 94. The fact that there are zero already visited links indicates that no duplicate URLs

Seed URL	<code>https://www.timeshighereducation.com/world-university-rankings/2023/world-ranking</code>
Allow Multi Elements	True
Max Pages	100
Threads	1
Max Depth	100
Pagination	<code>//*[contains(@class, 'pagination')]</code>
Actions	None
Inspectors	<code>//*[contains(@class, 'ranking-institution-title')]</code> <code>//*[contains(concat(' ', normalize-space(@class), ' '), ' location ')]</code> <code>//*[contains(@class, 'rank') and contains(@class, 'sorting_1') and contains(@class, 'sorting_2')]</code>
Max Docs	2350

Table 12: Crawler configuration

have been encountered. The total time spent during this process is approximately 8 minutes, which is significantly shorter compared to a similar test conducted using ParseHub, which took 20 minutes to complete all 94 pages. Interestingly, even though the crawler successfully parsed and collected the results correctly, it consistently received a 403⁷ status code in response to all HTTP requests instead of the expected 200. This issue may be attributed to the website’s use of Cloudflare service, as suggested by the information available at ScrapeOps⁸. In this particular use case, the number of collected documents representing universities in the table matches the total results displayed in the page pagination, totalling 2345. It is worth noting that the website also deploys a Robots.txt file, which the crawler successfully detected and used.

The following use case involves extracting a specific category of products from an e-commerce website. In this scenario, we will gather various types of data, including text and images. You can find the crawler configurations for this task in Table 14, designed to scrape a particular product category. Given that the Seed URL link’s pagination indicates the presence of 7 pages, we can configure the Max Pages and Max Docs parameters to be 10. We can employ multithreading to increase performance by setting the number of threads to 4. Douglas employs lazy loading

⁷The HTTP 403 Forbidden status code signifies that the server comprehends the request but declines to grant authorization for it.

⁸<https://scrapeops.io/web-scraping-playbook/403-forbidden-error-web-scraping/>

Links	Collected	Visited Correctly	Already Visited	Cross Site	Excluded
	94/94	94	0	0	0
Time	Tot. Spent		Avg. Processing	Avg. Page Rendering	
	351.66 s		2.57 s	1.020 s	
Status Codes	403: 94				
Docs & Content	Tot. Docs	Duplicated Content	Avg. Docs Per Page	Avg. Page Size	
	2345	0	25	0.2227	
Robots.txt Exists	True				
Tot. Errors	0				

Table 13: Crawler configuration

for image loading, which, in turn, causes the crawler to fetch only 30 out of the available 48 products on the page. To address this issue, we can utilize the actions tab to introduce a scrolling action, repeating it ten times until we reach the bottom of the page. The ability to configure automated actions depends on the website’s functionality. For instance, if the website experiences prolonged loading times, we can include a waiting action to account for the estimated waiting time. If the website necessitates a clicking action to reveal more information, the click action can be utilized for that purpose. The inspectors in this context encompass four fields: brand, title, price, and product images.

Seed URL	https://www.douglas.de/de/c/parfum/damenduefte/duftsets/010111
Allow Multi Elements	True
Max Pages	10
Threads	4
Max Depth	10
Pagination	//*[contains(@class, 'pagination')]
Actions	Scrolling down 10 times
Inspectors	//*[contains(@class, 'top-brand')] //a[contains(@class, 'product-tile__main-link')]/div[1]/div/img //*[contains(@class, 'text')][contains(@class, 'name')] //div[contains(concat(' ', normalize-space(@class), ' '), ' price-row ')]
Max Docs	1000

Table 14: Crawler configuration

Table 15 displays the outcomes obtained following the execution of the Douglas crawler. The "Collected" and "Visited" links align with the expected numbers within the targeted pagination. The estimated time taken for this operation is approximately 6.7 minutes. Notably, the "Average processing time" is more than double the time reported in the uni-ranking results in Table 13. The primary reason for this extended processing time is the inclusion of additional scrolling-down actions.

It is noteworthy to consider an alternative approach: instead of scrolling down ten times to reach the page's end for image loading, why not employ a "scroll to the end of the page" event? This approach was tested on the Douglas website but proved ineffective. The reason is that specific frontend frameworks only load content when it is within the browser's view. Additionally, some websites, such as Facebook, initially display a limited number of posts on a user's home page, progressively loading more as the user scrolls down. In such cases, a single "jump to the end of the page" action will not suffice, as multiple scrolling-down actions are required.

The total count of collected documents indicates that 245 products were downloaded, which appears to be less than anticipated. Given that there are seven pages, with the first page containing 48 products, the theoretical result should be around 336 products. Further investigation revealed that only some pages contain exactly 48 products; some have more, while others have fewer.

