Master's Thesis

Scriburg: A Configurable Preferential Web Search Engine

Alhajras Algdairy

Examiner: Prof. Dr. Hannah Bast
Advisers: 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

Advisers

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

Online content analysis is crucial in making informed decisions in today's business landscape. However, existing solutions in this space are often proprietary, needing more transparency in their code and flexibility for customization and scalability. In this thesis, We 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 easily used and extended by a wide range of users.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Motivation

Since the beginning of the Digital Revolution, known as the Third Industrial Revolution, in the latter half of the 20^{th} 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 called 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 behavior, 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 **index**[2] 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.

Search engines like Google, Bing, and DuckDuckGo excel at web crawling and indexing, but businesses, especially in e-commerce, require competitive pricing insights

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

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

beyond standard search results. Google's parameters, including brand visibility, user location, SEO[3] ability and more 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. Data scientists do not have to reinvent the wheel to adjust a basic search engine and extend it whenever needed.

1.2 Task Definition

Given a set of URLs, denoted as $U = \{U_1, U_2, U_3, ...U_n\}$, the objective is to identify, for each URL in U, a set of prospective URLs, $\hat{U} = \{\hat{U}_1, \hat{U}_2, \hat{U}_3, ...\hat{U}_n\}$, to extend the existing URLs in U, creating $U_{tot} = U \cup \hat{U}$.

Furthermore, for each URL U, the goal is to specify the desired documents for downloading and indexing, forming a set denoted as $D = \{D_1, D_2, D_3, ...D_n\}$.

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 search engine?
- 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?

[3]SEO, 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 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 DOM[4] 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, describing 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.

[4]The 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 [Pinkerton, 2000], is considered the first actual web crawler-powered search engine. One of the significant innovations of WebCrawler was its **full-text**[1] 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, 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% [Statista, 2023], surpassing 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 available online, a practice that has yet to be observed among other search engines like Bing, making it easier to study.

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

Although the previously mentioned search engines 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 Scriburg 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**[2] 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 Scriburg, we aim to reduce the programming workload and save time by offering a user-friendly interface that can be effortlessly configured for individual websites. Moreover, Scrapy also does not render the JavaScript content out of the box and requires an extra library named Splash [3].

[2]Website pagination is dividing a long list of content or search results into multiple pages to make it easier for users to navigate and access information.

[3]Scrapy documentation: `https://docs.scrapy.org/en/latest/topics/dynamic-content.html`

Selenium: It is an open-source, robust, and adaptable solution for web scraping, automating browser actions, interaction with web pages, and data extraction from online sources. It shares some features with the Beautiful Soup 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 pagination. 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: 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 data extraction simplicity. ParseHub has free and paid plans[4], 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 5.

2.2 Google High-Level 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 [Brin and Page, 1998]. On the contrary, Scriburg is implemented in **Python 3.10**. 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.

Figure 1 shows the basic Google high-level architecture. Google employs a distributed crawler system to retrieve web pages from the internet. The *URL Server* maintains a list of discovered URLs that require crawling, effectively serving as a *load*

[4]ParseHub plans: `https://www.parsehub.com/pricing`

balancer[5] 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 *doc ID* 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. The *Indexer* later organizes these *hits* into *Barrels*. 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 [Brin and Page, 1998].

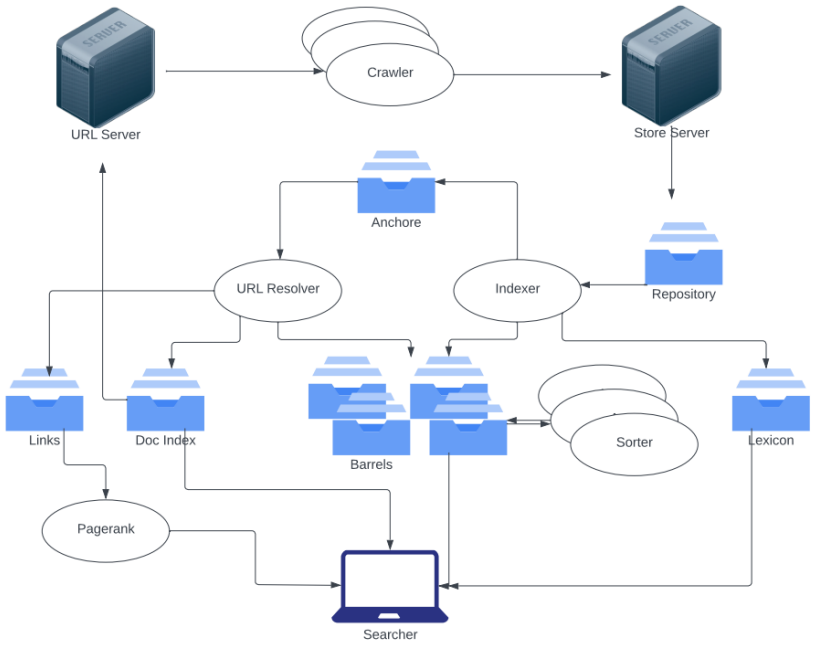


Figure 1: High-level view of Google search engine architecture, showing its main components [Brin and Page, 1998].

The *URL Resolver* reads the links from the *anchors'* file and converts the relative URLs into absolute URLs. The URLs are then assigned to their *doc ID*. The links database saves pairs of *doc IDs* that will be used to compute *Page Ranks* for all the documents.

[5] A load balancer is a network device or software application that evenly distributes incoming network traffic across multiple servers or resources to enhance efficiency and ensure high availability.

Initially organized by *doc ID*, the *barrels* are then rearranged by the *sorter* based on *word ID*. This process generates an inverted index. Moreover, the *sorter* generates a list of *word IDs* and corresponding offsets within the inverted index. More explanation about what an **inverted index** is and how to implement one will be discussed in Section 3.3.3.

3 Background

This chapter 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 it 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 (SERPs)**, present the returned results from a search. The result is then ranked based on predefined criteria.

Web search engines use *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 Requirements and Features

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

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

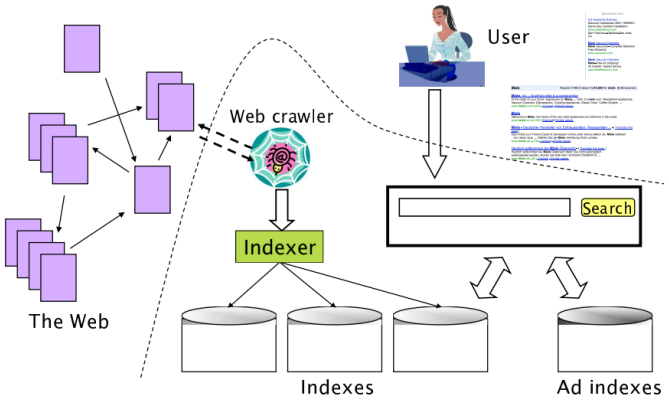


Figure 2: An overview of a generic search engine system [Schütze, 2014]

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. In Scriburg, this parsing occurs during the crawling process. It is worth noting that, in Scriburg, pages are not downloaded; the targeted documents are parsed and stored in the database, and the pages are discarded immediately. We only store the required information from the page, not the entire page. More explanation on crawling and indexing will be discussed in sections 3.2 and 3.3.

Ranking and Relevancy: As indicated in Figure 2, 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 relevant documents at the top and the less relevant ones further down is crucial. More explanation on ranking will be discussed in section 3.4.

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. More information on distributing the workload can be found in section 4.2.1.

3.2 Crawler

The essential role of crawlers is to effectively and reliably collect as much information from the web as possible. 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 web pages of a specific topic or domain; instead, they continuously traverse links without limitations, collecting all encountered web pages. Google and Bing are classified as universal search engines. The second category is called **Preferential crawler (Focused crawler)**. Focused crawlers target specific topics, themes, or domains [Kumar et al., 2017]. They are designed to gather information from a particular domain or subject area, providing specialized search results. Scriburg falls into this category as it only focuses on a subset of links on the internet.

3.2.1 Crawler 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 [Manning et al., 2008].

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 implementation. 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. Making it scalable when needed.

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

3.2.2 Crawler Architecture

Figure 3 shows a basic crawler architecture. The *Fetch* module communicates with the internet and collects the pages passed by the *URL Frontier* module using HTTP requests. 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 following links to be passed to the *URL Frontier* and also to parse any value needed from the page, like text and images. The next step involves filtering the parsed document to eliminate previously visited URLs, duplicate content, and pages 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 [Manning et al., 2008]. The DNS will be excluded in this thesis and will not be discussed.

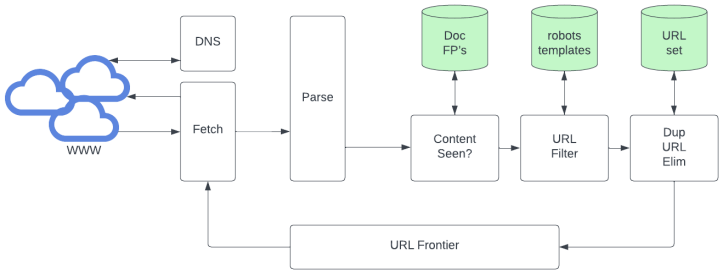


Figure 3: An overview of a web crawler architecture [Manning et al., 2008].

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[2] stored in *Doc FP's* database. Next, newly parsed URLs are filtered based on various criteria, such as excluding

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

[2] A checksum is a numerical value computed from data to verify its integrity by detecting errors or changes in the data.

URLs outside the target country or restricted URLs. Website administrators can specify additional filtering rules, often outlined in a `robots.txt`[3] 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)**.

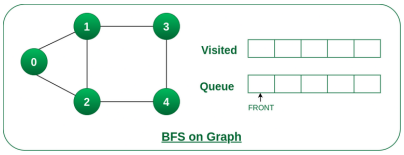
Breadth First Search (BFS): Considering the link planned for crawling as a **vertex** (node within a graph), it is worth noting that web pages can be conceptualized as graphs rather than trees. In contrast to trees, graphs can include cycles, which means we might revisit the same vertex. For instance, a basic illustration of this is the home page link, which essentially represents a self-loop[4] in a graph because clicking on it will land us on the same page.

To prevent looping and revisiting already visited vertices (pages in the context of the web), we can maintain two distinct data structures: *"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 not visited). The second data structure is a *queue* containing links that still need to be visited.

