Master's Thesis

Design and Implementation of a Configurable Preferential Search Engine using Scalable Crawlers

Alhajras Algdairy

Examiner: Prof. Dr. Hannah Bast
Advisers: M. Sc. Natalie Prange

Albert-Ludwigs-University Freiburg

Faculty of Engineering

Department of Computer Science

Chair of Algorithms and Data Structures

April 25th, 2021

Writing Period

01. 12. 2020 – 25. 04. 2021

Examiner

Prof. Dr. Hannah Bast

Second Examiner

Prof. Dr. Thomas Brox

Advisers

M. Sc. Natalie Prange

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date

Signature

Abstract

Web search indexing is an essential system that powers modern search engines. It automates the process of collection and organization of data from web pages to create an updated index of the web that can be optimally searched. Web search indexing consists of two essential components, a web crawler, in which search engine bots systematically traverse the web to find new or updated content based on rules declared beforehand, followed by the second component which is the indexing of the collected data. The process of web search indexing comes with its own challenges, including performance, managing dynamic content, and answering the question of what is the most relevant content. As the web continues to evolve and grow, the task of web search indexing will remain a key focus of search engine technology and research.The aim of this thesis is to design and implement a generic configurable web search indexing that can be used as a basic tool on different websites and can be further expanded and improved, and scaled. The approach included a simple UI design that allows users to configure and create crawlers and index the generated data.

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 guidance and supervision.
- M. Sc. Natalie Prange for her thoughtful ideas and suggestions.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Since the beginning of the Digital Revolution, known as the Third Industrial Revolution, in the latter half of the 20th century, the importance of data has increased as it became the new currency shaping the dynamics of our interconnected world. From social media platforms and e-commerce transactions to information sharing and entertainment consumption, online activities generate enormous amounts of data. The online data is sometimes referred to as the 'new oil' or the 'new currency', as it impacts almost the same economies and societies as oil. Businesses and organizations understand the power of data as they provide insight into consumer behaviour, refines 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) 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.

The 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 information from websites.

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 information from websites. The applications and use cases of internet crawlers are diverse and valuable, to name a few:

- Search Engines: Crawlers are essential components to build any search engine, such as Google, Bing, and Yahoo. Crawlers are run on supercomputers to crawl all the content on the internet index web pages and gather information about content, keywords, and links. This data is then used to rank and display search results, ensuring users can quickly find relevant information.

- 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.

- Fraud Detection: Cybersecurity companies use crawlers to catch fraudulent activities by monitoring online transactions, identifying unusual patterns, and tracking potential threats.

- Content Monitoring: E-commerce platforms utilize crawlers to extract product prices from various websites. This enables them to offer consumers real-time price comparisons and assist in finding the best deals. Moreover, social media platforms use crawlers to monitor their content to prevent unwanted posts and images.

1.2 Problem Statement

The World Wide Web (WWW) contains enormous data that escalates with each passing day. The total data created and replicated is expected to grow to more than 180 zettabytes by 2025 according to Statista. This upward trajectory is expected to continue due to the growing affordability of smartphones and the broadening reach of internet accessibility. Moreover, due to the COVID-19 pandemic, more companies started offering remote work, more local shops transformed into online stores, and more services switched to cloud-based. This social evolution over recent years has embedded the Internet as an integral cornerstone of our daily life.

The expansion of the Internet gives rise to an immense overflow of data, resulting in a noise that complicates the task of locating relevant information for both end users and organizational queues. To surmount this hurdle, the concept of Information Retrieval (IR) was coined by Calvin Mooers in 1951. IR involves the art of accessing and recovering data from an extensive pool of unstructured information. A particularly pragmatic manifestation of IR involves the extraction of data from the Internet, thus advocating the implementation of a universal algorithm for procuring and categorizing requisite information. In this pursuit, crawlers or spiders emerge as automated entities designed to adhere to predefined directives, allowing the automated fetching and extracting of data from the Internet.

One form of IR is a web search engine. A web search engine is a system engineered to index the Internet. Users can search for articles, documents and pages by entering keywords. The search will provide a list of the most related result that matches the search query. Using the crawlers explained earlier; the engine can index the collected information and optimize the search process using different algorithms and techniques.

Although search engines such as Google, Bing and DuckDuckGo display remarkable proficiency in their web crawling and indexing capabilities, specific businesses, like those in E-commerce, have a distinct interest in demonstrating the most competitive pricing for a given product, a key insight into their competitive landscape. However, more than a straightforward Google search is needed, as the search query index and rank the documents on the Internet based on Google's vendor parameters. These parameters include preeminent brand visibility, user geolocation, SEO optimization proficiency, and hidden variables, excluding the lowest price criterion from the page ranking equation. A second issue arises from the format of search results, with each search engine providing a distinctive result format. Companies may want to exclude some portion of the Internet from the index and rank. Also, they should prioritize some pages more than others.

The previous requirements are tiny use case, among others, that limits companies from using a simple search on Google. Search engines need to be tweaked and configured to match the domain of interest as E-commerce in the previous example and to match a specific use case like the price comparison mentioned.

The main concern is that businesses are often interested in only a portion of the Internet that intersects with their domain and expertise. Furthermore, the criteria for indexing and page ranking depend heavily on their use case and is vital to their business to take control of it and configure it as fit. Hiring domain expertise is inevitable for any business. However, the data scientists often have to go throw some basics steps to get their crawler up and running; those steps cost money and time; it would be helpful to have an infrastructure that allows the data scientist to have starting script that can be extended easily and needs little to no programming knowledge.

Amount of data created, consumed, and stored 2010-2020, with forecasts to 2025 Published by Petroc Taylor , Sep 8, 2022 https://www.statista.com/statistics/871513/worldwide-data-created/: :text=The

1.3 Contribution

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

- What are the challenges and bottlenecks to creating a scalable, configurable generic search engine?
- Can we implement a basic tool that can be easily scaled to crawl different websites independently from their DOM structure?
- Can we create a proper User Interface UI that allows users to crawl and index targeted websites from the internet?
- 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

2 Related Work

Creating a generic configurable search engine that includes a simple user interface will require research on how the current search engine works and what are the existing commercial or open-source solutions that offer a similar feature that allows the search engine to be configured via user interface reactions. This chapter will explain an overview of the Google search engine architecture since the basic architecture concepts will be reused with some modifications in this thesis. Then a list of the existing solutions used to crawl the web will be discussed.

2.1 High Level Google Architecture

The Google search engine's design gives a good overview of the essential components to create a scalable search engine. Hence, it is a great starting point for any search engine research; we will explain it in this section. Most code written in the Google search engine was implemented in C or C++ for efficiency and because it can run on either Solaris or Linux [2]. Google uses distributed crawlers to download internet web pages. The URLserver keeps a list of the available found URLs that need to be crawled by the crawlers. URLserver acts as a load balancer that sends the URLs to the following free crawler. Afterwards, the crawlers download the documents needed from the page, associate a unique ID for this page called docID, and then the page's content are stored in Soreservers. Storeservers then compress the pages and save them on a repository. The indexer component then uncompresses the pages and parses them. Each document is then converted into a set of words called hits. The hits represent the word and its position in the document. Afterwards, the indexer distributes those hits into barrels. Moreover, the indexer collects links found in the crawled page and stores them in the anchor's file. The anchors' file contains the links and their relationship with each other [2].

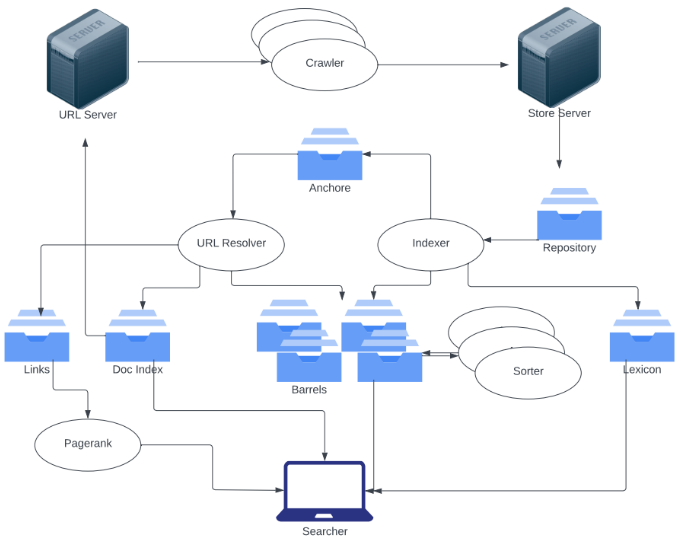


Figure 1: High level view of Google web crawlers archeticture.