It is important to note that, despite employing the robots.txt file for politeness and ensuring a relatively low crawler request rate (calculated as the number of threads divided by the Average Page Rendering, yielding 1.516 requests per second, which is relatively low and unlikely to overload the server), the IP address was eventually banned, and access to the site was blocked after several attempts. This highlights that each website may have its own unique security implementation based on its firewall⁹ rules and the reverse proxy¹⁰ it uses.

Additionally, it's worth mentioning that the Douglas crawler was used without issue for three months, but a ban was encountered recently. This underscores that websites can adapt and modify their security measures over time.

⁹A firewall is a network security tool that filters and controls network traffic to safeguard against unauthorized access and cyber threats, serving as a barrier between trusted internal networks and untrusted external networks, such as the Internet.

¹⁰A reverse proxy is a server or software component that sits between client devices and a web server, forwarding client requests to the appropriate server and often providing additional functionalities like load balancing, caching, and security protection.

Links	Collected	Visited Correctly	Already Visited	Cross Site	Excluded
	7/7	7	0	0	0
Time	Tot. Spent		Avg. Processing	Avg. Page Rendering	
	395.209 s		7.87 s	2.638 s	
Status Codes	200: 7				
Docs & Content	Tot. Docs	Duplicated Content	Avg. Docs Per Page	Avg. Page Size	
	245	0	49	1.509	
Robots.txt Exists	True				
Tot. Errors	0				

Table 15: Crawler results

Parshub encountered a crash while running Douglas’s project, resulting in the error message: "Segmentation fault (core dumped)." Although this made it challenging to compare performance, it shed light on Parsehub’s stability issues, as Parsehub frequently struggles to handle websites without crashing.

Another use case involved crawling Stackoverflow questions, focusing solely on the "python" tag in the seed URL to retrieve Python-related questions. The configured inspectors collected question titles, descriptions, and vote counts. Initially, running the crawler with four threads led to a website ban after only ten pages were crawled. To resolve this, I reduced the thread count to one, reducing the number of requests and resolving the issue.

Seed URL	https://stackoverflow.com/questions/tagged/python
Allow Multi Elements	True
Max Pages	100
Threads	1
Max Depth	100
Pagination	//*[contains(@class, 's-pagination')]
Actions	None
Inspectors	//*[contains(@class, 's-post-summary-content-title')] //*[contains(@class, 's-post-summary-content-excerpt')] //*[contains(@class, 's-post-summary-stats-item__emphasized')]
Max Docs	1000

Table 16: Crawler configuration

Table 17 presents the crawler’s results after this thread adjustment. Notably, the collected links exceeded those displayed in the pagination, indicating an issue with the pagination selector "s-pagination" collecting additional links. The number of visited pages reached 100, the configured limit, as intended, preventing the crawler from continuing to crawl all 27,200 found links. Many cross-site and excluded links signalled that the crawler had lost track and was collecting incorrect links. While 885 documents were collected correctly, there was no guarantee that they were all related to the chosen "Python" topic. Fortunately, termination conditions like Max Pages, Max Docs, and Max Depth were in place to conserve resources.

To troubleshoot the crawler, I enabled the 'Show Browser' option and reran it. This allowed for easier visualization of the crawler’s behaviour and the links it was crawling. It revealed that the crawler was indeed lost and opening the wrong links. Despite the correct seed URL and pagination, the pagination selector 's-pagination' was missing from the configuration. This issue demonstrates how easy it is to debug and identify problems when a crawler loses its way, highlighting the crawler’s politeness and stability.

Links	Collected	Visited Correctly	Already Visited	Cross Site	Excluded
	27200	100	0	460	666
Time	Tot. Spent		Avg. Processing	Avg. Page Rendering	
	588.505 s		6.38 s	0.561 s	
Status Codes	200: 99, 404:1				
Docs & Content	Tot. Docs	Duplicated Content	Avg. Docs Per Page	Avg. Page Size	
	885	241	14	0.155	
Robots.txt Exists	True				
Tot. Errors	0				

Table 17: Crawler results

After fixing the second issue and rerunning the crawler, it operated correctly and yielded results in Table 18. Notably, cross-site and excluded links were reduced to zero, a positive sign. Additionally, the number of collected links was lower than in the first attempt, totalling 900, with nine links collected per page. Interestingly, there were a significant number of 404 and 429 status codes. To address this, it could be beneficial to include a wait action between requests to mitigate the 429 errors.