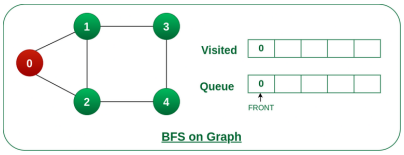
Figure 4 illustrates the BFS algorithm in operation to enhance visual understanding. Let us visualize a seed URL, denoted as vertex 0. The seed URL page contains two additional links, 1 and 2. The page represented by vertex 1 contains three links to pages 0, 2, and 3. Similarly, the page associated with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

[3] A robots.txt file is a text file on a website that instructs web crawlers and search engine robots on which parts of the site should be crawled or excluded from crawling. Source: `https://en.wikipedia.org/wiki/Robots.txt`

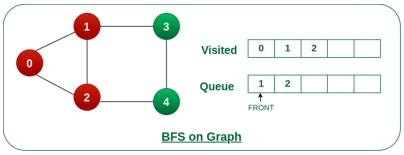
[4] A self-loop in a graph is an edge that connects a vertex to itself, creating a loop originating and ending at the same point.



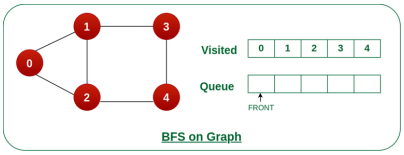
(a) Initial condition



(b) Visit the 0 vertex



(c) Visit 1 and 2 vertices



(d) Completed condition

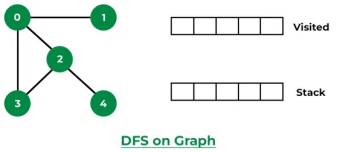
Figure 4: Illustration of the Breadth First Search (BFS) algorithm visiting five URLs using a queue [geeksforgeeks, 2023a].

previously visited nodes (pages).

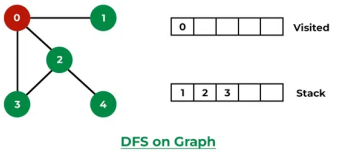
Initially, the *queue* and the *hashmap* are empty of any entries, which is the ideal state of the crawler. As the crawler launches its crawling journey, it begins by pushing the seed URL, node 0, into the *queue*. Then, node 0 will be crawled since it is at the beginning of the *queue* and flagged as visited in the *hashmap*. 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 pushing 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 [Cormen et al., 2001]. 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 the graph. The graph has an acyclic structure in our implementation since it prevents revisiting previously visited links. Consequently, the time complexity for the crawler is $O(|V|)$.

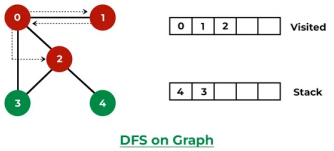
Depth First Search (DFS): It operates similarly to Breadth First Search (BFS). However, instead of visiting the nodes (pages) discovered first, it explores the most



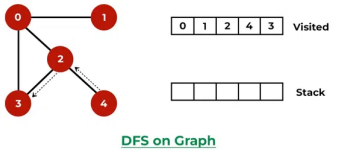
(a) Initial condition



(b) Visit the 0 vertex



(c) Visit 1 and 2 vertices



(d) Completed condition

Figure 5: Illustration of the Depth First Search (DFS) algorithm visiting five URLs using a stack [geeksforgeeks, 2023b].

recently discovered nodes. Unlike BFS, DFS can be implemented using a *stack*. Figure 5 visually represents how DFS operates while crawling a website.

The crawler begins with the seed URL 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*, and after node 0 is visited, we flag it as *"visited"* in the *"visited" hashmap*. 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.

The 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,"* *"university,"* *"of,"* and *"freiburg"*.

Stop Words

Tokens such as *"the"* and *"of"* in the previous example contribute little importance to the overall outcome of the query because the query is equivalent to *"What University*

Freiburg." Although the last query is grammatically incorrect, it contains the most critical tokens to understand the user's intention. This is also why Google understands the user query when it is incomplete. Eliminating these terms via **stop words** can lead to excluding 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. Selecting appropriate stop words can enhance search retrieval by using a more compact indexer while also allowing user queries to bypass terms contained in the stop words list. The exclusion of stop words from the indexer can result in more relevant search results, as the search engine can direct its attention to the informative words within the documents.

3.3.2 Document Unit

The term **document** frequently mentions the specific information intended for retrieval from a web page. 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

Let us define q as the query string provided by the user and D as the documents the user attempts to search in. If we were to implement a simple solution, we would need to iterate over each term in q with a length of $|q|$ and then search each term against every document containing $|D|$ terms. This means three nested for-loop with a cubic complexity of $O(|q| * D * |D|)$.

```
for q_token in q_tokens:
    for document in all_documents:
        for document_token in document.tokens:
            .....
```

Listing 3.1: A naive way of finding a query match in all the documents.

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 document IDs that occurred. The tokenized terms are called dictionaries, and the list of IDs is called **postings**.

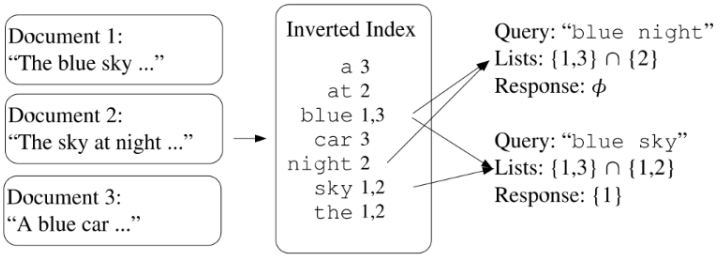


Figure 6: An illustration of an inverted index featuring three documents. All tokens are included in this example, and the sole text normalization applied is converting all tokens to lowercase. Queries that involve multiple tokens are resolved using intersection operations. However, we are using union operation in Scriburg. [Castillo, 2005]

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 tokens $|q|$ has an overall time complexity of $O(|q| * 1)$, which simplifies to $O(|q|)$. The next step involves merging the resulting documents for each token in the query q.

Since each token inside the query q will result in a list of documents (postings), the final result should be the union of each list. The number of lists to merge is $|q|$.

Given that 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 (postings) for each term in the query $|q|$, the number of lists to merge is $|q|$. Consequently, the time complexity merging all the postings is $O(|q| * |L|)$, where $|L|$ represents the average length of the posting list.

It is worth noting that $|q|$ is typically small; 40% of people use two search terms for online search queries in the United States as of January 2020, and only 0.46% uses ten or more [Statista, 2020]. 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, the indexing process prepares a dictionary that can be looked up to find relevant tokens that match the search query q; 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 city and not to return all documents that contain tokens like *"what"* and *"is,"* which are less important than the most informative term in the sentence which is *"Freiburg."* There are many algorithms for document ranking. However, this thesis will adopt **BM25**.

Given a query q, containing keywords $q_1, ..., q_n$, the **BM25** score of a document d is:

$$score(d,q) = \hat{tf}.\log_2(\frac{N}{df}) \tag{3.1}$$

$$\hat{tf} = \frac{tf.(k+1)}{k.\alpha + tf} \tag{3.2}$$

$$\alpha = \frac{1-b+b.DL}{AVDL} \tag{3.3}$$

$\boldsymbol{N}$: total number of documents. $\boldsymbol{tf}$: term frequency, the number of times a word occurs in a document. $\boldsymbol{df}$: document frequency, the number of documents containing a particular word. $\boldsymbol{DL}$: document length. $\boldsymbol{AVDL}$: average document length (measured in a number of words).

The parameter $\boldsymbol{b}$ prevents the impact of document length normalization. It is a numeric value within the range of 0 to 1. When b is set to 0, there is no normalization, implying that longer documents do not face any penalties. In contrast, a value of 1 indicates complete normalization, where longer documents are penalized in proportion

to their length. The default value will be set in the implementation is 0.75.

The parameter $\boldsymbol{k}$ governs the impact of term frequency saturation in scoring. It is a positive parameter that dictates the speed at which the term frequency component of the score achieves its peak value. When k is set to 0, there is no consideration of term frequency, implying that the score remains unaffected by the number of times query terms appear in the document. On the other hand, when k is set to a significant value, more weight is given to term frequency, and the score escalates linearly with term frequency. The default value will be set in the implementation is 1.75.

The following example dives into the details of the *BM25* equation and how it impacts ranking. Table 1 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.

Given that we have three documents, we set the variable N to 3. In the next step, we calculate each document's length DF, resulting in $\{1 : 26, 2 : 21, 3 : 49\}$. The average document length $AVDL$ can be calculated as $\frac{26+21+49}{3} = 32$. Substituting these values into the equation, we obtain the inverted list shown in Table 2.

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 content used as an example of MB25 ranking.

When we analyze the data in Table 2, it becomes evident that tokens shared by all three documents, such as *"the,"* *"of,"* and *"is,"* have corresponding scores of 0. It is normal to give low scores to tokens that contribute little to no to the query. Consider a scenario where a user inputs the word *"the"* in their query. In this case, every document contains this word, making it challenging to maintain relevance. The only way to exhibit bias toward one document over another is when the token's frequency within a document is significantly higher than the document's length.

On the other hand, unique words like *"albert"* and *"ludwig"* receive high scores since

they are exclusive to a single document. Words such as *"freiburg"* and *"university"* have varying scores for each document, depending on their relative occurrences in relation to the document's length.

Token	Postings: (Doc. ID, BM25 Score)
the	(1, 0), (2, 0), (3, 0)
university	(1, 1.071), (3, 0.466)
of	(1, 0), (2, 0), (3, 0)
freiburg	(1, 1.071), (2, 0.975)
officially	(1, 1.740)
albert	(1, 1.740)
ludwig	(1, 1.740)
is	(1, 0), (2, 0), (3, 0)
a	(1, 0.642), (3, 0.885)
public	(1, 1.740)

Table 2: The first ten tokens from the resulting inverted index and the corresponding document scores.

A user's query for *"university of freiburg"* will yield the following results: {(1, 2.142), (2, 0.975), (3, 0.466)}. The initial document with ID 1 receives the highest score (2.142) since it encompasses all three query terms (*"university,"* *"of,"* and *"freiburg"*). This outcome aligns with expectations, given that the first document is indeed about the University of Freiburg.

However, the concern regarding relevancy lies in the two primary keywords, *"university"* and *"freiburg,"* mentioned in documents 2 and 3. The question then becomes, which documents should be prioritized in this scenario? This distinction can be influenced by adjusting the values of the b and k parameters. In this case, document 2 takes importance over document 3 because the term *"freiburg"* is repeated twice, and the document itself is shorter than document 3.