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

Initially organized by docID, the barrels are then rearranged by the sorter based on wordID. This process generates an inverted index. Moreover, the sorter generates a list of wordIDs and corresponding offsets within the inverted index.

2.2 Web Crawlers

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

WebCrawler, created by Brian Pinkerton in 1994, is considered the first true web crawler-powered search engine. While some may claim that title for Wandex is due to its potential, it was never designed to be used in this way. Wandex lacked some critical features to make it a general-purpose search engine.

One of the major innovations of WebCrawler was its full-text searchability. This ability made it popular and highly functional. It continues to operate as a search engine, although not as popular as Google, Yahoo, Bing, Yandex, or Baidu.

Modern web crawlers face many challenges and complexities, such as dynamic content, user interaction, authentication, robots.txt files, and ethical issues. Some examples of modern web crawlers are Googlebot, Bingbot, and Internet Archive

Web crawlers have evolved during the last few decades, with different designs and implementations to crawl and index the internet. Below is an enumeration of some of the architectural designs utilized in the development of all-encompassing web crawlers [3]:

- **RBSE [Eic94]** Considered as one of the first Web crawlers to be published. Made of two components: the first component, "spider", uses a queue in a database. The second component, "mite", is a modified browser that downloads the pages from the Web.

- **WebCrawler [Pin94]** The initial publicly accessible full-text index of a specific portion of the World Wide Web was established. The approach involved leveraging lib-WWW for page downloads and employing an additional tool to parse and arrange URLs, ensuring a breadth-first approach to navigating the web graph.

- **World Wide Web Worm [McB94]** was a crawler designed to construct a basic index comprising document titles and corresponding URLs. This index could be queried by utilising the grep command in the UNIX operating system.

- **Google Crawler [BP98]** Google has been the market's dominant search engine for the last few decades. In March 2023, Google's global market share was 85.53The crawler was integrated with the indexing, and since this thesis has some similarity with the Google search engine design, we will explain this in-depth in the ext subsection [Google architecture]

- **Ubicrawler [BCSV04]** Is a Java-based distributed crawler with no central process and several identical "agents". The crawler is implemented to provide high scalability and be tolerant of failures.

Although the previously mentioned crawlers offer a wide range of features and are great to be used as generic crawlers to fetch all web pages, they need to provide a simple user interface to configure them based on user needs. This is what this thesis tries to tackle and investigate.

Nowadays, data scientist uses different tools to crawl and parse internet content. Each tool has its pros and cons and serves a different use case than the other. The following list goes through some of the most well know crawlers and explains how the proposed solution in this thesis differs.

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

- **Scrapy:** It is an open-source, powerful and flexible tool that easily crawls and parses different websites. It allows the creation of custom spiders to crawl multiple pages. Easy to scale makes it suitable for large projects. This tool is perfect for programmers but not for non-technical users, as it requires good knowledge of Python programming.

- **Selenium:** It is an open-source, robust and adaptable solution for web scraping, enabling the automation of browser actions, interaction with web pages, and data extraction from online sources. It shares some features with the Beautiful Soupe as it is an excellent tool for parsing the HTML DOM. Still, it also overcomes the issue previously mentioned about rendering Javascript and supporting dynamic contents as paginations. Interactive browser automation makes it easy to mimic the user's behaviour which makes it easier to navigate towards hidden content that requires events and human interactions. Selenium alone can be used as a search engine; however, it will be used in this thesis as a fundamental tool for the search engine implemented.

- **ParseHub:** A web crawler tool with a friendly User Interface requiring no programming skills. It is one of the top choices of most data scientists. The massive advantage of ParseHub is the point-and-click interface provided. It makes data extraction extremely easy. ParseHub offers both free and paid plans. The free plan allows users to scrape up to 200 pages per run, which, as we will see, is too slow for a crawler. Moreover, it is not possible to configure the crawling algorithms with this tool, and the indexing component does not exist. Since this tool is the most similar tool to the solution implemented by this thesis, it will be used as a comparison in the evaluation chapter.

3 Background

This section tackles the fundamental principles and groundwork of the theory encompassing concepts, terminology, and methodologies related to search engines as applied within this thesis.

3.1 The nature of the web

The web we know today, Web 2.0, is known as "the participative social web" and is massive, and its rate of change is enormous due to its highly dynamic content. Due to this big sample space, finding the relevant pages or documents from the web isn't easy. To overcome this issue, there are two main approaches to sampling: Vertical sampling: Focus only on the pages that are restricted by the domain name. This can be done in different levels. For example, one can restrict the crawling process based on the country, such as .de, .ly, uk. When vertical sampling is done at the second level, it limits the crawling to domains (e.g. stanford.edu) [3]. Horizontal sampling: In this approach, it is important to know the most visited pages and domains to keep crawling from them, giving them more priority than others. This can be done by collecting data logs from the ISP or utilising a web crawler to estimate the page's popularity [3].

3.2 History

The World Wide Web is an unlimited space to share provide and share information. Those information can have different format and cover different doamins. The use case of the web is only limtied by the developers imagination. This is benifital as the Web keept evolving rapidaly form Web 1.0 to Web 2.0 to Web 3.0. Web 1.0 used static pages to serve information, those information were moslty news, blogs and personal langing pages. Some refre to the Web 1.0 as "the read-only web". Although Web 1.0 was massive however most content were created was by deverlopers or at least users who knew basics of the HTML and CSS, moreover by that time content were only static they did not depen on fancy JavaScript libraries and frameworks like Angular and React, this made it limited to some use cases only. Fast forward, pages become more dynamic after using sessions, databases and clint rendering schemas. Those changes made the Web focused not only reading and gathering information by gave the power to more audiounce who did not know any programming or coding to participate and interact with the Web via browsers. Social media, e-commerce and trading stocks platforms was one of the reasons made the internet buble inflate, Use cases where unlimited as useres could create and deploy their own websites by using simple tools as Content Manament System CMS. This made Web 2.0 known as "the participative social web".

To optimize the allocation of crawler resources, estimating the page's freshness must be considered. This prevents outdated pages from remaining unrefreshed for prolonged periods or where lesser-significant pages are needlessly recrawled despite unchanged content.

It can be understood intuitively that the likelihood of a copy of page p being up-to-date at time t, denoted as up(t), declines over time when the page is not revisited.

$$u_p(t) \propto e^{-\lambda_p t} \tag{3.1}$$

The parameter p signifies the rate of modifications occurring on page p, and its estimation can be deduced from past observations, mainly when the web server indicates the page's last modification date during visits. Cho and Garcia-Molina derived this estimation technique for p [8].

$$\lambda_p \approx \frac{(X_p - 1) - \frac{X_p}{N_p \log(1 - X_p/N_p)}}{S_p T} \tag{3.2}$$

- Np number of visits to p.
- Sp time since the first visit to p.
- Xp number of times the server has informed that the page has changed.
- Tp total time with no modification, according to the server, summed over all the visits.

Note that some pages do not include the last-modified time stamp, and in this case, one can estimate this manually by comparing the downloaded copies at two different times and using the following equation. Where Xp now will be the number of times a modification is detected.