When the same test was conducted using ParseHub, it took 20 minutes to complete, which was slower than the crawler’s 6-minute runtime. It is worth noting that the two issues encountered during crawling were not experienced with ParseHub. This is because ParseHub’s request rate is slower, reducing the risk of being banned. This is achieved by reducing the number of threads and can be further improved by adding wait actions. The second issue, concerning incorrect selectors, is where ParseHub shines as it offers an easy autodetect feature, simplifying selector selection with a simple click instead of manual XPATH insertion as required in the current crawler implementation.

Links	Collected	Visited Correctly	Already Visited	Cross Site	Excluded
	900	100	0	0	0
Time	Tot. Spent		Avg. Processing	Avg. Page Rendering	
	354.734 s		2.66 s	0.155 s	
Status Codes	200: 54, 404:21, 429: 25				
Docs & Content	Tot. Docs	Duplicated Content	Avg. Docs Per Page	Avg. Page Size	
	2750	0	50	0.155	
Robots.txt Exists	True				
Tot. Errors	0				

Table 18: Crawler results

5.3 Indexer

Following the initial crawling phase, the next step involves indexing, which requires a dedicated section for evaluation. To perform a thorough assessment of indexing, we will utilize a real-world dataset obtained through one of the crawlers employed during the evaluation process. We selected the Stack Overflow dataset presented in Table 16 out of the three available use cases. The primary rationale for this choice is its larger size compared to the others, along with the presence of descriptions that can be employed for index evaluation. It is important to note that the crawler was rerun to gather additional Stack Overflow posts.

File Size	1.4MB
Entries Count	2415
Words Count	108122
Fields	Title, Summary, Votes

Table 19: Stack Overflow posts dataset

5.3.1 Datasets

5.3.2 Metrics

We assess precision at a given value k ($P@k$), calculate the average precision (AP), and compute the normalized discounted cumulative gain at a specific position k ($nDCG@k$).

Precision at k ($P@k$)

$P@k$ represents the proportion of valid predictions within the system’s top k predictions. We define Q_{valid} as the collection of valid predictions for a question prefix q , as specified in the ground truth. Additionally, we denote Q_{result}^k as the set of the system’s top k completion predictions for a given question prefix q . The calculation for $P@k$ is as follows:

$$P@k = \frac{|Q_{valid} \cap Q_{result}^k|}{k} \quad (5.1)$$

We will compute the precision at 5 ($p@5$) for all the various indexing configurations.

Average Precision (AP)

Consider R_1 through R_k as the ordered list of positions where relevant documents are located within the result list of a specific query. In this context, Average Precision (AP) is computed as the mean of the k Precision at R_i ($P@R_i$) values. AP is computed as:

$$AP = \frac{\sum_{i=1}^n P@r_i}{n} \quad (5.2)$$

For the predictions from Q_{valid} that are absent in Q_{result} , we assign a Precision at

position r_i (P@ri) value of 0. We then calculate the average precision by averaging these values across all question prefixes in the ground truth.

Mean Precisions (MP@k, MP@R, MAP)

Having a benchmark containing multiple queries and their corresponding ground truth data, we can assess the system’s performance by calculating the average value of a specific metric across all the queries.

MP@k represents the mean precision at k values across all queries, MP@R represents the mean precision at R values across all queries, and MAP signifies the mean average precision values across all queries.

5.3.3 Experiments

To assess the performance of our indexing, we will employ the Stack Overflow dataset listed in Table 19. We will modify the indexing attributes’ settings and examine how these changes impact the evaluation metrics.

We will initiate our indexing process using the default settings specified in Table 20. The Stack Overflow dataset consists of three inspector fields: Title, Summary, and Votes. However, we intend to index only the textual fields, such as Title and Summary, while retaining Votes as they contain numerical data intended solely for ranking purposes. All other configuration parameters are set to their default values. For a clearer understanding of the configuration attributes, please refer to Table 4, which provides descriptions of each attribute.

Inspectors	Title, Summary			
BM25 Parameters	B=0.75, K=1.75			
Stop Words	None			
Small Words Threshold	2			
Q-gram	3			
Boosting Formula	None			
Result	MP@5	MP@R	MAP	Time(s)
	0.73	0.72	0.87	0.31

Table 20: Stack Overflow indexing configuration, test the default settings without any changes.