3.5 Fuzzy Search

Frequently, users make spelling errors in their input queries or may employ American English spellings, such as *"color,"*, which is equivalent to British English spellings *"colour."* It would be less than ideal to return no results solely because the user chose a word variant over another. Other situations may occur where users need clarification about the correct spelling of a new term or someone's name. In Scriburg,

we will use the **fuzzy search** feature to offer users suggestions as they type their queries and to identify the closest matching query entered by the user.

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, phrasing variations, or other minor deviations from the original text.

In Scriburg, the fuzzy search method employed for spelling correction is labeled as **isolated-term** [Manning et al., 2008]. **Isolated-term** focuses on correcting individual query terms one by one rather than fixing the entire sentence within a contextual context.

Fuzzy search algorithms typically involve techniques like **Levenshtein distance (Edit Distance ED)**, which calculates the minimum number of single-character edits (insertions, deletions, substitutions) required to transform one string into another. Other techniques include using phonetic algorithms to find similar-sounding words or tokenization and comparing word **q-grams** to identify overlapping substrings. The greater the shared substring between the two texts, the more likely this is the intended word the user was trying to search for. More details on q-grams will be discussed in section 3.5.1.

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 include inserting a character into a string, deleting a character from a string, and replacing a character within a string with another character. In the context of these operations, the term *"Levenshtein distance"* is sometimes used interchangeably with *"Edit Distance"*. For example, the edit distance between *"black"* and *"back"* is one because we need to remove one letter, the *"l"* from *"black"* to transform it *"back."*

In Scriburg, we use **Prefix Edit Distance PED** instead of Edit Distance. Prefix edit distance is a variation of edit distance that focuses on finding the minimum number of edit operations needed to transform one string into a **prefix** of the other. The prefix edit distance between s_1 and s_2 is defined as $PED(s_1, s_2) = min_{s_2'} ED(s_1, s_2')$ where s_2' is a prefix of s_2 [Bast, 2023].

Fuzzy search requires the existence of a **dictionary**, which comprises a collection of words for conducting searches to locate the closest match and then return the result. The creation of this dictionary can take various forms, and in the case of Scriburg, we employ two distinct dictionaries.

The initial dictionary is the same one used during the document indexing process when crawling the web. This choice derives from our desire to ensure that the words indexed from the crawled documents can be effectively looked up.

The second dictionary, which is a user predefined, is presented by using **Wikidata** in Scriburg. This dictionary is particularly valuable in scenarios where synonyms are employed. For instance, if a user inputs *"USA,"* a dropdown menu will display *"United States"* as a suggestion.

In the Scriburg user interface) we offer users the flexibility to select the dictionary of their choice. This option is essential, especially in specific domains where using a particular dictionary is crucial for accurate results. The dictionary construction process is carried out before indexing the crawled documents.

3.5.1 Fuzzy Search With Q-grams

Calculating the prefix edit distance for a given word within a dictionary is computationally costly and can be accelerated. One approach involves using **q-grams**. For a given string s and a natural number $n \in N$, we form the multiset of q-grams, represented as $Q_n(s)$, which encompasses all substrings of length n (Freiburg, 2023).

For example if $s = \textit{"freiburg"}$ and $n = 3$ then the resulting q-grams are:

$$Q_3(\text{"}freiburg\text{"}) = \{\text{"}fre\text{"}, \text{"}rei\text{"}, \text{"}eib\text{"}, \text{"}ibu\text{"}, \text{"}bur\text{"}, \text{"}urg\text{"}\}$$

Where the number of q-grams of a string s can be calculated as:

$$|Q_n(s)| = |s| - n + 1 = 8 - 3 + 1 = 6$$

3.5.2 Q-gram Index

We have already explained what an **inverted index** is in section 3.3.3, where we explained that tokens derived from documents would be organized within a dictionary or hashmap. This q-gram dictionary is structured with q-grams as keys and values containing lists of tokens from the documents, along with the corresponding q-gram frequencies.

Figure 7 presents a visual representation illustrating the fundamental concept of the q-gram inverted list. It is essential to clarify that in the actual implementation, we store references to the tokens associated with the q-gram rather than the tokens themselves as strings. Furthermore, we will also record the frequency of occurrences of the q-gram within a token. The fuzzy search implementation is rooted in the

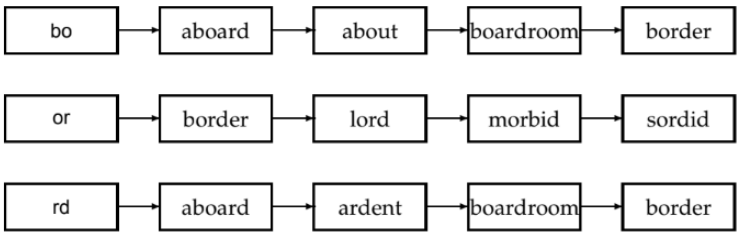


Figure 7: A simple q-gram dictionary where three q-grams are linked to the tokens that contain them [Manning et al., 2008].

principles of the Information Retrieval course [Bast, 2023]. It is highly recommended to review the course lectures for a more detailed understanding of its functionality and implementation guidance.

4 Approach

This chapter outlines our comprehensive software architecture and implementation. Section 4.1 showcases all the required components for constructing the Scriburg search engine. In Section 4.2, we will clarify the functionality of Scriburg's web crawlers. This will contain the workflow, practical challenges encountered during crawling, and corresponding solutions. Moving on to Section 4.3, we will demonstrate the indexing workflow. Finally, in Section 4.4, we will delve into the user interface design, highlighting the configurations made available to users and how they enhance the overall user experience.

4.1 Software Architecture

Figure 8 shows an overview of the software architecture employed by our search engine. Microservices architecture was used to simplify scalability and split each component's responsibilities. **Docker** is used to simplify adding and removing crawling nodes. **Ubuntu 18.04** image is used for each image. Below is a compilation of the utilized technology stack:

Frontend (Angular & PrimeNG): One of the challenges in this thesis involves creating a user-friendly interface for configuring the search engine, targeting non-expert users. We will implement an attractive HTML component with an intuitive interface using the **PrimeNG** framework to address this challenge. Another design decision is to employ client-side rendering[1] with **Angular** to enhance user experience and responsiveness.

Backend (Django): Serving as the core intelligence of the search engine, the backend houses both the crawler and indexer modules. It facilitates interaction

[1] Client-side rendering (CSR) is an approach where web content is primarily rendered and processed in the user's web browser, allowing for dynamic, responsive user experiences but potentially longer initial load times. It's commonly used in single-page applications and may pose SEO challenges due to content generated via JavaScript.

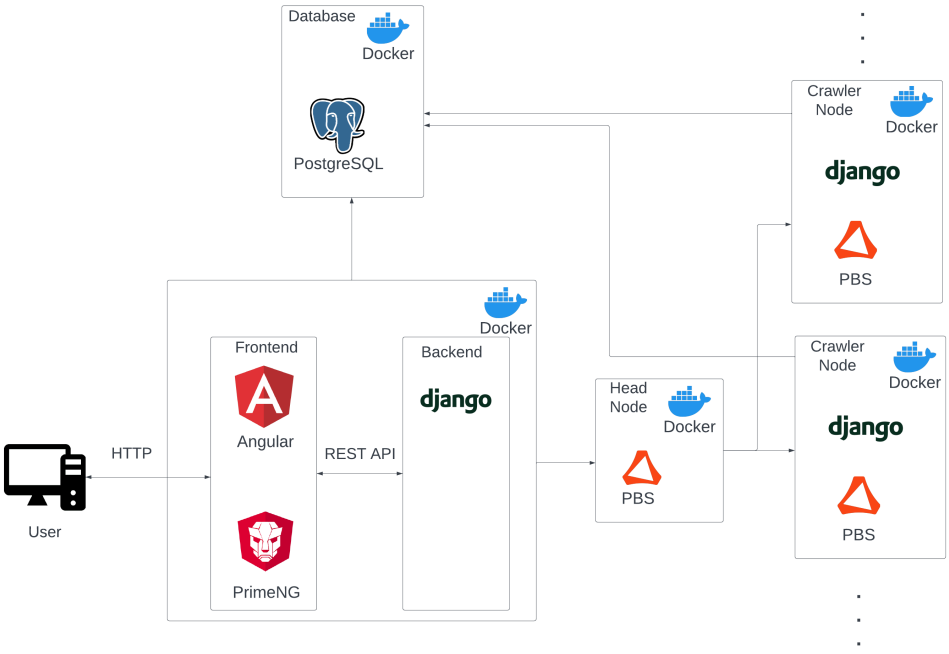


Figure 8: Scriburg software architecture overview.

with the **PBS Head node** to initiate crawling based on user-defined configurations. Moreover, it establishes a connection with **PostgreSQL** database for storing crawler and indexer configurations, along with job-related information. The selection of the **Django** as a backend framework was used for several factors. It offers a user-friendly admin dashboard by default, simplifying adding and removing nodes from the cluster. Additionally, Django provides a robust API library, facilitating seamless communication with Angular. The choice of Python as the primary language was influenced by its simplicity, enabling future developers to modify and expand the codebase easily.

Selenium: Lives under the backend component, served as a web browser automation tool employed to establish a web session that emulates a genuine user session, allowing for the rendering of specific pages intended for crawling and the downloading of documents to be indexed.

Head Node (PBS): Since crawling is a job that can run in multiple nodes and a distributed system is used, a job scheduler must be used in the design. The **PBS head node** is the central node that distributes the workload between crawler nodes. **Portable Batch System (PBS Pro.)** is a job scheduling

and workload management system in **high-performance computing (HPC)** environments. It allows users to submit and manage batch jobs on a cluster of computers, making it easier to utilize the available computing resources efficiently. PBS typically provides job submission, queuing, resource allocation, prioritization, and status monitoring features. It helps administrators and users manage and optimize the execution of computational tasks on a cluster, like crawling in our use case. Other alternatives like **Slurm** can be used instead of PBS as a job management system.

Crawler Node (PBS): The PBS crawler nodes are responsible for web crawling and storing documents within a PostgreSQL database. They are powered by the Django backend for the crawling process. Manually adding crawler nodes is possible, and after adding a node, one can initiate crawling jobs on it as needed.

The application setup initiates with a minimal requirement of four microservices to operate the entire search engine. A Docker container containing both Angular and Django communicating via API. Django communicates with the database to store the crawling and indexing configurations and saves the submitted job metadata and statistics. The PBS head node must be configured at least to use one crawler node. The crawler node will perform the crawling job and save the results to the shared database. The more computing power needed, the more nodes can be added to scale the system horizontally[2].