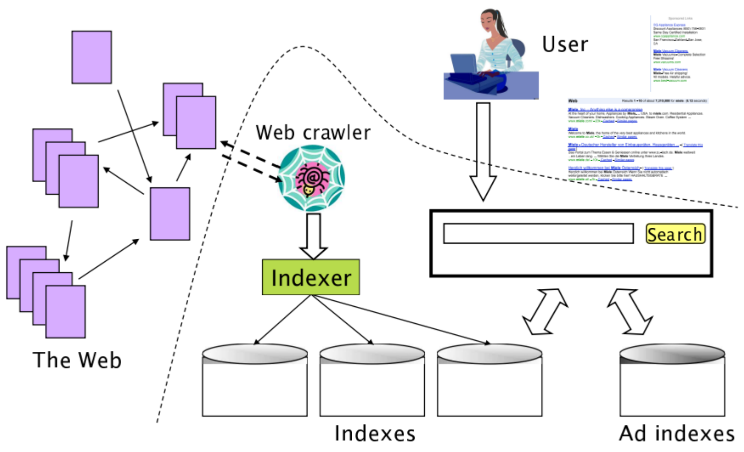
$$\lambda_p \approx \frac{-N_p \log(1 - X_p/N_p)}{S_p} \tag{3.3}$$

3.3 Web Search Engine

Web Search Engine is software that collects information from the web and indexes them efficiently to optimize the searching process by the user. Users enter their queries to ask for information. The engine performs queries, looks up the pre-built organized index, and returns relevant results. The returned result is presented by

Search Engine Results Pages as known as SERPs. The result is then ranked based on predefined criteria.

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



3.3.1 Requierments and Features

Search engines, regardless of their imp[lmentation and design there, are some features and requirements that make a good one; following is a list of the most fundamentals features:

- Web Crawling and Indexing: Each search engine needs two main big components, Crawler and Indexer. The Crawler is the component responsible for collecting pages and downloading them from the web. An indexer is used to create an index to facilitate efficient searching.

- Ranking and Relevancy: The algorithm determines the order in which search results are presented to users based on relevance.

- User Interface: The user interface where users enter their search queries and view results.

- Scalability and Performance: Distributed Architecture: A distributed system helps handle the vast amount of data and traffic. This needs a Load balancer to distribute the cralwing tasks betrween the nodes and threads.

- Data Storage and Management: A robust database system is necessary for storing indexed data and metadata.

3.4 Cralwer

Web crawler or spider is a software which gathers pages information from the web, to prived the necessary data to the Indexer to build a search engine. The essential role of crawlers is to effectively and reliably collect as much information from the web. This thesis invests more time on this component than the Indexer as it serves as the bottleneck to the Search engine performance.

3.4.1 Cralwer Specifications

Crawlers can have a wide vireity of features and specifications, however some are necessary to include and others are vital to have a reliable useable one. More information can be found in the book [9]

- Robustness: Web crawlers can be fragile and easy to break; this is due to the nature of the dynamic contents on the web and the internet connection. Web crawlers must identify those edge cases and obstacles and tackle them.

- Politeness: The implementation of the crawler can be unintentionally Mellitus and dangerous if not designed correctly. A Denial of service DoS and a Distributed Denial of service DDoS attacks can occur due to a bad crawler implementation. Hence crawlers must respect websites policies and avoid breaking up web services and loading the servers.

- Distributed: For optimal efficiency, the crawler should possess the capacity to operate in a distributed manner across numerous machines.

- Scalable: The crawler's design should enable the expansion of the crawl rate by seamlessly integrating additional machines and bandwidth.

- Performance and efficiency: The crawling system should adeptly utilize various system resources, including processor capacity, storage capabilities, and network bandwidth.

- Quality: The crawler should be biased towards fetching "useful" pages first.

- Freshness: Acquiring recent versions of previously accessed pages. This is particularly relevant for search engine crawlers, ensuring the search index remains updated.

- Extensible: Crawler design should possess extensibility across numerous dimensions, facilitating adaptation to novel data formats, emerging fetch protocols, and similar challenges. This necessitates a modular architecture that accommodates expansion.

3.4.2 Crawler architecture

The simple crawler architecture is made of the following fundamental modules, as shown in the following Figure. Fetch module that communicates with the internet and collects the pages passed by the URL Frontier module using HTTP requests from the URLs. 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. Duplication filter that is used to exclude seen URLs. The DNS resolution module is responsible for identifying the web server from which to retrieve the page indicated by a given URL [9].

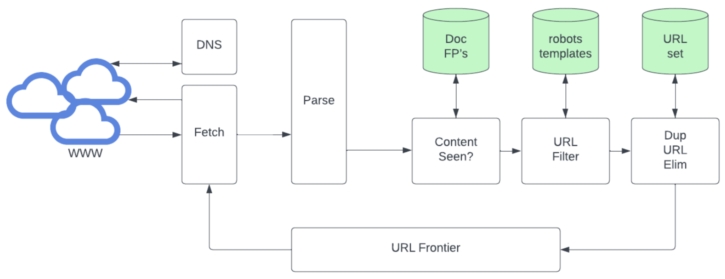


Figure 2: The basic crawler architecture.

The first step is to add a seed URL to the URL frontier. This URL works as a starting point for crawling. The crawler then fetches the page corresponding to the seed URL and stores it to be parsed by the parser. Subsequently, the page undergoes parsing to extract both its textual content and embedded links. The content will be used by the indexer component in the search engine. Moreover, each identified link by the parser module is subjected to a set of evaluations to determine its eligibility for addition to the URL frontier.

After finding the future links and content by the parser, filtering both found links and content is needed. The first step is to check if the page content has already been seen; this can be done by checking the page content fingerprint. The most straightforward method is to use a checksum (stored in the Doc FP's). The next filter is to exclude the parsed new URLs. The URL filter will run some tests to exclude unwanted URLs. This can be some URLs out of the country target, like .de, or some restricted URLs that should not be visited by the crawler. Excluded

URLs list can be added manually to the filter. However, there are more rules written by the domain admins that should be followed. Those rules can be found under a standard text file named Robots Exclusion Protocol (robots.txt).

"robots.txt" acts as the selected filename for implementing the Robots Exclusion Protocol, which is a widely adopted standard employed by websites to signal to web crawlers and other web robots the specific sections of the website that are permissible for them to access [10]. The "robots.txt" can be fetched at the starting point of crawling and can be cached through the whole crawling process, as it can be assumed it will not change during the crawling process. This assumption is still better than making an HTTP request to get the robots.txt file for each URL that needs to be fetched, as this will duplicate the number of requests and reduce the crawler efficiency and also load the server with unwanted requests. Including the robots.txt in the crawling, process should be mandatory as this will serve the point about politeness mentioned in the Crawler specifications section.

3.4.3 Web Crawler Types

It is essential to understand that although all web crawlers' main goal is to crawl pages from the internet, however, there are different types and categories that some crawlers fall to. The first category is **Universal or Broad crawler**. This category of web crawlers doesn't confine itself to webpages of a specific topic or domain; instead, they continuously traverse links without limitations, collecting all encountered webpages. The most significant search engines use this type of crawler, such as Google and Bing, this is understandable as these search engines' main aim is to make the entire web searchable, and they try to fetch all kinds of pages and contents. The second category is called **Preferential crawler (Focused crawler)**. Focused crawlers target specific topics, themes, or domains. They are designed to gather information from a particular niche or subject area, providing specialized search results. In this thesis, a Focused crawler has been implemented and used. The last category is **Hidden Web crawlers (Deep Web Crawlers)**. Deep web crawlers target databases and content hidden behind web forms. They can interact with online databases and retrieve information that general search engines might miss [11].

3.4.4 Challenges and issues

Researchers encounter a variety of challenges when working with different crawlers implementation. A compilation of these challenges is presented below.

- The scale of the web: The web is vast and virtually infinite, so crawlers must prioritize which pages to crawl and which to skip to use resources efficiently.
- Content Changes: Webpages can change frequently, requiring crawlers to revisit and reindex content to ensure freshness and accuracy.
- Blocking and IP Bans: Some websites may block or ban crawlers' IP addresses if they perceive them as causing too much traffic or disruption. Crawlers need to manage IP addresses to avoid being blocked.
- Nonuniform structures: The Web is dynamic and uses inconsistent data structures, as there is no universal norm to build a website. Due to lack of uniformity, collecting data becomes difficult. The problem is amplified when crawler has to deal with semistructured and unstructured data.
- Error handling: Crawlers may encounter broken links, leading to errors and incomplete indexing. Handling broken links requires additional processing. Moreover, the internet connection may disconnect, and the crawler may stop crawling.

- Crawlers traps: Some websites intentionally create spider/crawlers traps to make crawlers go into an infinite loop or redirect them in different directions. Calendar Traps and Infinite URL Parameters are examples of spider traps.

