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

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Statement 2

1.3 Contribution 4

1.4 Chapter Overview 5

2 Related Work **7**

2.1 High Level Google Architecture 7

2.2 Web Crawlers 9

3 Background **13**

3.1 The nature of the web 13

3.2 History 14

3.3 Web Search Engine 15

3.3.1 Requierments and Features 16

3.4 Cralwer 17

3.4.1 Cralwer Specifications 17

3.4.2 Crawler architecture 18

3.4.3 Web Crawler Types 20

3.4.4 Challenges and issues 21

3.5 Indexing 22

3.5.1 Inverted index 23

3.6 Ranking 23
3.6.1 Ranking Refinements 26
3.6.2 Fuzzy search 27

4 Approach **31**
4.1 Software Architecture 31
4.2 Crawler Implmentation 33
4.2.1 Links Data Structure 35
4.2.2 Practical Challenges 35
4.3 Indexer Implmentation 37
4.4 Storage Handling 37
4.5 User Interface Design 37

5 Evaluation **39**

6 Conclusions and Future Work **41**

Bibliography **45**

List of Figures

1 High level view of Google web crawlers archeticture. 8

2 The basic crawler architecture. 19

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] 24

4 High-level view of the software archtiticture. 32

List of Tables

1 Documents sample 25

2 The first 10 tokens of the result inverted index and the scores of the docuemnts. 25

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].

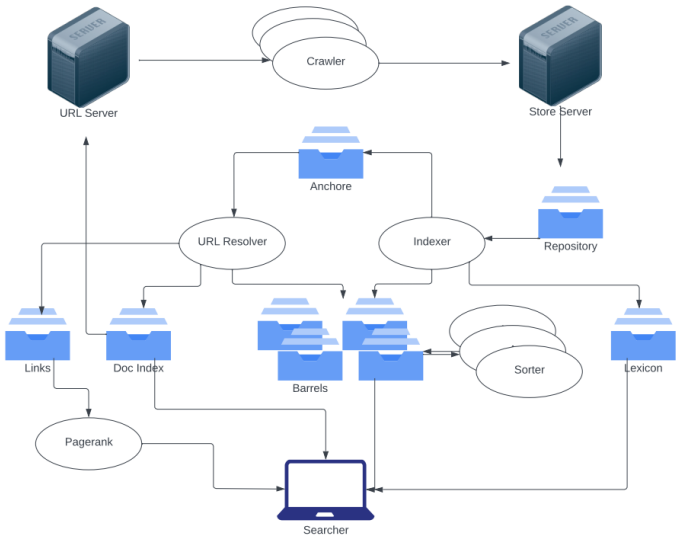


Figure 1: High level view of Google web crawlers archeticture.

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.

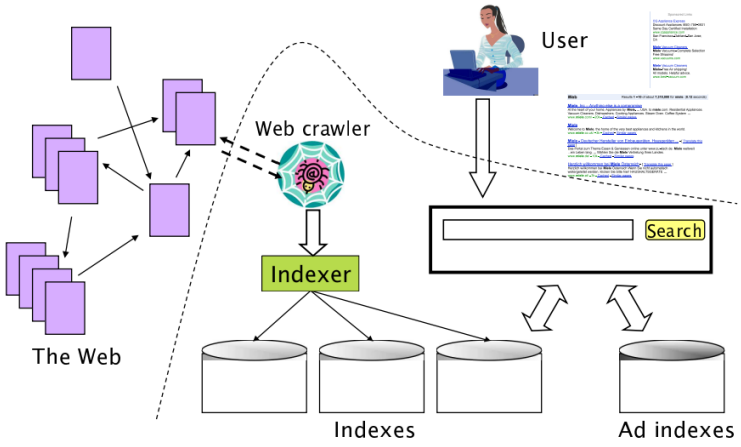
$$\lambda_p \approx \frac{-N_p \log(1 - X_p/N_p)}{S_p} \tag{3.3}$$

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].

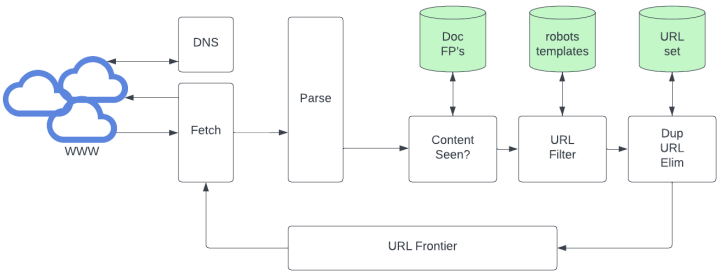


Figure 2: The basic crawler architecture.

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.