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 is 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 does not support a distributed architecture and will be executed locally and not on the PBS cluster.

The primary reason for bypassing the distribution of the indexing task is that, in contrast to the intensive computing requirements of crawling, indexing the gathered data often demands significantly less computational power. It is important to note

[2]Horizontal scaling involves expanding a system's capacity by introducing more machines (nodes) instead of enhancing the capabilities of the existing machines.

that this is not an absolute rule, as it relies on the volume of documents to be indexed. However, in the case of specialized crawlers like Scriburg, designed for specific domains rather than comprehensive web crawling, this step can often be avoided.

4.2 Crawler Implementation

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 4.4. Based on the crawler configuration, a thread pool will be created. 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 ensures that if one thread has no URLs, it can ask other threads to split the workload. The choice between a Breadth First Search (BFS) or Depth First Search (DFS) algorithm in the crawler depends on the user's configurations. However, for clarity, Algorithm 1 assumes using a BFS approach, and accordingly, a queue is used instead of a stack.

Algorithm 1 Start Crawling

```
1: load_crawler_configurations()          ▷ Configurations submitted by the user
2: thread ← create_threads_pool() ▷ Generate a thread and add it to the 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: robots_file ← load_robots_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()
12:        load(current_url)                          ▷ Load and render the page
13:        execute_automated_actions()        ▷ If enabled, clicking, waiting, and scrolling actions are executed
14:        new_links ← find_all_page_links()   ▷ Collect all links in the current page
15:        filter_unwanted_urls(new_links) ▷ Remove duplicated, disallowed and cross-origin links
16:        docs ← find_page_documents()  ▷ Download all documents in the page
17:        filter_unwanted_documents(docs)
18:    end if
19: end while
```

A seed URL is pushed into the present thread queue, representing the initial point for the crawling procedure as specified by the user configuration. This seed URL enables retrieving the `robots.txt` content, which is downloaded once and utilized throughout the crawling operation. Typically, `robots.txt` files are located at the root path of the URL, but it is also possible to configure the crawler to fetch them from a user-defined location. Without a specific user configuration, the default behavior involves automatically reading the `robots.txt` file from the root.

Each crawler goes into an infinite loop that will continue to run either if the queue still contains URLs to be fetched or if at least one crawler is still running. This guarantees that although one thread is busy, it contains many URLs that need help with crawling, and the free threads can share the load with it. If the thread queue becomes empty, the thread will ask the pool to find the following URLs to fetch. Otherwise, the next URL in the queue will be fetched, and a Selenium page request will be made. Afterward, automated actions such as scrolling down, waiting, and clicking defined by the user are executed. Those actions give the user the power to control the browser to mimic real agent behavior. The action chain will be discussed more in the user interface design Section 4.4.

Once the page has fully loaded and the specified actions have been executed, the crawler proceeds to retrieve all the next links, filter them, and then push them into the queue. The final step involves downloading the documents intended for indexing.

4.2.1 Threads Pool

The threads pool for sharing URLs includes all the crawlers running on a single machine, not all the clusters. Currently, each node operates independently, so if two crawlers run on different nodes, the visited URLs can be revisited by other nodes. To address this issue, a solution is to transfer the in-memory visited URLs dictionary to a key-value database, such as **Redis**. Instead of including visited URLs in a dictionary shared by the thread pool, they can be stored in the Redis database, which is accessible to all other crawling nodes.

When a thread queue becomes empty, instead of idling, the thread will pause for five seconds and then inspect the threads in the thread pool. The thread with the largest queue will divide it into two halves with the available free thread. If a thread is available and another thread has fewer than five URLs, no division will occur, as the threshold is set at five. Adjusting the splitting threshold to higher values like 20 or 50 would diminish the benefits of multithreading and not enhance performance, as this will result in more ideal free threads. Reducing the threshold to smaller values

like 1 or 2 would lead to frequent queue sharing among all threads, increasing the unnecessary overhead of link sharing. After experimenting with four threads, the optimal balance was a split threshold between five and ten.

4.2.2 Scaling the System

One of our primary goals was to create a system capable of performance scalability by adding low-cost workstations to support extra components. Whether to scale vertically by adding more threads or horizontally by introducing more machines, it can be challenging to decide which scale to use and when. We projected that a single instance of Scriburg would suffice for approximately four to eight crawlers. However, a second crawl manager (a separate crawling system node) would be needed beyond this point. Other approaches, such as those mentioned in [Shkapenyuk and Suel, 2002], recommend introducing additional crawler managers or new crawler nodes when the number of crawlers reaches eight.

4.2.3 Practical Challenges

Canonical URLs: As previously mentioned, avoiding revisiting already-seen pages prevents content duplication and eliminates the risk of the crawler getting stuck in a loop, thereby enhancing the efficiency of the crawling process. Nevertheless, the task of identifying previously visited URLs is complicated by dynamic URLs, where different URLs can ultimately lead to the same page. To address this challenge, every fetched URL must undergo normalization and be transformed into its fundamental form. In 4.1, we illustrate various examples of different URL forms that ultimately point to the same URL. In Scriburg, we adopt a straightforward approach by disregarding parameters and fragments from each fetched URL.

```
https://uni-freiburg.de
https://uni-freiburg.de/
https://uni-freiburg.de/#fragment
https://uni-freiburg.de/index.html
https://uni-freiburg.de/index.html?tab=research
https://uni-freiburg.de/index.html?tab=research&user=12
```

Listing 4.1: Example of different forms of the same URL.

Duplicated Content: Around **29.2%** of the web content is duplicated and it is growing [Fetterly et al., 2003]. While a web crawler avoids revisiting identical URLs to prevent content duplication, it is important to note that identical content may exist in different URL 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 unreliable. Moreover, as we discussed, URLs can have different forms, and sometimes, we revisit the same page unintentionally. A more effective approach involves comparing the content 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**[3] hashing algorithm based on the content string intended for storage. This hash code is then stored in the database. Before saving a new document, we can verify if the hash code 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. Note that sometimes, when the page admin edits their content, even one character edited will be classified as a new document.

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**[4] requests. Selenium establishes a genuine session to address these challenges and fully renders the web page. This approach emulates user interactions using action chains, which simulate waiting, scrolling, and clicking. More details will be discussed in the User Interface Design Section 4.4.

Robustness: The dynamic nature of web content can present numerous ex-

[3] SHA-1 (Secure Hash Algorithm 1) is a cryptographic hash function that takes an input and produces a fixed-length 160-bit (20-byte) hash value, typically represented as a 40-character hexadecimal number. It was widely used for data integrity and security purposes.

[4] AJAX (Asynchronous JavaScript and XML) requests are a technology for sending and receiving data from a web server without requiring a full page refresh, enabling dynamic and asynchronous data updates in web applications.

ceptional scenarios for crawlers, making ensuring comprehensive coverage of these edge cases challenging. Nonetheless, we simplified troubleshooting by implementing logs and employing effective monitoring and termination criteria. We minimized the risk of falling into unnecessary web traps, a particularly crucial aspect when integrating cloud-based computing solutions like **Amazon Web Services AWS**. Crawlers can be halted 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 use 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 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 blocked and subsequently banned. To mitigate this risk, it is 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, more than these precautions are required to prevent users from adding more nodes and executing DDoS attacks on the servers. 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

Unlike crawling, which can be computationally intensive, indexing thousands of documents is relatively lightweight. Hence, it is carried out on the localhost without any multithreading support. Algorithm 2 clarifies the process of creating an inverted list. Firstly, the indexer loads the user-defined indexing configurations from the database. We will list more configurations in Section 4.4. Subsequently, an empty inverted list is initialized. Within Python, one can implement it as a dictionary, with

each word as a key and an associated value representing a list of document IDs that include that word (postings). While one can include additional metadata in the list, such as word frequency and positions of the words in the document, we will omit these details for algorithmic simplicity.

Algorithm 2 Create Inverted List

Require: *documents* not empty

```
1: config ← load_indexer_configurations()        ▷ Load user chosen configurations
2: inverted_list ← {}                            ▷ We initialize an empty dictionary
3: threshold ← get_small_words_threshold(config)
4: stop_list ← stop_words_list(config)
5: docs_length ← [ ]                             ▷ We initialize an array of documents lengths
6: for doc in documents do
7:     doc_length ← 0
8:     tokens ← tokenize(doc)                    ▷ Split the document into tokens
9:     for token in tokens do
10:        if token > threshold and token not in stop_list then
11:            add_token_and_doc_id_to_inverted_list(token, doc.id)
12:            doc_length ← doc_length + 1
13:        end if
14:    end for
15:    docs_length.add(doc_length)
16: end for
17: calculate_bm25_score(inverted_list)          ▷ Compute the BM25 score
18: cache(inverted_list)                         ▷ Cache the indexer for future usage
```

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. Afterward, we iterate 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 for each token

in a document based on equation 3.1. Afterward, it is saved into a cache for future retrieval and use.

4.4 User Interface Design

This thesis extends beyond merely building and designing a search engine. It also encompasses the critical goal of enabling users to configure and utilize it effortlessly. This section will explore the user interface design, workflow, and user-facing configurations.

4.4.1 Templates and Inspectors

The user begins their workflow on the homepage, where they can access documentation explaining how to use the application. The application's first component to be created is the **Template**, which serves as a blueprint for specifying the document fields to be extracted from web pages. Establishing a unique template for each page is a prerequisite, although the same Template can also be applied across different websites. These templates consist of a list of **Inspectors**. An inspector can be thought of as a field. For example, creating a template for parsing a title and a product price will require two different inspectors. Each inspector contains the attributes shown in Table 3.

Name*	Represents the inspector's identifier, such as *"Title"* or *"Price."*
Selector*	It contains the XPath[5] expression identifying the chosen element value.
Type*	This can take values like *"Text,"* *"Link,"* or *"Image,"* signifying the nature of the content to be extracted.
Variable Name	An optional shorthand representation of the selector, facilitating its use during the indexing process to enhance search results (Ranking).
Clean-up Expression List	Used to refine the extracted value from the inspector. This proves beneficial in eliminating unwanted noise (remove the currency when extracting a price).
Attribute	Allows the user to specify an HTML element attribute, such as *"src,"* *"name,"* or *"href,"* as an optional parameter to be saved into the database.

Table 3: Inspector form fields. Fields with * are required.

The **Actions Chain** list can be used depending on the targeted website's characteristics. An **Actions Chain** constitutes an array of sequenced actions replicating user interactions. This functionality is valuable for tasks such as accepting cookies,

scrolling to load additional content or waiting for the website to render in cases where the process may exceed the expected duration.