- Politeness and Ethical Concerns: Crawlers must be programmed to be polite and respectful to websites' server resources. Aggressive crawling can overload servers, leading to ethical concerns and potential website blocking. This might be simple, but the main challenge is that each domain uses different Firewall settings. One server might allow the crawler to make five requests per second; the other will block the crawler. There are no hardcoded rules to follow; however, following the robots.txt file might help improve the Politeness of the crawler.

3.5 Indexing

In a search engine, an indexer is a component responsible for analyzing and organizing the content of web pages or documents in order to create an index, which is a structured database that enables efficient and fast retrieval of relevant information during search queries. When a search engine's crawler or web spider gathers data from websites, the indexer processes this data by breaking down the content into smaller units like words, phrases, and metadata. It then associates these units with the URLs or documents they came from. This organized information is stored in the index, which serves as a map or reference for the search engine to quickly locate and present relevant results when a user makes a search query. The indexer plays a crucial role in improving the speed and accuracy of search results because it precomputes and structures the data in a way that enables the search engine to match queries with indexed content more efficiently. Although supporintg indexing is fundemantal in this thesis however it will be given less attention than the crawling component.

3.5.1 Inverted index

Crawlers collect information from the web and prepare them to be searched. However, looking up each term with brute forcing is a performance issue and is impossible. Hence inverted index data structure is used. 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 compared to traditional databases, where documents are associated with their content. In an inverted index, each unique term in the collection of documents is treated as a key, and the value associated with each term is a list of references to the documents where that term appears. This list of references allows for rapid access to all the documents containing a specific term.

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

3.5.2 Document Unit

The term "document" has frequently been mentioned in reference to the specific information intended for retrieval from a webpage. While, in most instances, this term encompasses the entirety of a page's content, this holds true primarily for Universal crawlers. However, in the case of the Preferential crawler employed in this thesis, the definition of a document unit is adaptable, contingent upon the nature

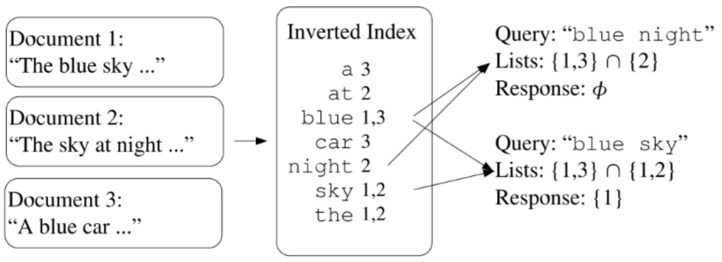


Figure 3: An illustration of an inverted index featuring three documents. All tokens are included in this example, and the sole text normalization applied is converting all tokens to lowercase. Queries that involve multiple keywords are resolved using set operations. [3]

of the website and the specific data the user seeks to gather. For instance, in an E-commerce website featuring product titles, prices, and descriptions, the document unit may be viewed as a single product listing. Conversely, a news website might treat each news article as an individual document. More comprehensive guidance on creating a template that corresponds to a document will be provided in the Approach chapter.

3.5.3 Tokenization

Tokenization in indexing is the process of breaking down a text document or a string of text into smaller units called tokens. These tokens are typically words or subwords, and they serve as the basic units for indexing and searching within a text corpus. Tokenization is a fundamental step in natural language processing and information retrieval tasks.

Input: The University of Freiburg Output: The University of Freiburg

3.6 Ranking

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

$$BM25\_score = tf^{*}.\log_2(\frac{N}{df}) \tag{3.4}$$

$$tf^{*} = \frac{tf.(k+1)}{k.\alpha + tf} \tag{3.5}$$

$$\alpha = \frac{1 - b + b.DL}{AVDL} \tag{3.6}$$

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

The following example dives into the details of the BM25 equation and how it impacts ranking. Table [] shows a list of documents as an example of an input to be indexed and ranked against different search queries. We start by calculating the variables needed to find the BM25 scores for each term in a document.

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

Examining Table [], we can note that the tokens that appear in all three documents, like 'the', 'of' and 'is' have scores of 0. If the term searched for is common, we

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

Table 1: Documents sample

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

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

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

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

3.6.1 Ranking Refinements

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

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

am, are, is $\Rightarrow$ be
car, cars, car's, cars' $\Rightarrow$ car

Stemming typically involves employing a simple rule-based approach to truncate word endings, aiming to achieve accurate results in most cases. This approach often involves eliminating derivational affixes. In contrast, lemmatization follows a more meticulous process that involves utilizing vocabulary and morphological analysis of words. The primary objective is to exclusively remove inflectional endings and to restore words to their fundamental or dictionary forms, which are referred to as lemmas [5].

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

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

3.6.2 Fuzzy search

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

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

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

Considering two character strings, s1 and s2, the edit distance that separates them represents the minimal count of edit operations needed to transform s1 into s2. The typical edit operations permitted for this purpose encompass: (i) the insertion of a character into a string, (ii) the deletion of a character from a string, and (iii) the replacement of a character within a string by another character. In the context of these operations, the term "Levenshtein distance" is sometimes used interchangeably with edit distance. To illustrate, the edit distance between "cat" and "dog" is 3. It's worth noting that the concept of edit distance can be extended to encompass varying weights assigned to different types of edit operations. For instance, assigning a greater weight to the replacement of the character "s" with "p" compared to its replacement with "a" (with the latter being physically closer on the keyboard) can be explored. This weight assignment strategy, dependent on the probability of letter substitutions, proves highly effective in practical scenarios. Nonetheless, the subsequent discussion primarily concentrates on scenarios where all edit operations bear identical weights [5].

Fuzzy search can be used with q-gram to find how similar words are. For a string x, and an integer q $\in$ N, the multiset of q-grams, denoted by Q q(x), , consists of all substrings of length q 3("freiburg") = "fre", "rei", "eib", "ibu", "bur", "urg" We define it as a multiset because the same q-gram may occur multiple times and we want to know when it does

Q 3("ababa") = "aba", "bab", "aba"

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

|Q q(x)| = |x| - q + 1

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

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

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

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

4 Approach

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

4.1 Software Architecture

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

- **Frontend (Angular & PrimeNG)**: Positioned closest to the user, this component encompasses all pages and views. Angular is leveraged in conjunction with the contemporary CSS library, PrimeNG. To communicate with the backend, it employs the REST API.

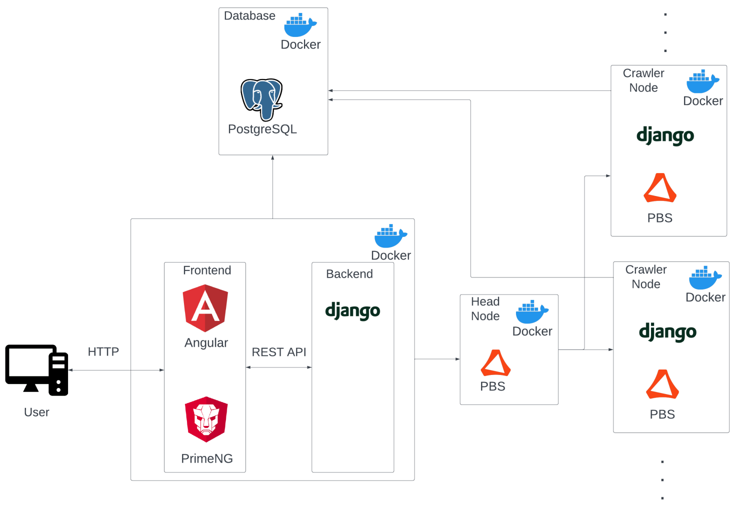


Figure 4: High-level view of the software archtiticture.

- **Backend (Django)**: Serving as the core intelligence of the search engine, the backend houses both the crawler and indexer modules. It facilitates interaction with the Head node to initiate crawling based on user-defined configurations. Moreover, it establishes a connection with PostgreSQL for the storage of crawler and indexer configurations, along with job-related information.
- **Head Node (PBS)**: Operating as the central hub, this node orchestrates job management and determines the allocation of tasks to Crawler nodes, which are responsible for traversing the specified websites.
- **Crawler Node (PBS)**: These instances are designated to execute the crawling process and store the resulting data in the PostgreSQL database.