Upon initiating the indexing process for the first time, without the availability of any caching, it may take up to two minutes to complete. This indexing procedure consists of two distinct stages: the first involves the creation of a dictionary, which aids in providing suggestions in the dropdown menu to help users locate the appropriate queries. Creating this dictionary is the longer of the two stages, typically taking around 1.8 minutes, while the indexing phase takes approximately seven seconds. The primary disparity between these stages lies in the size of the entities involved; the dictionary comprises 2.6 million entities, whereas the Stack Overflow entities used for indexing consist of only two thousand entities. It is worth noting that the overall duration of the indexing process is highly contingent on the size of the file being indexed and, in this case, the volume of documents crawled by the web crawler. Following the initial indexing process, the dictionary index will be cached and no longer require further indexing.

Even though the search results return 25 documents, we will set the value of k for the $P@k$ metric to 5. This choice is based on the everyday user preference for focusing on the top results in a search list. The overall metrics are presented in Table 20. While these results indicate reasonable accuracy, it is essential to acknowledge that assessing relevance can be subjective, as it varies among users. For instance, Google’s ranking system considers factors like user location, link authenticity, and text matching, leading to potentially inconsistent results for the same query across different users. Furthermore, aiming for an extremely high level of accuracy can lead to model overfitting¹¹, causing it to struggle with generalization. While it may achieve a high precision score with benchmark data, it may need to improve when faced with new, unseen queries. This is because all the model’s parameters have been fine-tuned to fit the benchmark datasets perfectly. Therefore, balancing achieving a reasonable precision score and ensuring that the model performs well on new, previously unseen datasets is crucial. Therefore, achieving perfect accuracy scores in evaluations is challenging and involves a trade-off. The Precision@K metric is significantly affected by the selection of both the value of K and the relevance threshold. Different choices for K and the threshold can result in varying Precision@K scores for the same model. As a result, it is essential to carefully and consistently choose these parameters when comparing different models.

¹¹Overfitting in data science happens when a model fits its training data too closely. This leads to poor performance when dealing with new, unseen data.

Inspectors	Title, Summary			
BM25 Parameters	B=0.1, K=0.81			
Stop Words	None			
Small Words Threshold	2			
Q-gram	3			
Boosting Formula	None			
Result	MP@5	MP@R	MAP	Time(s)
	0.8	0.84	0.89	0.31

Table 21: Stack Overflow indexing configuration

The initial parameters to adjust are b and k. Table 21 displays the configuration of the Stack Overflow indexer, with modifications made to the default b and k values. These values have increased. The key takeaway is that modifying the attributes available in the indexing user interface can enhance or diminish accuracy, providing users with a convenient way to fine-tune their model.

Inspectors	Title, Summary			
BM25 Parameters	B=0.1, K=0.81			
Stop Words	how, to, by, with, in, not, does			
Small Words Threshold	2			
Q-gram	3			
Boosting Formula	None			
Result	MP@5	MP@R	MAP	Time(s)
	0.66	0.624	0.76	0.31

Table 22: Stack Overflow indexing configuration

Let us retain the modified b and k parameters instead of using the default values, as they produce improved results. The following attribute we consider for indexing includes stop words and examining their impact on accuracy. Initially, the intuition is to remove common words such as "how," "to," "by," "with," "in," "not," and "does" from the benchmark queries since they may seem insignificant and lack essential information. Surprisingly, though, removing these words leads to a reduction in accuracy as illustrated in Table 22.

This could be attributed to the Stack Overflow dataset not being exceptionally large, and the assumption that these words are common in the English language

may not hold for some queries in the small Stack Overflow dataset. Furthermore, stop words are not just about eliminating common words; they can also be used to consistently disregard words that typically provide no meaningful information in a query. For example, in the current Stack Overflow dataset, which encompasses all posts related to Python, including the term "python" in the query should have no impact, as all the posts are Python-related, even if they do not explicitly mention the word "python" but discuss Python libraries, for instance.

It is important to note that stop words containing two characters or fewer have no impact in this context and can be safely eliminated. The rationale is that they should already be filtered out due to the Small Words Threshold being set to two.

Inspectors	Title, Summary			
BM25 Parameters	B=0.1, K=0.81			
Stop Words	None			
Small Words Threshold	0			
Q-gram	3			
Boosting Formula	None			
Result	MP@5	MP@R	MAP	Time(s)
	0.8	0.77	0.84	0.31

Table 23: Stack Overflow indexing configuration

In the following evaluation in Table 23, we adjust the Small Words Threshold to zero, which implies that no words are excluded during the indexing process. While the results remain reasonably good, there has been a decrease in precision. This suggests that in certain cases, increasing the Small Words Threshold and not setting it to zero may be advisable. However, it's essential to exercise caution when eliminating small words, particularly those with one to three characters, depending on the language. Additionally, it's worth noting that some two-character words can be acronyms, such as "OS" (Operating System).