4.4.2 Crawlers

Once the Template is created, the subsequent step is to access the Crawlers page and create a new Crawler. A **Crawler** comprises various essential configurations as shown in Table 4.

Name*	A user-defined identifier for the crawler
Template*	The blueprint for collecting documents with
Seed URL*	The starting point of crawling
Max Pages	The upper limit for the number of pages to be visited
Max Docs	The maximum number of documents to be collected
Max Depth	Maximum jumps between pages (crawling depth)
Robots.txt	The URL where the `robots.txt` file can be located
Threads	Number of threads used in the crawling process
Pagination	Scope to collect the following URLs
Excluded URLs	URLs that the crawler must refrain from visiting
Walltime (ms)	Sets the duration for which the crawler should continue crawling
Show Browser	Deactivate the headless mode in Selenium

Table 4: Crawler configurations options. Fields with * are required.

4.4.3 Runners

Once the crawler is set up with the appropriate configurations tailored to the targeted website, 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 nodes. It is important to note that each runner can use multithreading based on the crawler's configurations and employ distinct crawler settings. This approach provides an effective means to assess the crawler's performance until 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 online. By default, **localhost** is set as the value, where some will use their local machine as a crawling node.

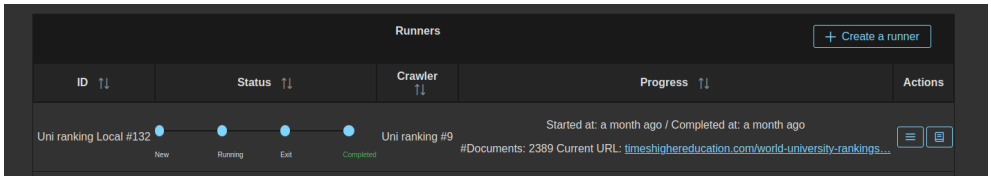


Figure 9: An overview of the runners table, showing the runner's status and progress.

It is essential 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 collects targeted data accurately. The ***Runners*** table provides a straightforward and informative progress overview through four primary status indicators. The initial status is ***New***, representing the runner's initialization before the crawling process begins. ***Running*** signifies that the crawling process is currently undergoing. ***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 collected, 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 drop-down menu.

4.4.4 Indexers

Once the runner has finished its job and the user is satisfied with the results, the collected documents can be indexed and prepared for future searches. To access this feature, one can navigate to the **Indexers** view, where the indexers table displays the current indexers and their respective statuses.

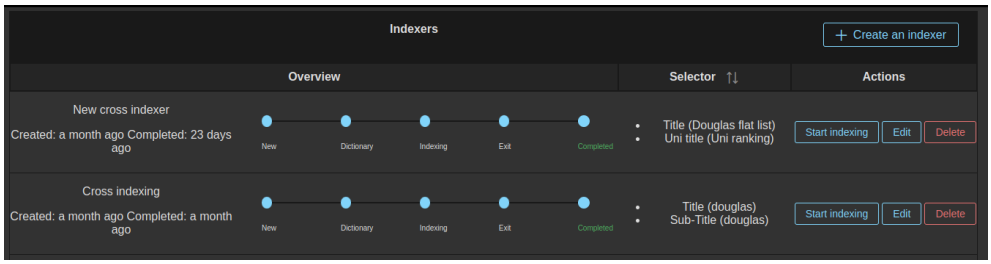


Figure 10: An overview of the indexers table, showing their status and progress.

In this context, inspectors come into play. These inspectors are responsible for

mapping the fields extracted from the document, such as *Title*, *Price*, and *Image*. Users can select which inspectors to index from a drop-down menu, with the choice limited to only text fields. Images and links are excluded from this indexing. Note that if one template is used to fetch from different websites when indexing the collected documents, one can select the inspectors related to this template, and the indexing process will contain all the websites that use this template, which is a nice feature.

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 the indexing process.
Small Words Threshold	The threshold of which the word can be considered small and will be skipped from the indexing process.
Words Weight List	Boost some words by giving them weight, e.g. *"Freiburg=5"* will add more 5 points to the score when the *"Freiburg"* word is found.
Boosting Formula	This formula result will be added to the final score. It uses inspectors variable.
Dictionary File Name	The dictionary file name that helps the suggestions list by using synonyms.
Use Synonyms	Enable using synonyms in the suggestions list. For example, typing *"USA"* will result in *"United States of America"*.
Q-Gram	The n value of the q-grams for creating a q-gram inverted index, see Section 3.5.1

Table 5: Indexer configurations options. Fields with * are required.

While some of the indexing configurations in Table 5 are already explained within the table, others may benefit from further clarification. The **Words Weight 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 added and contribute to the overall query score. Adding more weights to some words can be beneficial as users find them more important than others.

The **Boosting Formula** feature can be used with inspector variables to influence the ranking process. For example, suppose one wants to rank products based on text relevance and factors like reviews or prices (which are numeric values rather than text). In that case, one can utilize the Boosting Formula. To do this, one can assign a variable name to an inspector, such as *review*. Then, in the Boosting Formula field,

one can insert a formula 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.

4.4.5 Search Engine Result Page (SERP)

All the indexed documents can be easily searched on the search page, consisting of three primary components. First, there is the search bar, which leverages the suggestions list dictionary configured during the indexing phase. The second component is a drop-down menu containing all the cached indexers that have previously been indexed. The last component is the result table, which uses a dynamic layout depending on the inspectors used for each document. For example, if the indexer indexes only the *title* inspector of a product, then the table will only contain one column with the inspector's name as a header.

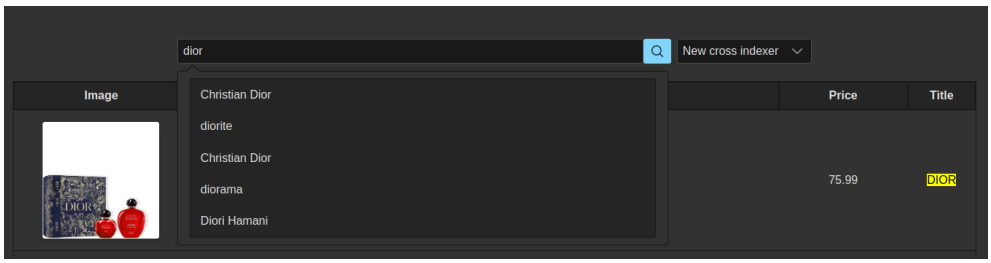


Figure 11: Scriburg Search Engine Result Page (SERP).

Table 11 shows the search result of products where we can note that it supports different data types like images. Note that the search result shows the 25 top matching results.

5 Evaluation

In this chapter, our primary goal is to assess and examine the implementation of the **Scriburg** search engine while also drawing a comparison to the **ParseHub** solution. The evaluation process is organized into three main sections. Within Section 5.2, we will conduct various experiments to assess the crawling process and discuss our findings and outcomes. In Section 5.3, our focus will shift to evaluating the indexing procedure. Finally, in the last section, Section 5.4, we will explore user experience and the pros and cons of current user interface design.

5.1 Testing Environment

Demonstrating information about the testing machine used for the evaluation can provide enhanced clarity and facilitate meaningful comparisons to reproduce similar results.

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

Table 6: The testing environment of the machine used in this evaluation.

5.2 Crawler

Evaluating web crawlers can be challenging, but certain aspects can be discussed and reflected upon to provide a more comprehensive understanding of the crawler's performance and effectiveness.

Coverage: This can be accomplished by maintaining a list of the URLs of the targeted website and verifying whether the crawler successfully located and

processed all of them. It is worth noting that precise measurement can be challenging because most websites do not disclose the total number of links they contain. Websites often undergo dynamic changes, resulting in some links being edited during the crawling process. Nevertheless, as previously mentioned, the evaluation metrics contribute to more comprehensively examining the crawler's performance and efficacy.

Scalability: How manageable it is to scale the computing performance vertically and horizontally, allowing users to crawl more and bigger websites.

Resilience: Assessing the crawler's ability to navigate complex situations and handle errors, including its performance with inaccessible pages, broken links, slow networks, and dynamic content.

Politeness: The ability to respect the `robots.txt` rules and avoid getting blacklisted from the website's servers.

5.2.1 Datasets

Evaluating a web crawler requires a more static website as a reliable reference point. Having a static website makes it easier to compare the content and the number of links. The **crawler-test**[1] website is an excellent choice for this purpose due to its diverse content and links, containing a wide range of scenarios that a crawler might encounter. This website effectively employs `robots.txt` to guide the crawler, allowing for an assessment of its politeness. Moreover, it includes a section containing links yielding various HTTP request status codes, such as **4xx** and **5xx**, which is 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 three 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 **Douglas**[2]. This website offers over 160,000 diverse products, making it an ideal candidate for testing different content types, including images.

[1] A website suitable for crawler testing: `https://crawler-test.com/`
[2] An e-commerce website: `https://www.douglas.de/`

The second website, **Times Higher Education**[3], specializes in annually ranking universities. Since this website ranking is often updated yearly, it is a great candidate to test the coverage as we can count the number of universities easily.

The third website is **Stack Overflow**[4]. Given this website's extensive volume of questions, our focus will be specifically on Python-related questions. We will also limit the number of documents to be crawled.

5.2.2 Experiments

Crawler Test Base Evaluation

As we mentioned, it is vital to test the crawler coverage, and the first website to test the crawler against is the *crawler-test* website.

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

Table 7: Crawler configuration for the crawler-test website

Table 7 displays the crawler testing configurations used in the experiment. The Seed URL is 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. 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

[3]Universities world ranking 2023: `https://www.timeshighereducation.com/world-university-rankings/2023/world-ranking`

[4]Stack Overflow Python-related posts: `https://stackoverflow.com/questions/tagged/python`

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.

Coverage Evaluation

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 8. Looking at the Links row, we can note that the crawler found 406 out of the expected 415 links. This is a coverage of 97.8%. The other nine missing links can be excluded due to different reasons. Normalizing the links can result in duplicated links that can be skipped. 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 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** 402	**Already Visited** 0	**Cross Site** 4	**Excluded** 1
Time	**Tot. Spent** 671.19 s		**Avg. Processing** 1.68 s	**Avg. Page Rendering** 0.697 s	
Status Codes	**1XX** 2	**2XX** 328	**3XX** 4	**4XX** 52	**5XX** 15
Docs & Content	**Tot. Docs** 255	**Duplicated Content** 24	**Avg. Docs Per Page** 1	**Avg. Page Size** 1.6 MB	

Table 8: Completed crawler result of the crawler-test.com web site

The *Docs & Content* section provides information regarding the collected and downloaded content. The *Tot. Docs* metric indicates that 255 documents have been

successfully downloaded. Twenty-five documents are duplicates and not saved in the database. This duplication is expected, 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.