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

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

4.2 Crawler Implmentation

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

Algorithm 1 Start Crawling

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

A seed URL is added to the current thread queue. The seed URL represents the starting point for the crawling, and the user configuration defines it. Using the sead url, the robots.txt content is downloaded once and can be reused for the rest of the crawling process.

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

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

4.2.1 Links Data Structure

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

4.2.2 Practical Challenges

- **Avoiding Loops**: Looping is when a web crawler repeatedly visits and requests the same web pages or URLs in a never-ending cycle, often resulting in excessive traffic to the same content. This is problematic as it wastes resources can also be inefficient, and can prevent the crawler from continuing. The first method to prevent looping is to record all the URLs visited and crawled. Before making a new request, we check if the URL is in this list. If it is, skip crawling it again to prevent loops. The second method is to use URL normalization. Normalize URLs by removing unnecessary components such as query parameters, fragments, or trailing slashes. This helps ensure that URLs with different representations (e.g., with and without a trailing slash) are treated as the same URL.

- **Duplicated Content**: While the same web crawler avoids revisiting identical URLs to prevent content duplication, it's important to note that identical content may exist in different URLs paths within the same website. For instance, a men's shoe might be accessible via various links like "/winter/shoes/", "/men/shoes/", or "/sales/shoes/". Relying solely on the URL as a unique identifier to prevent content duplication is not foolproof. A more effective approach involves comparing the content itself with the database after parsing. Instead of a straightforward content check against the database, which can pose performance challenges, we employ a more efficient method. We generate a unique hash code using the SHA-1 hashing algorithm based on the content string intended for storage. This hash code is then stored in the database. Before saving any new content, we can verify if the hash code already exists in the database. This method ensures content uniqueness, even when it appears under different URLs on the same site, without the computational overhead of directly comparing lengthy content strings in the database.

- **Dynamic Content**:Crawling dynamic websites presents a distinct set of challenges compared to static websites. Dynamic sites generate content on the client side through technologies like JavaScript, adding complexity to the task of accessing and extracting data. A primary concern lies in uncovering concealed content that necessitates user interaction. For instance, certain websites hide lengthy content portions, revealing them only upon clicking a "read more" button. Additionally, most websites implement lazy loading, fetching content on-demand via AJAX requests. To address these challenges, Selenium establishes a genuine session and fully renders the webpage. This approach allows for emulating user interactions using action chains, which simulate actions such as waiting, scrolling, and clicking. More details regarding this can be found in the User Interface section.

- **Termination Conditions**: Crawlers can be brought to a halt by establishing specific criteria to ensure termination. The initial criterion involves defining a maximum depth, which restricts the number of page transitions to a single level. Additionally, monitoring and restricting the total count of visited pages and collected documents is possible. Another method is to employ a wall time measurement to monitor the crawler's runtime duration and trigger an abort if the crawler exceeds the expected time frame.

- **Avoiding DOS**: Increasing the number of requests and expanding the crawler's capacity by adding more threads or nodes may seem enticing to boost performance. However, this approach carries a significant risk of overwhelming the targeted servers, potentially resulting in Distributed Denial of Service (DDoS) or Denial of Service (DoS) attacks. Servers can perceive this surge in requests as an attack, which could lead to the crawler being blacklisted and subsequently banned. To mitigate this risk, it's crucial to introduce a waiting period between each request made by the same crawler. Additionally, when using Selenium, a deliberate delay of at least one second or more is already integrated to allow for the complete rendering of web pages. Nevertheless, these precautions alone may not prevent users from adding more nodes and executing DDoS attacks. Consequently, it is strongly advisable to exercise cautious management by monitoring and regulating the number of threads and nodes. This approach demonstrates respect for the targeted servers and helps prevent overloading them.

4.3 Indexer Implmentation

The indexing phase follows the crawling step, as it necessitates the downloading of documents. Unlike crawling, which can be computationally intensive, indexing is relatively lightweight. Hence, it is carried out on the localhost with multithreading support. Algorithm [2] clarifies the indexing process. Firstly, the algorithm loads the user-defined indexing configurations. You can find more details about these configurations in the User Interface Design section. Subsequently, an empty inverted list is initialized. An inverted list is a data structure used to associate words or terms with the documents in which they appear. In Python, it can be implemented as a dictionary (Dict[str, list[int]]), where each word serves as a key, and the corresponding value is a list of document IDs containing that word.

Algorithm 2 Create Inverted List

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

The user specifies three variables: "threshold" and "stop_list," which are retrieved from the database. The "threshold" represents the minimum word length required for tokenization from a document. The "stop_list" is a predefined list of terms that should be omitted from the indexing process. The algorithm then iterates through all the documents, performing the following steps for each one: Initializes the document length as a counter, initially set to zero. Tokenizes the document to obtain a list of words. Iterates through the word list, checking each word's length against the threshold and verifying if it is not in the stop_list. If these conditions are met, the word is added to the inverted list, and the document length counter is incremented by one.

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

4.4 User Interface Design

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

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

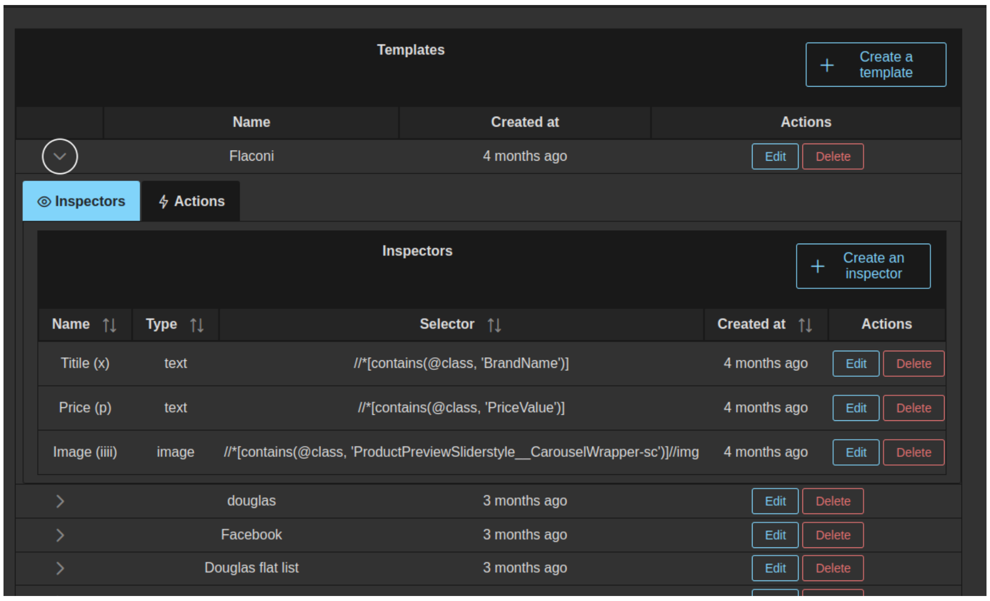
- **Name**: This denotes the inspector's identifier, such as "Price" or "Title."
- **Selector**: It encompasses the XPath expression pinpointing the chosen element, for instance, //*[contains(@class, 'product-title')].
- **Type**: This can assume values like "Text," "Link," or "Image," signifying the nature of the content to be extracted.

- **Variable Name (*Optional*)**: This is an optional shorthand representation of the selector, facilitating its use during the indexing process to enhance search results (Ranking).
- **Clean-up Expression List (*Optional*)**: Here, you can specify a regular expression utilized to refine the extracted value from the inspector. This proves beneficial in eliminating unwanted noise. A use case for this can be to remove the currency when extracting a price.
- **Attribute (*Optional*)**: This field allows you to specify an HTML element attribute, such as "src," "name," or "href," as an optional parameter.