3.6 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

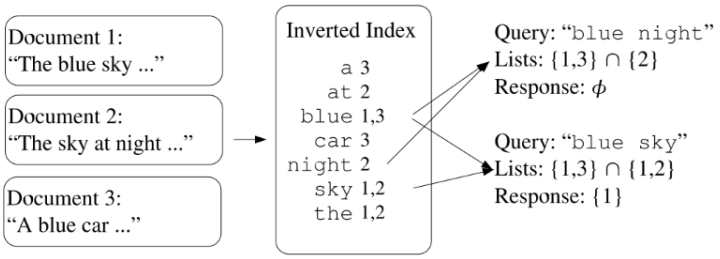


Figure 3: An illustration of an inverted index featuring three documents. All tokens are included in this example, and the sole text normalization applied is converting all tokens to lowercase. Queries that involve multiple keywords are resolved using set operations. [3]

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:

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

Table 1: Documents sample

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

Token	(Doc. ID, BM25 Score)
the	[(1, 0.0), (2, 0.0), (3, 0.0)]
university	[(1, 1.0715011288144682), (3, 0.46664430652429906)]
of	[(1, 0.0), (2, 0.0), (3, 0.0)]
freiburg	[(1, 1.0715011288144682), (2, 0.975283837792052)]
officially	[(1, 1.7407388463458566)]
albert	[(1, 1.7407388463458566)]
ludwig	[(1, 1.7407388463458566)]
is	[(1, 0.0), (2, 0.0), (3, 0.0)]
a	[(1, 0.6424549149886022), (3, 0.8859381868218585)]
public	[(1, 1.7407388463458566)]

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)| ≤ 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.

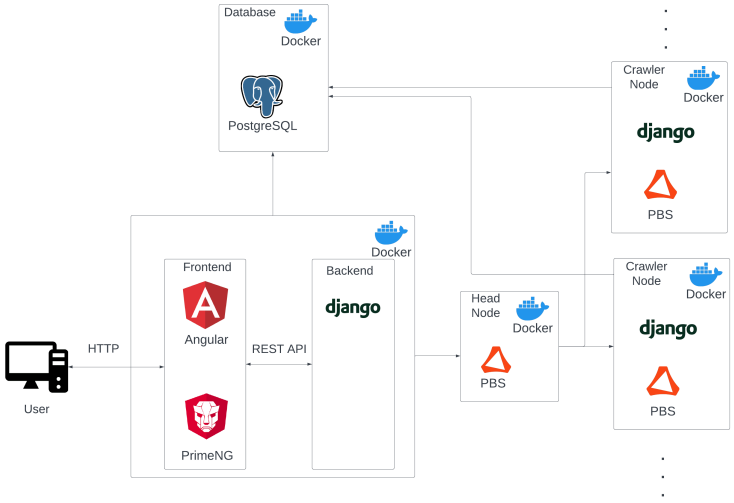


Figure 4: High-level view of the software archtiticture.

- **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.

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.

Algorithm 1 Start Crawling

```
load_crawler_configurations()
thread ← create_threads_pool()
urls_queue ← get_thread_urls_queue(thread))
seed_url ← get_seed_url()
add_url_to_queue(urls_queue, seed_url)
robot_file ← get_robot_file_content()
while urls_queue not empty or all threads not done do
    if urls_queue is empty then
        urls_queue ← get_thread_urls_queue(thread)
    else
        current_url ← urls_queue_next_url()
        filter_unwanted_urls(current_url)
        request_page(current_url)
        execute_automated_actions()
        find_next_urls_and_add_them_to_urls_queue()
        docs ← find_and_download_targeted_documents()
        filter_unwanted_documents(docs)
    end if
end while
```

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**:

4.3 Indexer Implmentation

4.4 Storage Handling

4.5 User Interface Design

5 Evaluation

Evaluating a web crawler can be challenging as it is usually made of different components, like the web crawler and the indexer. However, in this section, one crucial aspect of the evaluation is evaluating the overall design and architecture. As introduced in the conference of [6], the following issues made the distributed parallel crawlers challenging:

- Overlap: Overlap happens when numerous simultaneous processes engage in downloading pages, potentially resulting in multiple instances of the same page being fetched by different processes. This situation arises because one process might not detect if another process has already visited the page. This results in redundant downloaded pages that should conserve network bandwidth and enhance the overall efficiency of the crawler. The question then arises: how can we effectively orchestrate these processes to mitigate overlap?

- Quality: A crawler frequently prioritizes downloading "important" pages to optimize the overall quality of the gathered content. However, within a parallel crawling environment, individual processes might need a comprehensive overview of the results of the other processes. Consequently, each process could base its crawling choices solely on its limited web perspective, potentially leading to suboptimal decisions.

- Communication bandwidth: As already explained that the crawlers try to increase their quality. They must communicate with each other. However, for a large number of processes, the communication overhead can be problematic. One should reduce the communication to only the critical information that can increase the quality with minimizing the communication overhead.

- Scalability: Those targets can contain millions of pages, even focused crawlers that only crawl one site, like Reddit, Amazon or StackOverflow. As mentioned, in certain cases, a single-process crawler cannot achieve the required download rate for huge sites. Hence scalability is necessary to overcome this challenge.

- Coverage: denoted as c/u, is represented by the ratio of pages crawled (c) to the overall pages (u) explored by the entire crawler. In an error-free scenario, an optimal coverage would be 1. No duplicated pages will be stored and visited, increasing efficiency [5].

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