Performance Evaluation

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. Furthermore, adding additional machines to enhance performance while not overburdening the server with excessive requests is possible, presenting a challenging balance between achieving excellent performance and maintaining proper politeness. However, some valuable matrices can be helpful to give a good insight like those shown in the Time row in Table 8. The total time spent to crawl **402 pages** took approximately **11 minutes**. To give a better perspective, the enterprise solution **ParseHub** [5] the free plan (without IP Rotation[6]), can crawl **200 pages** in **40 minutes**, and the Standard expensive plan (with IP Rotation) that costs $189 can crawl 200 pages in 10 minutes. This means crawling the **402 pages** inside the *crawler-test* will take **20 minutes**. Comparing the crawler with the ParseHub, it is two times faster than the standard plan and eight times faster than the free plan. Note that in this evaluation, the performance can be increased by using more threads or nodes to distribute the loads, which we will evaluate.

While some crawlers can achieve crawling rates of up to **300 pages/second** [Shkapenyuk and Suel, 2002], this comparison is not applicable. Crawling 300 pages from the same domain, as we are doing, is more challenging than simply crawling 300 different pages from 300 different websites. Distributing 300 HTTP requests evenly across these 300 websites is acceptable and does not result in a Denial of Service (DoS) issue because each domain receives only one request per second. However, in the context of Scriburg, users aim to target a specific set of websites, which could include just one website. If we send 300 requests to the same website, it could overload and crash the site.

[5] ParseHub plans: `https://www.parsehub.com/pricing`

[6] IP rotation is the practice of periodically changing the public IP address used by a device or server to improve security, avoid detection, or access geographically restricted content.

Rendering Dynamic Content Evaluation

Improving the average page rendering time (**0.679 seconds**) in Table 8 is achievable by avoiding using rendering engines like Selenium. Processing a simple HTTP request is often quicker than rendering the entire page and waiting for the page to finish loading. However, it is essential to note that the rendering step is crucial in handling dynamic content. For instance, consider the dynamically inserted text, indicated by the link [7]. This link is a straightforward example to illustrate the significance of the rendering process for web crawlers.

Using a simple `wget`[8] command in Linux, one can download the content shown in **??**. This content reveals that the HTML tags, `h1` and `p`, lack the inner text the crawler can collect as a document. Although a JavaScript code is designed to replace the `innerHTML` for each tag with text, without a rendering engine, the JavaScript logic 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.

```
<body>
  <h1 id="h1"></h1>
  <p id="par"></p>
  <script>
    document.getElementById("h1").innerHTML = 'Some random text';
    document.getElementById("par").innerHTML = 'Some long content ...';
  </script>
</body>
```

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

While this is a simplified example, it is easy to visualize 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, it is essential to discover all the HTML content.

```
<body>
  <h1 id="h1">
    'Some random text'
```

[7] Rendering test: `https://crawler-test.com/javascript/dynamically-inserted-text`

[8] wget is a command-line utility for retrieving files from the internet using HTTP, HTTPS, FTP, and FTPS protocols, primarily used in Unix-like operating systems.

```
  </h1>
  <p id="par">
    'Some long content ...'
  </p>
  <script>
    document.getElementById("h1").innerHTML = 'Some random text';
    document.getElementById("par").innerHTML = 'Some long content ...';
  </script>
</body>
```

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

Robustness and Politeness Evaluation

The *Status Codes* metric captured in Table 8 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. The crawler should remain operational even when encountering a status code other than **2xx**. In the specific test case, it is 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. In contrast, testing ParseHub to crawl the same links from the crawler-test website hangs at some links. One of the links that ParseHub stopped crawling and hanging was the redirect link[9].

The *Tot. Errors* metric accounts for various Selenium exceptions[10] 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 indication.

The presence of the `robots.txt` flag covers the commitment to polite crawling behavior. The flag is set to True when this file is located and successfully downloaded. It is essential to emphasize the importance of respectful crawling and commitment to the `robots.txt` file protocol. For monitoring purposes, the *Excluded* links in Table 8 record the count of links disallowed from being crawled. In this specific test case, there was only a single disallowed link.

[9] Redirect page test: `https://crawler-test.com/redirects/redirect_target`

[10] Selenium exceptions: `https://www.selenium.dev/selenium/docs/api/py/common/selenium.common.exceptions.html`

Changing Crawler Depth

Up to this point, our evaluation of the crawler has been limited to a single page to assess its simplicity and test its coverage. However, the crawler's functionality should extend beyond merely identifying links; it should also be able to navigate between them and continuously gather the target documents. The next step is to evaluate if the crawler jumps between pages and can increase the coverage. To evaluate this, we will change the *Max Depth* from 1 to 10. This will allow the crawler to jump up to 10 levels deeper, collecting more links and documents. Table 9 shows the configurations used for this test case. In practice, determining the precise depth can be challenging. In many cases, this number can be aligned with the pagination structure on the website, as the goal is to navigate through all pagination pages until they are completed. If the exact number is unknown, an approximate value can be assigned.

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

Table 9: Crawler configuration for the crawler-test website, increasing the depth to 10

Unfortunately, knowing exactly how many links the entire crawler-test site contains is challenging. However, since we have evaluated how the crawler behaves when the *Max Depth* equals one to test only the home page, we can have some degree of assurance on the rest of the results on the rest of the pages. The Table 10 shows 895 links have been found. This number was expected to increase compared to the first evaluation when the *Max Depth* equals one where the collected links were 406.

Given the increase in depth to 10, we anticipate more documents being collected. We would raise the maximum page limit and the maximum number of collected documents to allow the crawler to crawl more pages. One vital difference between

setting the *Max Depth* and *Max Pages* is that if one single page contains 1000 links, and the *Max Depth* is set to 1, then all the 1000 pages will be visited because all are living in the first level of crawling (home page with level equals to one) so limited by the depth only will not work. This is the reason for adding the *Max Page* as a termination condition.

Links	**Collected**	**Visited**	**Already Visited**	**Cross Site**	**Excluded**
	895	697	0	198	1
Time	**Tot. Spent**		**Avg. Processing**	**Avg. Page Rendering**	
	760.84 s		3.26 s	1.03 s	
Status Codes	**1XX**	**2XX**	**3XX**	**4XX**	**5XX**
	2	292	4	47	15
Docs & Content	**Tot. Docs**	**Duplicated Content**	**Avg. Docs Per Page**	**Avg. Page Size**	
	375	667	1	1.6 MB	

Table 10: Completed crawler result of the crawler-test.com web site when the depth is 10.

When comparing the increased depth test of the base crawler to the base one in the previous test, one notable metric is the significant increase in *Cross Site* links from 4 to 198. Typically, this suggests that the crawler is attempting to navigate beyond the originating website specified in the seed URL. When this number rises, it often indicates a need to adjust and refine the crawler settings. However, in the case of this test website, such an increase is expected and considered normal. This is because the website intentionally includes various tests designed to redirect the crawler to external websites. While search engines like Google benefit from crawling across different domains to index multiple websites, our specific crawler primarily focuses on a single website, as this aligns with the user's typical interest.

Multithreading Evaluation

It is crucial to assess the scalability of the crawler, primarily since it uses a unique links-sharing multithreading pool. We must thoroughly test it. We will use the same configurations detailed in Table 9 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 **17.6%** improvement, and 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 comparison to using **four** threads. This is primarily because all threads communicate to share unvisited links and avoid revisiting links already processed by other threads. Consequently, as the number of threads increases, the communication between threads and the resources sharing (database, logs, and buffers) overhead escalates. Users must fine-tune this parameter until they find the optimal setting, which appears to be **four** threads in the tested local environment.

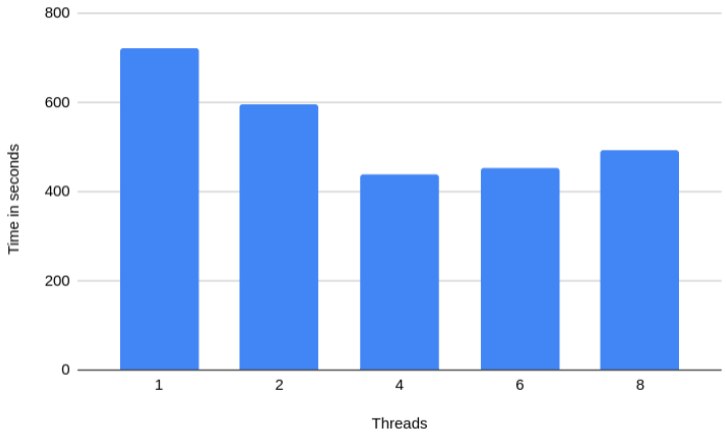


Figure 12: Multithreading performance comparison.

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 inactive, which would be inefficient, especially when using cloud services like AWS, where resources come at a cost. We will evaluate using the same configurations from Table 9, employing four threads and rerunning the same crawler four times.

To test the workload of each thread, we rerun a crawler with four threads four times and calculated each thread and how many documents each has collected. Then, the average of those four runs is also recorded. Figure 13 displays the results of four runs, with each run showcasing the distribution of crawled documents among each thread.

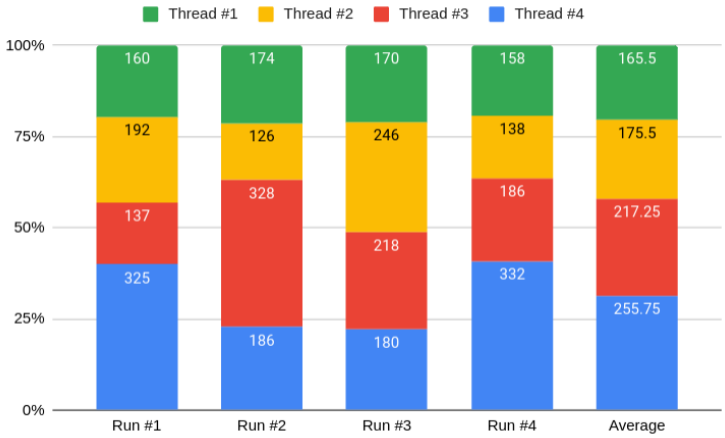


Figure 13: The workload distribution among four threads in four different runners. The chart shows each thread how many documents it has collected and the percentage.