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

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

Once you've set up the crawler with the appropriate configurations tailored to the specific website you're targeting, the next step is to create a job referred to as a "runner." Multiple runners can be associated with each crawler, allowing them to run on different days or machines. It's important to note that each runner can utilize 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 must be online. By default,



Name	Created at	Actions
Flaconi	4 months ago	Edit Delete

Name	Type	Selector	Created at	Actions
Titile (x)	text	//*[contains(@class, 'BrandName')]	4 months ago	Edit Delete
Price (p)	text	//*[contains(@class, 'PriceValue')]	4 months ago	Edit Delete
Image (iiii)	image	//*[contains(@class, 'ProductPreviewSliderstyle__CarouselWrapper-sc')]//img	4 months ago	Edit Delete

Name	Created at	Actions
douglas	3 months ago	Edit Delete
Facebook	3 months ago	Edit Delete
Douglas flat list	3 months ago	Edit Delete

Figure 5: An overview of the Templates table containing inspectors list projecting the targeted feilds in a document

"localhost" is set as the value, which can be employed for testing purposes and for circumventing the PBS cluster.

It's crucial to keep a close eye on the crawler runner to monitor its performance and configuration effectiveness before finalizing it. This proactive approach helps save time and guarantees that the crawler is collecting the accurate data. The "runners" table provides a straightforward and informative progress overview through four primary status indicators. The initial status is "New," representing the runner's initialization before the crawling process begins. "Running" signifies that the crawling process is currently underway. "Exit" indicates a status change that occurs when an error occurs, leading to the termination of the crawling process. Finally, "Completed" marks the last status, indicating that the runner has finished its task and exited. During the crawling process, various statistics about the runner are gathered, including information such as the total number of visited pages, the average number of documents discovered per page, and the various HTTP status codes encountered. One can retrieve the documents collected by the runner by selecting

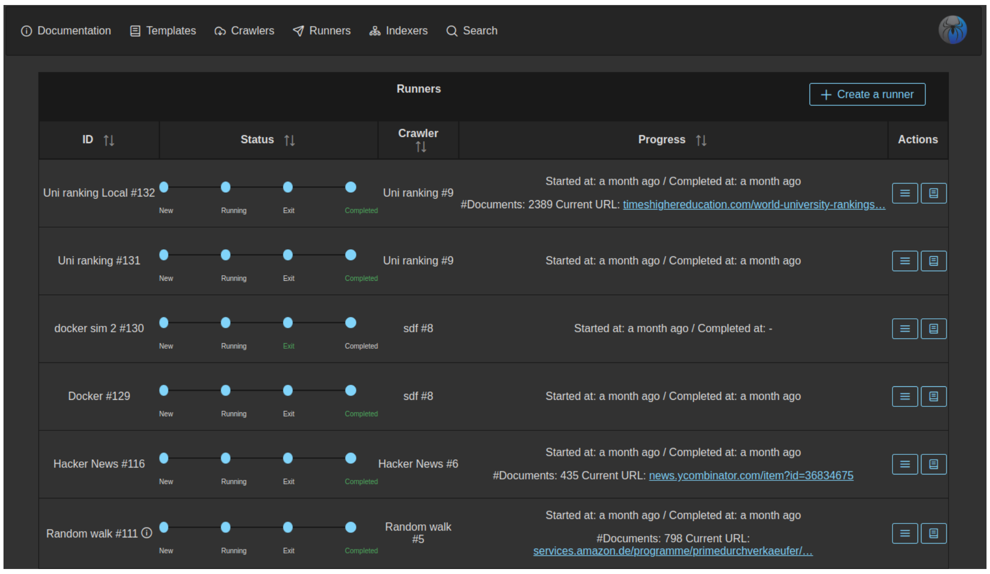
Attribute	Description
Name	A user-defined identifier for the crawler
Template	The template utilized by the crawler to recognize HTML elements for crawling and storage
Max pages	The upper limit for the number of pages to be visited during the crawling process
Max collected docs	The maximum number of documents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

Table 3: Crawler configurations options

the "Download CSV" option from the actions dropdown menu.

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

As previously mentioned, indexers are responsible for handling documents, each of which contains a string field suitable for indexing. In this context, inspectors come



Runners

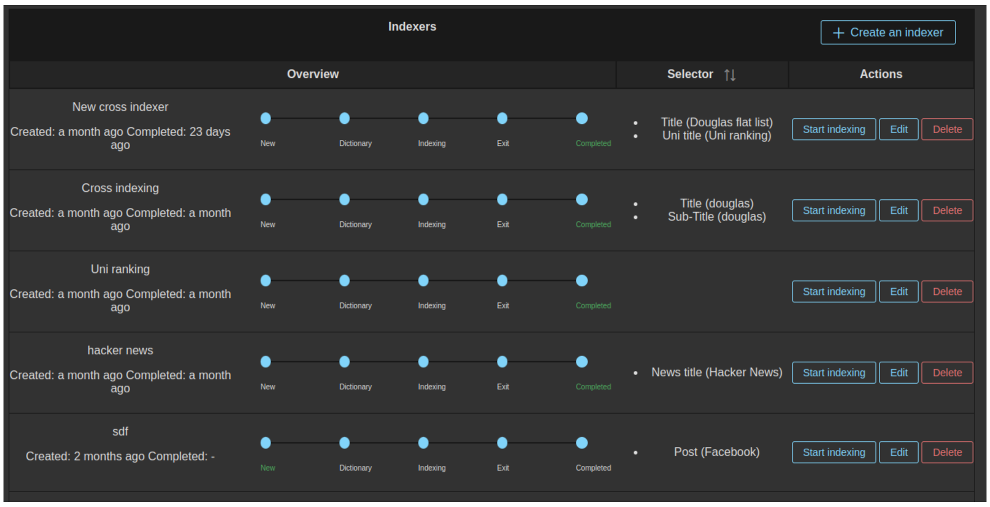
ID	Status	Crawler	Progress	Actions
Uni ranking Local #132	New Running Exit Completed	Uni ranking #9	Started at: a month ago / Completed at: a month ago #Documents: 2389 Current URL: timeshighereducation.com/world-university-rankings...	
Uni ranking #131	New Running Exit Completed	Uni ranking #9	Started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

Figure 6: An overview of the runners table, showing the runners status and progress.

into play. These inspectors are responsible for mapping the fields extracted from the document, such as Name, Title, Price, and Image. Users have the option to select which inspectors or fields to index from a dropdown menu, with the choice limited to only text fields, images and links are excluded from this selection.

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

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

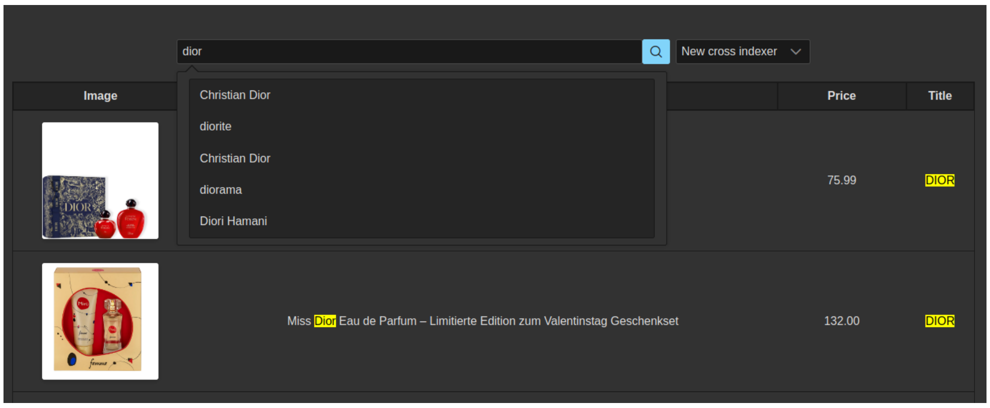

Indexers

Create an indexer

Overview		Selector	Actions
New cross indexer Created: a month ago Completed: 23 days ago	New · Dictionary · Indexing · Exit · Completed	Title (Douglas flat list) Uni title (Uni ranking)	Start indexing · Edit · Delete
Cross indexing Created: a month ago Completed: a month ago	New · Dictionary · Indexing · Exit · Completed	Title (douglas) Sub-Title (douglas)	Start indexing · Edit · Delete
Uni ranking Created: a month ago Completed: a month ago	New · Dictionary · Indexing · Exit · Completed		Start indexing · Edit · Delete
hacker news Created: a month ago Completed: a month ago	New · Dictionary · Indexing · Exit · Completed	News title (Hacker News)	Start indexing · Edit · Delete
sdf Created: 2 months ago Completed: -	New · Dictionary · Indexing · Exit · Completed	Post (Facebook)	Start indexing · Edit · Delete

Figure 7: An overview of the indwxwea table, showing the their status and progress.