The final attribute to adjust is the Boosting Formula. The Boosting Formula is an optional field and particularly useful for ranking documents containing a numeric field. Examples of such fields include product prices, product reviews, post likes, post shares, or the order of items in a list, like the example of university rankings. In the specific context of the Stack Overflow example, each post is associated with an upvote number, which indicates how helpful an answer is. This numeric field can be

Inspectors	Title, Summary			
BM25 Parameters	B=0.1, K=0.81			
Stop Words	None			
Small Words Threshold	2			
Q-gram	3			
Boosting Formula	$\frac{votes}{10}$			
Result	MP@5	MP@R	MAP	Time(s)
	0.53	0.62	0.63	0.31

Table 24: Stack Overflow indexing configuration

a valuable indicator of document quality and relevance. The more users find a post helpful, the more valuable it becomes to display it as one of the top results. To assign a higher score to posts with a high number of votes, we can employ the Boosting Formula. One practical choice is to use the logarithm of the votes. However, since votes can be zero or even negative, more suitable options may exist. We will utilize the formula shown in 24 for this straightforward use case.

5.4 User Experience

Throughout this project, I had the opportunity to work with ParseHub software. While this experience does not make me an expert, I did gain practical knowledge in using ParseHub for various aspects of web crawling, which proved beneficial for the thesis. There are notable advantages and disadvantages when comparing ParseHub with the current implementation used in the thesis.

One of the standout features of ParseHub is its automatic detection of document fields, achieved by clicking. In contrast, my implementation requires manual input of HTML element XPATH to obtain this information. Creating the crawlers manually in my implementation often took more than ten minutes, whereas ParseHub accomplished the same results in half the time. Another advantage of ParseHub is its project-based approach to crawling, as opposed to my implementation's list of crawlers. Starting a crawler in ParseHub can be done by providing only the Seed URL without further options, simplifying the process. However, it is essential to note that this feature can be easily extended and is a manageable hurdle.

In terms of performance, the current implementation excels, as demonstrated in the

evaluation tables. Additionally, the current implementation allows for the addition of more threads and nodes to enhance performance, a flexibility not available in ParseHub. Furthermore, the current implementation has proven resilient in handling various edge cases, whereas ParseHub struggled to manage these scenarios effectively. The adaptability and configuration options offered by the current implementation make it well-suited for various websites and scenarios.

The user interface employed in ParseHub needs to be updated; it lacks responsiveness and features a font that is challenging to read. However, the most frustrating aspect is the frequent occurrence of the browser unexpectedly crashing and shutting down without apparent cause. In contrast, utilizing a frontend framework like Angular with PrimeNG significantly improved the current implementation, making it responsive and delivering a seamless user experience.

One notable drawback of ParseHub is its need for indexing capabilities. When crawling a large dataset, having an indexed version of the data becomes crucial, mainly if the dataset is intended to be served as an API, for example.

6 Conclusions and Future Work

In this thesis, I delved deeply into the complexities of building a search engine and crafting a crawling and indexing process suitable for distributed cluster operation. The crawling phase presented the most formidable challenges among these due to its various edge cases. Achieving the right balance between being a fast crawler and maintaining proper politeness to avoid overloading servers and causing potential Denial of Service (DoS) issues proved one of the most demanding aspects. Additionally, the prevalence of high-speed internet applications posed a tough challenge in creating a crawler capable of effectively handling diverse use cases.

Crawling extensive websites encompassing millions of documents often necessitates multiple machines, as relying on a single machine can be problematic. Fortunately, many gigantic websites offer APIs to access their database information, presenting a viable alternative to crawling.

While evaluating Douglas and StackOverflow, we encountered issues such as IP address bans. To address this, we mitigated the problem by reducing the HTTP request rate by reducing the number of threads. However, other services like ParseHub circumvent this issue by implementing IP Rotation. This technique, while advantageous, typically requires users to invest in an IP pool for rotation purposes. Incorporating this feature into our crawler is a potential enhancement, though it necessitates additional consideration, including cost implications.

I suggest adding IP Rotation if it is affordable and always remaining polite for future work. Furthermore, I am enthusiastic about enhancing our inspection tool to be more user-friendly and accessible, similar to the one utilized in ParseHub. Additionally, I find it fascinating to introduce new features, such as a health bar, to offer users quick insights into the progress of their crawling processes. While we currently display status codes, errors, and helpful statistics, I aim to simplify this information into a more intuitive visualization.

Another route for improvement involves the development of a tailored protocol

that both crawlers and websites can attach to. While using the Robots.txt file is a positive step, there is potential to expand and refine this approach further.