While the ideal scenario would involve each thread downloading **25%** of the collected links, the chart shows otherwise. Looking at the *Average* column, we can note that thread number 4 crawls more than **25%**, while thread number **1** crawls less, which is expected because as long as one thread contains less than five links, it will keep crawling those links, and the other threads will remain idle. Only when the thread contains more than five links in the queue will it share those links with one other thread, and the others will remain idle.

For example, if a website contains only ten links, then only two threads will be needed, where each will process five links, and the other two threads will stay idle. We do not split all ten links among the threads equally because we want to reduce the thread's communication and resource-sharing overhead, and we only use a thread when needed.

Although each thread was not precisely processing **25%** of the workload, the result looks decent as each thread is slightly more or less than **25%** on average. The worst-case scenario is to run **four** threads; only one keeps running while the other three are idle; this wastes resources.

World University Rankings Evaluation

In order to assess the versatility of the web crawler and its adaptability to various usage scenarios, as we discussed, 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 11. 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.

Seed URL*	`https://www.timeshighereducation.com/world-university-rankings/2023/world-ranking`

Inspectors*	`//*[contains(@class, 'ranking-institution-title')]`
	`//*[contains(concat(' ', normalize-space(@class), ' '), ' location ')]`
	`//*[contains(@class, 'rank') and contains(@class, 'sorting_1') and contains(@class, 'sorting_2')]`

Allow Multi Elements	True
Max Pages	100
Threads	1
Max Depth	100
Pagination	`//*[contains(@class, 'pagination')]`
Actions	None
Max Docs	2350

Table 11: World University Rankings website crawler configuration.

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 have been encountered on the website. The total time spent during this process is approximately eight minutes, which is significantly shorter than 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 **403**[11] status code in response to all HTTP

[11]The HTTP 403 Forbidden status code signifies that the server comprehends the request but

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 ScrapeOps[12]. Cloudflare is a security service used by websites to prevent DoS attacks. In this particular use case, the number of collected documents representing universities in the table matches the results displayed in the page pagination, totaling **2,345**. It is worth noting that the website also deploys a `robots.txt` file, which the crawler successfully detected and used.

Links	**Collected** 94/94	**Visited** 94	**Already Visited** 0	**Cross Site** 0	**Excluded** 0
Time	**Tot. Spent** 351.66 s		**Avg. Processing** 2.57 s	**Avg. Page Rendering** 1.020 s	
Status Codes	**1XX** 0	**2XX** 0	**3XX** 0	**4XX** 94	**5XX** 0
Docs & Content	**Tot. Docs** 2345	**Duplicated Content** 0	**Avg. Docs Per Page** 25	**Avg. Page Size** 7.3 MB	

Table 12: World University Rankings crawler results

Douglas Evaluation

The following use case involves extracting a specific category of products from an e-commerce website. We will collect various data types in this scenario, including text and images. Table 13 shows the crawler configurations for this task, 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 Depth* parameters to be 10. We can employ multithreading to increase performance by setting the number of threads to four. Douglas employs lazy loading 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, which results in loading the rest of the content. The inspectors in this context encompass four fields: *brand*, *title*, *price*, and *product images*. The ability to configure automated actions depends on the website's functionality. For instance, if the website experiences lengthened loading times, we can include a waiting action to account for the estimated

declines to grant authorization for it.

[12] `https://scrapeops.io/web-scraping-playbook/403-forbidden-error-web-scraping/`

waiting time. If the website necessitates a clicking action to reveal more information, the click action can be utilized for that purpose.

Seed URL*	`https://www.douglas.de/de/c/parfum/damenduefte/duftsets/010111`

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 ')]`

Allow Multi Elements	True
Max Pages	10
Threads	4
Max Depth	10
Pagination	`//*[contains(@class, 'pagination')]`
Actions	Scrolling down 10 times
Max Docs	1000

Table 13: Douglas website crawler configuration

Table 14 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, which shows seven pages. 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 12. The primary reason for this extended processing time is the inclusion of additional scrolling-down actions.

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 (The End Key in Keyboard) 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. 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 **7 * 48 = 336 products**. Further investigation revealed that only some pages contain exactly 48 products; some have more, while others have fewer. This is why it

is advisable to limit the *Max Docs* in the crawler configurations to more than the anticipated number with a small margin.

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 **Avg. Page Rendering**), yielding **1.516 requests/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 **firewall**[13] rules and the reverse **proxy**[14] it uses.

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

Links	**Collected**	**Visited**	**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	**1XX**	**2XX**	**3XX**	**4XX**	**5XX**
	0	7	0	0	0
Docs & Content	**Tot. Docs**	**Duplicated Content**	**Avg. Docs Per Page**	**Avg. Page Size**	
	245	0	49	15.5 MB	

Table 14: Douglas crawler results

It was already checked that the `robots.txt` file has been found, downloaded, and used to filter the disallowed URLs in this use case.

ParsHub, on the other hand, 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.

[13] A 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.

[14] A 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.

Stack Overflow Evaluation

Another use case involved crawling Stack Overflow 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 ban after only ten pages were crawled. To resolve this, we reduced the thread count to one, reducing the number of requests and resolving the issue.

Seed URL*	`https://stackoverflow.com/questions/tagged/python`

Inspectors*	`//*[contains(@class, 's-post-summary-content-title')]`
	`//*[contains(@class, 's-post-summary-content-excerpt')]`
	`//*[contains(@class, 's-post-summary-stats-item__emphasized')]`

Allow Multi Elements	True
Max Pages	100
Threads	1
Max Depth	100
Pagination	`//*[contains(@class, 's-pagination')]`
Actions	None
Max Docs	1000

Table 15: Stack Overflow crawler configuration

After completing the runner, we noticed that 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 signaled that the crawler had lost track and needed to collect correct 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.

We have enabled the *Show Browser* option to troubleshoot and rerun the crawler. This allowed for easier visualization of the crawler's behavior 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.

After fixing the second issue (pagination selector) and rerunning the crawler, it operated correctly and yielded results in Table 16. **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, totaling **900**, with nine links collected per page. Interestingly, there were a significant number of **429**[15] status codes. To address this, it could be beneficial to include a wait action between requests to mitigate the **429** errors.

Links	**Collected**	**Visited**	**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	**1XX**	**2XX**	**3XX**	**4XX**	**5XX**
	0	54	0	46	15
Docs & Content	**Tot. Docs**	**Duplicated Content**	**Avg. Docs Per Page**	**Avg. Page Size**	
	2750	0	50	3.4 MB	

Table 16: Stack Overflow crawler results

It was already checked that the `robots.txt` file has been found, downloaded, and used to filter the disallowed URLs in this use case.

When the same test was conducted using **ParseHub**, it took **20 minutes** to complete, which was slower than our crawler's **6 minutes** 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. 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.

[15]The HTTP 429 status code suggests the crawler has sent too many requests.

5.3 Indexer

Following the 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 will also illustrate how easy and configurable the indexer parameters are and how they affect the evaluation score.

5.3.1 Datasets

We selected the **Stack Overflow dataset**[16] presented in Table 17 out of the three available use cases we have used in the crawling evaluation. 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	1.4 MB
Entries Count	2,415
Words Count	108,122
Fields	Title, Summary, Votes

Table 17: Stack Overflow posts dataset

As discussed, we will also support adding a dictionary for the autocomplete queries feature. The dictionary can be added under the directory `/dictionaries`, and during the evaluation, we will use a **Wikidata dataset** as shown in the Table 18.

File Size	437,41 MB
Entries Count	2,642,529
Words Count	30,309,063

Table 18: Wikidata dictionary dataset

In order to evaluate the indexing process, we need a benchmark that can be used for testing. We have created a small benchmark[17] made of 6 queries, which will be used to evaluate all the following indexing tests.

[16] Stack Overflow dataset: `https://github.com/Alhajras/webscraper/blob/main/datasets/stack_overflow_posts_dataset.csv`

[17] Stack Overflow benchmark: `https://github.com/Alhajras/webscraper/blob/main/datasets/benchmark.txt`

5.3.2 Metrics

We assess precision at a given value $\boldsymbol{k}$ ($\boldsymbol{P@k}$), and calculate the average precision ($\boldsymbol{AP}$).

Precision at k (P@k)

$\boldsymbol{P@k}$ represents the proportion of valid predictions within the system's top $\boldsymbol{k}$ predictions. We define $\boldsymbol{Q_{valid}(q)}$ as the collection of valid predictions for a user query $\boldsymbol{q}$, as specified in the ground truth. Additionally, we denote $\boldsymbol{Q^k_{result}(q)}$ as the set of the system's top $\boldsymbol{k}$ completion predictions for a given user query $\boldsymbol{q}$. The calculation for $\boldsymbol{P@k}$ is as follows:

$$P@k = \frac{|Q_{valid}(q) \cap Q^k_{result}(q)|}{k} \tag{5.1}$$

We will compute the precision at **5** ($\boldsymbol{p@5}$) for all the various indexing configurations.

Average Precision (AP)

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

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

For the predictions from $\boldsymbol{Q_{valid}}$ that are absent in $\boldsymbol{Q_{result}}$, we assign a Precision at position $\boldsymbol{r_i}$ ($\boldsymbol{P@r_i}$) value of 0. We then calculate the average precision by averaging these values across all queries 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.

$\boldsymbol{MP@k}$ represents the mean precision at $\boldsymbol{k}$ values across all queries, $\boldsymbol{MP@R}$ represents the mean precision at $\boldsymbol{R}$ values across all queries, and $\boldsymbol{MAP}$ signifies the mean average precision values across all queries.

5.3.3 Experiments

Base Stack Overflow Evaluation

We will initiate our indexing process using the default settings specified in Table 19. 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 5, 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 19: 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 the **Wikidata dictionary**, which aids in providing suggestions in the drop-down 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 difference between these stages lies in the size of the entities involved; the Wikidata 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 19. 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. Moreover, there is no solid threshold for the indexing metrics to be classified as good. However, in our evaluation, any value above **0.5** will be considered acceptable.

Furthermore, aiming for an extremely high level of accuracy can lead to model **overfitting**, which is when a model fits its training data too closely, leading to poor performance when dealing with new, unseen data, 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 at 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 $P@k$ scores for the same model. As a result, it is essential to carefully and consistently choose these parameters when comparing different models.

It can be noted that those default values already show a decent result without any fine-tuning. This indicates that the default values provide a decent result to non-technical users. Our returned result takes, on average, around **0.31 seconds**, which is too slow in comparison to the Google search engine, which suggests the speed of processing a query should be less than **0.1** [Hoelzle, 2012].