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


dior

New cross indexer

Christian Dior
diorite
Christian Dior
diorama
Diori Hamani

Image		Price	Title
		75.99	DIOR
	Miss Dior Eau de Parfum – Limitierte Edition zum Valentinstag Geschenkset	132.00	DIOR

Figure 8: High-level view of the software archtiticture.

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

Table 4: Indexer configurations options

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

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

5 Evaluation

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

5.1 Testing Environment

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

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

Table 5: Local machine setup

5.2 Crawler

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

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

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

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

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

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

5.2.1 Datasets

Evaluating a web crawler necessitates the utilization of a static website as a reliable reference point. The crawler-test[1] website serves as an excellent choice for this purpose due to its diverse range of content and links, containing a wide range of scenarios that a crawler might encounter. This website effectively employs robots.txt to provide guidance to the crawler, allowing for an assessment of its politeness. Moreover, it includes a section containing links that yield various HTTP request status codes, such as 4xx and 5xx, which proves valuable for ensuring the crawler's robustness. Additionally, it incorporates multiple instances of page redirection, including scenarios like infinite redirection, which serves the dual purpose of evaluating the crawler's ability to avoid traps and enhancing its overall resilience.

To enhance the coverage and versatility of our crawler testing, we are considering two additional websites encompassing a broader range of use cases, ensuring that the crawler can effectively handle various HTML structures and more generic scenarios.

[1]`https://crawler-test.com/`

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. We will categorize our testing into three distinct datasets: small (with 100 products), medium (with 1,000 products), and large (with 10,000 products) to evaluate the crawler's performance under different data sizes thoroughly. The second website, Times Higher Education[3], specializes in annually ranking universities. This unique characteristic allows us to evaluate content changes occurring on a yearly basis, setting it apart from other websites that undergo daily alterations, which can prove challenging for accurate evaluation.

5.2.2 Experiments

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

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

Table 6 displays the testing configurations used in the experiment. The Seed URL serves as the initial point from which the crawler begins its procedure, set to the root path of the crawler-test. The "Allow Multi Elements" checkbox is disabled (set to False) because the objective is not to gather a list of documents; each page contains a single text field. The Max Pages parameter is configured to a limit of 500, ensuring that the crawler does not exceed this number of pages. This figure can be adjusted based on the website's size to be crawled; for instance, smaller websites with around 50 pages may require a lower limit. The number of threads is set to the

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

default value of 1. We have set the Max Depth to 1 to enable easier coverage testing. This choice allows for easier comparison between the number of visited pages and the number of URLs discovered in the site's root path, which, upon simple page inspection, contains 415 links. Since the expected maximum number of pages is 415, the Max Docs parameter can be constrained to 500. The inspectors are set to target the content of each page; thus, one inspector is only needed. No automated actions, such as scrolling or waiting, are necessary for this use case; therefore, they can be left. Any properties not explicitly mentioned can be left at their default settings.

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

Table 6: Crawler configuration

After creating a runner that runs, starts the crawling process and is completed, the result of the crawler should be similar to the one shown in Table 7. Looking at the Links row, we can note that the crawler found 406 out of the expected 415 links. This is normal as the crawler will normalize all the found links, including skipping the link fragments, duplicated links and any broken links. 402 pages out of 406 found links are crawled correctly. The other four pages are categorized as Cross Site links, meaning they do not belong to the Seed URL hostname crawler-test.com. This is important to evaluate to ensure that the crawler stays focused, does not jump to sites out of the intended scope, and does not spend valuable resources. Already Visited links is a counter that checks how many times the crawler found a link that has already been visited and skipped, in this case, 0. When no multithreading is used, and the Max Depth is only set to 1, the Already Visited is expected to be

0 because the duplicated links will be already excluded in the normalizing process before starting to crawl.

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

Table 7: Crawler configuration

Evaluating the performance of a web crawler is challenging as it depends on different aspects, such as the page's size and how fast the page loads. Moreover, if the site uses pagination, this can add extra waiting time to render the rest of the content. However, some valuable matrices can be helpful to give a good insight like those shown in the Time row. The total time spent to crawl 406 pages took approximately 11 minutes. To give a better perspective, the enterprise solution Parsehub [4] the free teer without IP Rotation, can crawl 200 pages in 40 minutes, and the Standard expensive teer with IP Rotation that costs $189 can crawl 200 pages in 10 minutes. This means crawling the 406 pages inside the crawler-test will take 20 minutes. Comparing the crawler with the Standard tier, it is faster by 2X. Note that in this evaluation, the performance can be increased by using more threads or nodes to distribute the loads, which we will evaluate.

```html
<body>
  <h1 id="h1"></h1>
  <p id="par"></p>
```

[4]https://www.parsehub.com/pricing

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

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

Improving the average page rendering time (0.679) is achievable by avoiding using rendering engines like Selenium. Processing a simple HTTP request is often quicker than rendering the entire page. However, it's essential to note that the rendering step is crucial in handling dynamic content. For instance, consider the dynamically inserted text, indicated by the link [5]. This link is a straightforward example to illustrate the significance of the rendering process for web crawlers. Using a simple wget command in Linux, you can download the content depicted in 5.1. This content reveals that the two HTML tags, h1 and p, lack the inner text that the crawler can collect as a document. Although there's a script tag designed to replace the innerHTML for each tag with text, without a rendering engine, the JavaScript logic remains unexecuted. Consequently, gathering the inner text of these tags becomes an impossibility. On the other hand, 5.2 shows the same page content after rendering where both tags are updated, and both contain the right content that the crawler can see and download. While this is a simplified example, it's easy to visualise more complicated websites employing advanced JavaScript frameworks and libraries for client-side rendering. This complexity increases the rendering time due to the

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

execution of all JavaScript logic and the subsequent updating of the HTML DOM. Given that the crawler's objective is to locate and collect all HTML content, striking a balance between efficiency and comprehensiveness is a challenge that must be addressed.

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

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

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

The "Docs Content" section provides information regarding the collected and downloaded content. The "Tot. Docs found" metric indicates that 255 documents have been successfully downloaded. Among these, 25 documents are duplicates. This duplication is intentional, as this testing site contains repetitive information designed to verify the functionality of this feature. The "Avg. Docs per page" value is 1, indicating that the site typically presents one document per page. The presence of the "Robots.txt Exists" flag covers the commitment to polite crawling behaviour. The flag is set to True when this file is located and successfully downloaded. It's essential to emphasize the importance of respectful crawling and commitment to the Robots.txt file protocol. Lastly, the "Tot. Errors" metric accounts for various Selenium exceptions[6] that may arise during the crawling process. As mentioned, many errors are expected since this is a testing site with different edge cases, and not terminating under those conditions is a good sign.

Checking if changing thew depth will increase the coverage.

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

Unfortunately, knowing exactly how many links the crawler-test site contains is challenging. However, since we have evaluated how the crawler behaves when the Max

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

Depth is, one can give some assurance on the rest of the results. The table 9 shows 895 links has been found which was expected as the depth of the crawler has increased from 1 to 10.

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

Table 8: Crawler configuration

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

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

Multi threading

It is crucial to assess the scalability of the crawler, especially since it employs intelligent multi-threading for link sharing. We must thoroughly test it. We will use

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

Table 9: Crawler configuration

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

Table 10: Crawler configuration

the same configurations detailed in Table 8 but adjust the threads parameter and monitor the overall time spent. The results are presented in Figure 9, illustrating the outcomes of eight different runs using identical crawler settings while varying the number of threads. When we switch from one thread to two, we observe a reduction in time from 722 seconds to 595 seconds, representing a 17.5% improvement, increasing the threads to four results in a time of 438 seconds, which is a 39.33% improvement compared to using just one thread. However, further increasing the threads to six or eight does not yield any additional performance enhancements; in

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

Table 11: Crawler configuration

fact, it begins to have a detrimental effect, causing the time spent to increase. This is primarily because all threads communicate to share unvisited links and avoid revisiting links already processed by other threads. Consequently, as the number of threads increases, the communication overhead escalates. Users must fine-tune this parameter until they find the optimal setting, which appears to be four threads.