BM25 Parameters Effect

It is essential to allow users to quickly fine-tune their indexer since the b and k values of the **BM25** formula can be edited from the UI. The values of both parameters b and k depend highly on the datasets, and there are no magic numbers that work for all models. Table 20 displays the configuration of the Stack Overflow indexer, with modifications made to the default b and k values to have higher accuracy than the basic configurations shown in the previous Table 19. Changing the b and k

parameters resulted in the increase of the ***MP@5*** by **8.75%**, ***MP@R*** by **14.2%**, and ***MAP*** by **2.2%**. 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	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 20: Stack Overflow indexing configuration, the effect of changing BM25 parameters

Stop Words Effect

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 that are already used in the benchmark, 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 reduces accuracy, as illustrated in Table 21. In comparison to the previous Table 20 (without using stop words), the ***MP@5*** was reduced by **17.5%**, ***MP@R*** reduced by **25.7%**, and reduced ***MAP*** by **14.6%**

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

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** 0.66	**MP@R** 0.624	**MAP** 0.76	**Time(s)** 0.31

Table 21: Stack Overflow indexing configuration, the effect of changing *Stop Words*

impact, as all the posts are Python-related, even if they do not explicitly mention the word *"python"* but discuss Python libraries, for instance. There are already available lists[18] of stop words for each language that can be used; however, as we have discovered, it is also essential to analyze the common words in the dataset to be indexed as well.

It is important to note that stop words used that contain two characters or fewer (*"to," "by,"* and *"in"*)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.

Small Words Threshold Effect

In the following evaluation in Table 22, 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 compared to the best result found in Table 20, there has been a decrease in precision. This suggests that in some instances, increasing the *Small Words Threshold* and not setting it to zero may be advisable. However, it is essential to exercise caution when eliminating small words, particularly those with one to three characters, depending on the language. Additionally, it is worth noting that some two-character words can be acronyms, such as ***"OS"*** (***Operating System***).

A better refinement can be done here; instead of having two different parameters,

[18]Stop words lists: `https://www.ranks.nl/stopwords`

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 22: Stack Overflow indexing configuration, the effect of reducing the *Small Words Threshold* attribute

Stop Words and *Small Words Threshold*, we could keep the *Stop Words* list, and instead of saving them as words, we can save them as a regular expression. Using regular expressions will also have the advantage of excluding patterns. For example, instead of excluding all *Python* versions like *Python 3.10.0* and *Python 3.10.1*, one can add a stop word rule to remove all of them instead of adding them one by one. It is worth noting that there is a workaround in the current implementation, which is to use the *Clean-up Expression List* in inspectors form shown in Table 3. This will remove the words that match the given pattern instead of skipping them in the indexing process.

Boosting Formula Effect

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 5.2.2.

In the specific context of the Stack Overflow example, each post is associated with an **up-vote** number, which indicates how helpful an answer is. This numeric field can be a valuable indicator of document quality and relevancy. The more users find a post helpful, the more valuable it becomes to display 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, more suitable options may exist since votes can be zero or even negative. We will

utilize the formula shown in Table 23 for this straightforward use case.

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 23: Stack Overflow indexing configuration, the effect of using the *Boosting Formula*

After boosting the score for each post by the number of up-votes, we can note that the $\boldsymbol{MP@5}$ was reduced by **33.75%** $\boldsymbol{MP@R}$ was reduced by **26.2%**, and $\boldsymbol{MAP}$ by **29.21%**. This is because some posts have negative votes (down-votes), and some have positive, resulting in significant differences in the score generated by the BM25 formula. For example, if the query is *"how to import panda library in python"* and the returned results contain one matching post but have fewer votes than a second post that only mentions *"python"* but has higher up-votes than the first post, which is less relevant will be at the top.

5.4 User Experience

There are notable advantages and disadvantages when comparing **ParseHub** with **Scriburg**.

One of the standout features of ParseHub is its automatic detection of document fields, achieved by clicking. ParseHub opens a live browser session, allowing the user to click on any field that wants to be collected. In contrast, Scriburg requires manual input of HTML element XPath to obtain this information, which takes much work. Creating the crawlers manually 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 instead of Scriburg lists view. 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 adding 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. The monitoring used by Scriburg made it easy to debug the crawlers and stop them before wasting resources.

When crawling a large dataset, having an indexed version becomes crucial, mainly if the dataset is intended to be served as an API, for example, which Scriburg supports but not ParseHub.

6 Conclusions and Future Work

Within this chapter, we address the questions initially introduced at the opening of the thesis (Section 1.3), conclude our experiments, and open the door for additional research to enhance the solution and explore areas that have not been previously examined.

6.1 Conclusions

It is crucial to remember the primary questions and contributions intended to be accomplished by this thesis and whether we have successfully addressed them or if they remain abstract. In the following list, we iterate through the contributions mentioned in Section 1.3 and discuss our final findings.

What are the challenges and bottlenecks to creating a scalable, configurable search engine? Two primary bottlenecks restrict the crawler's performance. The first is the loading time, or the time it takes to render a page, which varies for each page and ultimately affects the rate at which the crawler can make requests. The second challenge is that websites can block the crawler if the request rate increases, which is also contingent on the configurations of the websites' firewalls.

Can we outperform a similar tool like ParseHub? ParseHub demonstrated an advantage with its smoother workflow and improved automated field selection. Nonetheless, Scriburg surpassed ParseHub in terms of performance and overall robustness.

How does changing the configurations provided by the user interface affect the results in the crawling and indexing accuracy? The user interface offers forms, including optional and advanced crawling and indexing options. The used options deliver a helpful and adaptable user experience. Even users with limited programming experience can easily configure and execute straightforward crawling and indexing tasks by relying on the default values provided.

Can we create a decent User Interface UI that intuitively allows users to crawl and index targeted websites from the internet? Yes, we did. However, manually adding and removing inspectors was time-consuming and should be refined.

How well do crawlers react to different websites with different DOM structures? While the internet hosts millions of websites, making it impossible to ensure coverage for all possible use cases, we have evaluated many websites and effectively managed their diverse implementations.

Can we integrate the indexing and crawling processes in the same tool? It was possible to provide a user-friendly interface that combines both functionalities.

Can we find meaningful evaluation metrics for the implemented search engine? We have discovered that evaluating crawling can pose challenges, but certain aspects can be addressed to provide insights into the crawler's performance.

6.2 Future Work

While evaluating **Scriburg**, the most wanted feature was to automate the inspector's selections. The solution should be similar to what **ParseHub** is implementing. The user should be faced with a live session where they can click on any title or price they want to collect simply by clicking. However, This feature is not easy to implement and can take up to **three months** to perfect.

Adding **IP Rotation** is highly recommended because although the crawling rate was not high in the evaluation, the crawler got banned twice from different websites. Implementing IP Rotation is relatively easy. Some services provide free proxy list API[1] to be used to make proxied requests. One can add each proxy bypassing the flag `proxy-server` in Selenium. This feature can take up to **three days** to implement and test.

Adding a **Steps**[2] component can allow the users to crawl and index step by step. For example, the first step is to create a project name like Stack Overflow. This name will be used for all templates, runners, crawlers, and indexers instead of entering the name in each form. The steps will make running a crawler intuitive and reduce user confusion. Using the **Steps** component from **PrimeNg** will take **two weeks** to implement and add this functionality. This feature can solve the issue by making the user interface more intuitive and easily used.

[1] SSL Proxies: `https://www.sslproxies.org/`
[2] PrimeNG Steps: `https://primeng.org/steps/personal`

Early stopping the crawlers once they are inefficient helps save resources. This can be due to an internet connection; the website is down, the crawler needs to be better configured, and more issues can make crawling inefficient. Although we are serving valuable information to help monitor the crawling process, for non-technical users, it can be hard to understand what the HTTP status codes stand for. To fix this, we can set up some rules to convert errors into valuable messages that users can understand. This feature can take between **two weeks** to **one month** to implement.

Leveraging the `robots.txt` file to direct web crawlers can be effective, but there is a need for further research into establishing a robust protocol for crawler behavior. For instance, it proves beneficial for crawlers to gain insights into website characteristics, such as estimating the number of links, products, or items present. This information helps users in determining the appropriate scaling of their crawlers. Additionally, the file can specify the maximum allowable requests per second to prevent potential denial-of-service (DoS) issues. It could also contain preferences regarding crawl timings, such as restricting activity to nighttime.

7 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.
My advisor Natalie Prange for her competent opinions and suggestions.

Bibliography

[Bast, 2023] Bast, H. (2023). Information retrieval course: (inverted index, zipf's law), (ranking and evaluation) and (fuzzy search, prefix edit distance, q-gram index).

[Brin and Page, 1998] Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117.

[Castillo, 2005] Castillo, C. (2005). Effective web crawling. *SIGIR Forum*, 39(1):55–56.

[Cormen et al., 2001] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. The MIT Press, 2nd edition.

[Fetterly et al., 2003] Fetterly, D., Manasse, M., and Najork, M. (2003). On the evolution of clusters of near-duplicate web pages. In *Proceedings of the IEEE/LEOS 3rd International Conference on Numerical Simulation of Semiconductor Optoelectronic Devices (IEEE Cat. No. 03EX726)*, pages 37–45. IEEE.

[geeksforgeeks, 2023a] geeksforgeeks (2023a). Breadth first search or bfs for a graph.

[geeksforgeeks, 2023b] geeksforgeeks (2023b). Depth first search or dfs for a graph.

[Hoelzle, 2012] Hoelzle, U. (2012). The google gospel of speed.

[Kumar et al., 2017] Kumar, M., Bhatia, R., and Rattan, D. (2017). A survey of web crawlers for information retrieval. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(6):e1218.

[Manning et al., 2008] Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

[Pinkerton, 2000] Pinkerton, B. (2000). *Webcrawler: Finding what people want*. University of Washington.

[Schütze, 2014] Schütze, H. (2014). Introduction to information retrieval: Web search.

[Shkapenyuk and Suel, 2002] Shkapenyuk, V. and Suel, T. (2002). Design and implementation of a high-performance distributed web crawler. In *Proceedings 18th International Conference on Data Engineering*, pages 357–368. IEEE.

[Statista, 2020] Statista (2020). Average number of search terms for online search queries in the united states as of january 2020.

[Statista, 2023] Statista (2023). Market share of leading desktop search engines worldwide from january 2015 to july 2023.