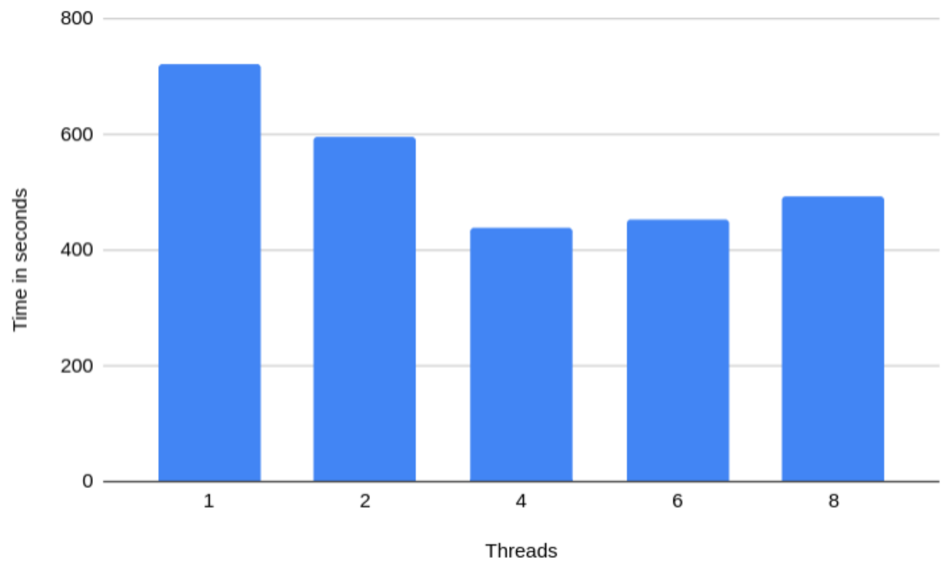


Figure 9: Comparing threads performance.

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

Figure 10 displays the results of four runs, with each run showcasing a distinct distribution of crawled links among threads. While the ideal scenario would involve each thread handling 25% of the discovered links, the averages in the columns reveal variations. Thread 4 crawls more than 25%, while Thread 1 crawls less, which is normal due to considerations like communication overhead and thread-specific conditions. Additionally, threads won't split links if their queue contains fewer than five links, further affecting equal distribution.

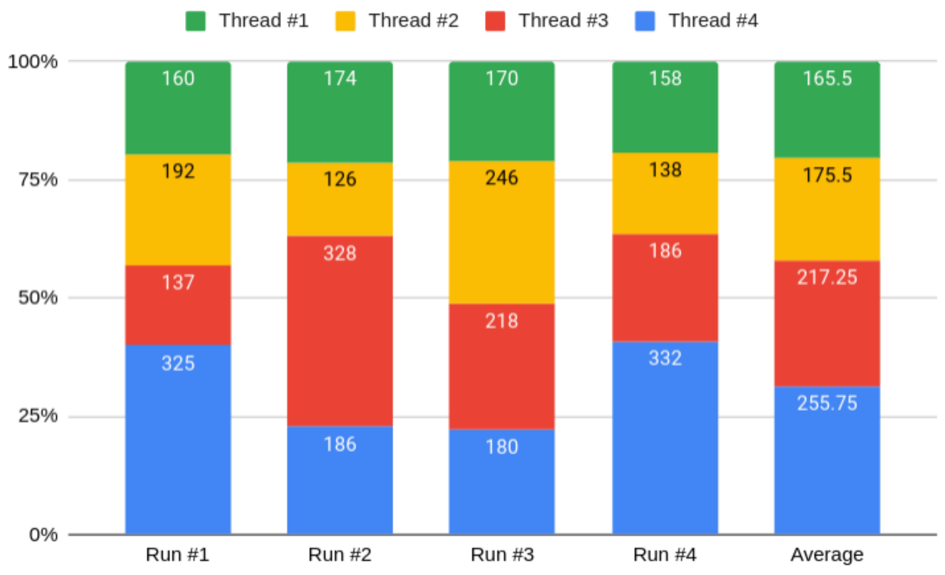


Figure 10: Threads documents distribution.

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

The collected and visited links are accurate and match the expected count of 94.

Seed URL	https://www.timeshighereducation.com/world-university-rankings/2023/world-ranking
Allow Multi Elements	True
Max Pages	100
Threads	1
Max Depth	100
Pagination	//*[contains(@class, 'pagination')]
Actions	None
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')]
Max Docs	2350

Table 12: Crawler configuration

The fact that there are zero already visited links indicates that no duplicate URLs have been encountered. The total time spent during this process is approximately 8 minutes, which is significantly shorter compared to a similar test conducted using ParseHub, which took 20 minutes to complete all 94 pages. Interestingly, even though the crawler successfully parsed and collected the results correctly, it consistently received a 403[7] status code in response to all HTTP requests instead of the expected 200. This issue may be attributed to the website's use of Cloudflare service, as suggested by the information available at ScrapeOps[8] . In this particular use case, the number of collected documents representing universities in the table matches the total results displayed in the page pagination, totalling 2345. It is worth noting that the website also deploys a Robots.txt file, which the crawler successfully detected and used.

The following use case involves extracting a specific category of products from an e-commerce website. In this scenario, we will gather various types of data, including text and images. You can find the crawler configurations for this task in Table

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

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

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

Table 13: Crawler configuration

14, designed to scrape a particular product category. Given that the Seed URL link's pagination indicates the presence of 7 pages, we can configure the Max Pages and Max Docs parameters to be 10. We can employ multithreading to increase performance by setting the number of threads to 4. Douglas employs lazy loading for image loading, which, in turn, causes the crawler to fetch only 30 out of the available 48 products on the page. To address this issue, we can utilize the actions tab to introduce a scrolling action, repeating it ten times until we reach the bottom of the page. The ability to configure automated actions depends on the website's functionality. For instance, if the website experiences prolonged loading times, we can include a waiting action to account for the estimated waiting time. If the website necessitates a clicking action to reveal more information, the click action can be utilized for that purpose. The inspectors in this context encompass four fields: brand, title, price, and product images.

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

Seed URL	https://www.douglas.de/de/c/parfum/damenduefte/duftsets/010111
Allow Multi Elements	True
Max Pages	10
Threads	4
Max Depth	10
Pagination	//*[contains(@class, 'pagination')]
Actions	Scolling down 10 times

Inspectors	//*[contains(@class, 'top-brand')]
	//a[contains(@class, 'product-tile__main-link')]/div[1]/div/img
	//*[contains(@class, 'text')][contains(@class, 'name')]
	//div[contains(concat(' ', normalize-space(@class), ' '), ' price-row ')]

Max Docs	1000

Table 14: Crawler configuration

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

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

It is important to note that, despite employing the robots.txt file for politeness and ensuring a relatively low crawler request rate (calculated as the number of threads divided by the Average Page Rendering, yielding 1.516 requests per second, which is relatively low and unlikely to overload the server), the IP address was eventually banned, and access to the site was blocked after several attempts. This highlights that each website may have its own unique security implementation based on its firewall[9] rules and the reverse proxy[10] it uses.

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

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

Table 15: Crawler configuration

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

Another use case involved crawling Stackoverflow questions, focusing solely on the "python" tag in the seed URL to retrieve Python-related questions. The configured inspectors collected question titles, descriptions, and vote counts. Initially, running the crawler with four threads led to a website ban after only ten pages were crawled.

[9] 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.

[10] 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.

To resolve this, I reduced the thread count to one, reducing the number of requests and resolving the issue.

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

Table 16: Crawler configuration

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

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

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

Table 17: Crawler configuration

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

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

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

Table 18: Crawler configuration

5.3 Indexer

5.4 User Experience

6 Conclusions and Future Work

Bibliography

[KimathiKimathi2020] Kimathi2020Kimathi, G. 2020June. What Is Ray Tracing Technology and How It Works in GPUs. What is ray tracing technology and how it works in gpus. https://www.dignited.com/62084/how-ray-tracing-works